1. (Mostly) Public Domain Korn Shell. The Korn shell, ksh, is a command interpreter that operates interactively or by reading a saved script. This version of ksh is the port of pdksh which was released with OpenBSD 7.0, converted to use literate programming.

In this first iteration the goal is to make as few changes to the C code itself as possible in order to maintain a clear link between this document's source and the original OpenBSD source code. Unfortunately some changes have had to be made but they are localised and/or trivial. These are described in the next section.

This document has been laid out so that concepts are introduced after sufficient groundwork has been laid to understand them and before later parts which require in turn that they be understood, which is completely different to the order a C compiler expects it in. However rather than produce a single C source file with perhaps some helpers this `CWEB` document produces the same `c` and `h` files as were present in the OpenBSD sources, after having the code processed by `CTANGLE`.

It is unfortunate that I cannot find a means of vertically aligning a sequence of comments.

2. Changes From OpenBSD 7.0. Comments are the biggest change from the original. Comments are not ignored but parsed by TEX so they had to be at least sanitised of all TEX/CWEB instructions. CWEB always associates a comment with the code that came before it, trying wherever possible to include it on the same line after a small gap, and so comments cannot be used as the title for a block of code.

In fact comments are often themselves part of the narrative so they have been chopped and changed with wild abandon: Some are used as the title of CWEB sections, some have been moved into a section's descriptive text, some simply removed. Those that have been kept more or less as-is have been sanitised, edited and moved to the now 'correct' place.

The other major change was to rename some identifiers where CWEB had trouble with conflicting types of the same symbol, for example the function "include" has been renamed to *Include* so that it doesn't clash with the C-preprocessor directive **include**. The complete list is:

* include: *Include*.
* define: *Define*.
* struct block: **struct Block**.
* struct cclass: **struct CClass**.
* struct coproc: **struct Coproc**.
* struct env: **struct Env**.
* struct limits: **enum Limits**.
* struct op: **struct Op**.
* struct prec: **enum Prec**.
* struct shf: **struct Shf**.
* struct source: **struct Source**.
* struct temp: **struct Temp**.
* token (**static** function in `expr.c`): *Mtoken*. TODO: NO!

Although the source files generated from this document should match their original counterpart the header files are bare of preprocessor definitions: these are all together in `sh.h` instead.

3. main.c. CWEB has the concept of a "primary" output file who's name matches the name of the input file—two in fact, the C source and the TEX source. In our case the input file is named ksh.w and there was no ksh.c so CTANGLE should be invoked as "ctangle ksh.w - main.c" or the resulting ksh.c ranemed to main.c (or the Makefile changed). In the CWEB source snippets which belong in main.c are preceeded by plain @c while the rest are preceeded by the filename, eg. @<sh.h@)=.

```
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <paths.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
  extern char **environ;
  static void reclaim(void);
  static void remove_temps(struct Temp *tp);
  static int is_restricted(char *name);
  static void init_username(void);
  ⟨ Global variables 5 ⟩
```

4. ⟨ Shared function declarations 4 ⟩ ≡

```
  int Include(const char *, int, char **, int);
  int command(const char *, int);
  int shell(Source *volatile , int volatile);
  void unwind (int) __attribute__((__noreturn__));
  void newenv(int);
  void quitenv(struct Shf *);
  void cleanup_parents_env(void);
  void cleanup_proc_env(void);
```

See also sections 10, 70, 104, 243, 413, 462, 555, 638, 667, 742, 813, 864, 883, 1159, 1190, 1203, 1204, and 1294.

This code is used in section 7.

5. These variables are widely used and so are made visible everywhere.

⟨ Global variables 5 ⟩ ≡

```
  const char *kshname;      /* $0 */
  pid_t kshpid;      /* $$, shell pid */
  pid_t procpid;      /* pid of executing process */
  uid_t ksheuid;      /* effective uid of shell */
  int exstat;      /* exit status */
  int subst_exstat;      /* exit status of last $( ... )/` ... ` */
  const char *safe_prompt;      /* safe prompt if $PS1 substitution fails */
  char username[_PW_NAME_LEN + 1];      /* username for \u prompt expansion */
  int disable_subst;      /* disable substitution during evaluation */
```

See also sections 18, 33, 61, 72, 109, 152, 161, 183, 188, 241, 263, 537, 635, 657, 674, 679, 743, and 860.

This code is used in section 3.

6. ⟨ Externally-linked variables 6 ⟩ ≡

```
extern const char *kshname;
extern pid_t kshpid;
extern pid_t procpid;
extern uid_t ksheuid;
extern int exstat;
extern int subst_exstat;
extern const char *safe_prompt;
extern char username[];
extern int disable_subst;
```

See also sections 14, 19, 34, 49, 73, 82, 110, 153, 162, 184, 206, 242, 264, 315, 538, 636, 658, 662, 675, 680, 744, 861, 865, 1202, and 1328.

This code is used in section 7.

7. Shared Header & Utilities. `sh.h` is used by all parts of ksh to import the C environment.

```
#define NELEM(a) (sizeof (a)/sizeof ((a)[0]))
#define BIT(i) (1 ≪ (i))  /* define bit in flag */
#define NUFILE 32  /* Number of user-accessible files */
#define FDBASE 10  /* First file usable by Shell */
#define BITS(t) (CHAR_BIT * sizeof (t))
#define LINE 4096  /* input line size */
⟨ sh.h 7 ⟩ ≡
#include "config.h"  /* system and option configuration info */
  ⟨ Common headers 8 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Define MAGIC 767 ⟩
  ⟨ Type definitions 17 ⟩
  ⟨ Externally-linked variables 6 ⟩
  ⟨ Detect compilation of a debugging build 12 ⟩
#include "shf.h"
#include "table.h"
#include "tree.h"
#include "expand.h"
#include "lex.h"
  ⟨ Shared function declarations 4 ⟩
```

8. ⟨ Common headers 8 ⟩ ≡

```
#include <limits.h>
#include <setjmp.h>
#include <stdarg.h>
#include <stddef.h>
#include <signal.h>
#include <stdbool.h>
```

This code is used in section 7.

9. Some things are used broadly accross ksh' codebase but have no obvious position to fit in within this document. They are collected in `misc.c` along with some things which are not so miscellaneous (and *do* fit somewhere logical).

⟨ `misc.c` 9 ⟩ ≡
```
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "sh.h"
#include "charclass.h"

⟨ misc.c variables 48 ⟩
⟨ misc.c declarations 52 ⟩

static int dropped_privileges;
static int do_gmatch(const unsigned char *, const unsigned char *, const unsigned char *, const
    unsigned char *);
```
See also sections 20, 35, 37, 43, 50, 51, 53, 54, 56, 102, 209, 210, 211, 212, 213, 214, 215, 216, 219, 279, 280, 306, 308, 329, 614, 615, 620, 621, 622, 746, and 807.

10. ⟨ Shared function declarations 4 ⟩ +≡
```
void setctypes(const char *, int);
void initctypes(void);
char *u64ton(uint64_t, int);
char *str_save(const char *, Area *);
char *str_nsave(const char *, int, Area *);
int option(const char *);
char *getoptions(void);
void change_flag(enum sh_flag, int, int);
int parse_args(char **, int, int *);
int getn(const char *, int *);
int bi_getn(const char *, int *);
int gmatch(const char *, const char *, int);
int has_globbing(const char *, const char *);
const unsigned char *pat_scan(const unsigned char *, const unsigned char *, int);
void qsortp(void **, size_t, int(*)(const void *, const void *));
int xstrcmp(const void *, const void *);
void ksh_getopt_reset(Getopt *, int);
int ksh_getopt(char **, Getopt *, const char *);
void print_value_quoted(const char *);
void print_columns(struct Shf *, int, char *(*)(void *, int, char *, int), void *, int, int prefcol);
int strip_nuls(char *, int);
int blocking_read(int, char *, int);
int reset_nonblock(int);
char *ksh_get_wd(char *, int);
```

11. Conditional Compilation. Little used; mainly checks that at least one of `EMACS` or `VI` is defined.

```
⟨ config.h 11 ⟩ ≡
#ifndef CONFIG_H
#define CONFIG_H
#if 0 /* Strict POSIX behaviour? */
#undef POSIXLY_CORRECT
#endif
#if 0 /* Specify default $ENV? */
#undef DEFAULT_ENV
#endif
#if ¬defined (EMACS) ∧ ¬defined (VI)
# error "Define␣either␣EMACS␣or␣VI."
#endif
#endif /* CONFIG_H */
```

12. Internal Debugging. ksh can be built with debugging enabled by compiling with `-DKSH_DEBUG`.

```
⟨ Detect compilation of a debugging build 12 ⟩ ≡
#ifdef KSH_DEBUG
#define kshdebug_init( ) kshdebug_init_( )
#define kshdebug_printf (a) kshdebug_printf_a
#define kshdebug_dump(a) kshdebug_dump_a
#else /* KSH_DEBUG */
#define kshdebug_init( )
#define kshdebug_printf (a)
#define kshdebug_dump(a)
#endif /* KSH_DEBUG */
```

This code is used in section 7.

13. Version. The version of the shell is available in the `$KSH_VERSION` or `$SH_VERSION` variable depending on how the shell was started (see below). The value of this is always the contents of *ksh_version*.

⟨ `version.c` 13 ⟩ ≡
```
#include "sh.h"
  const char ksh_version[] ← "@(#)PD␣KSH␣v5.2.14␣99/07/13.2";
```

14. ⟨ Externally-linked variables 6 ⟩ +≡
```
  extern const char ksh_version[];
```

15. Global Initialisation. The main function in `main.c` is *main*. This takes care of initialising everything and then runs the shell itself in *shell*, which never returns.

```
static char **make_argv(int, char **);      /* TODO: wart */
int main(int argc, char *argv[])
{
  int i;
  int argi;
  Source *s;
  struct Block *l;
  int restricted, errexit;
  char **wp;
  struct Env env;
  pid_t ppid;
  kshname ← argv[0];
  ⟨ Restrict potential system calls 16 ⟩
  ainit(&aperm);      /* Initialise permanent Area */
  ⟨ Set up the base environment 185 ⟩
  initio();      /* Initialise I/O first so output routines (eg. errorf, shellf) can work */
  initvar();      /* Initialise special variable table */
  initctypes();      /* Initialise character type lookup database */
  inittraps();      /* Initialise signal handlers and traps */
  coproc_init();      /* Initialise coprocess data structures */
  ⟨ Set up variable and command dictionaries 186 ⟩
  initkeywords();      /* Define shell keywords */
  ⟨ Define built-in commands 187 ⟩
  init_histvec();      /* Initialise the history vector */
  ⟨ Set the global $PATH variable 193 ⟩
  ⟨ Set default flag values 23 ⟩
  ⟨ Import the calling process' environment 189 ⟩
  ⟨ Figure out $PWD 191 ⟩
  ⟨ Store initial $PPID & $(K)SH_VERSION 192 ⟩
  ⟨ Execute initialisation statements 62 ⟩
  ⟨ Set $USER and the main prompt 190 ⟩
  ⟨ Detect privilege early 25 ⟩
  ⟨ Parse command line arguments 26 ⟩      /* s is discovered here */
  ⟨ POSIXise standard input 31 ⟩
  ⟨ Initialise job control 673 ⟩
  ⟨ Initialise interactive editing 63 ⟩
  ⟨ Create a copy of argv 59 ⟩
  ⟨ Read the profile file(s) 65 ⟩
  ⟨ Initialise (non-)interactivity 66 ⟩
  shell(s, true);      /* doesn't return */
  return 0;
}
```

16. The *pledge* utility restricts the system calls that a process can make to a limited subset. Given ksh' purpose the system calls it requires are nearly all of those which are unrelated to networking.

```
⟨ Restrict potential system calls 16 ⟩ ≡
  if (issetugid()) {      /* could later drop privileges */
    if (pledge("stdio␣rpath␣wpath␣cpath␣fattr␣flock␣getpw␣proc␣""exec␣tty␣id", Λ) ≡ −1) {
      perror("pledge");
      exit(1);
    }
  }
  else {
    if (pledge("stdio␣rpath␣wpath␣cpath␣fattr␣flock␣getpw␣proc␣""exec␣tty", Λ) ≡ −1) {
      perror("pledge");
      exit(1);
    }
  }
```

This code is used in section 15.

17. The behaviour of ksh is influenced by flags which are be set by command-line options and in most cases later toggled at runtime.

The order of the flags in this list **must** match the order in the global *sh_options* array.

⟨ Type definitions 17 ⟩ ≡

```
enum sh_flag {
  FEXPORT ← 0,      /* -a: export all */
  FBRACEEXPAND,     /* enable {...,...} globbing */
  FBGNICE,      /* bgnice */
  FCOMMAND,     /* -c: (invocation) execute specified command */
  FCSHHISTORY,      /* csh-style history enabled */
#ifdef EMACS
  FEMACS,       /* emacs command editing */
#endif
  FERREXIT,     /* -e: quit on error */
#ifdef EMACS
  FGMACS,       /* gmacs command editing */
#endif
  FIGNOREEOF,       /* eof does not exit */
  FTALKING,     /* -i: interactive */
  FKEYWORD,     /* -k: "name=value" anywhere */
  FLOGIN,       /* -l: a login shell */
  FMARKDIRS,        /* mark directories with "/" in file name completion */
  FMONITOR,     /* -m: job control monitoring */
  FNOCLOBBER,       /* -C: don't overwrite existing files */
  FNOEXEC,      /* -n: don't execute any commands */
  FNOGLOB,      /* -f: don't do file globbing */
  FNOHUP,       /* -H: don't kill running jobs when login shell exits */
  FNOLOG,       /* don't save functions in history (ignored) */
  FNOTIFY,      /* -b: asynchronous job completion notification */
  FNOUNSET,     /* -u: using an unset variable is an error */
  FPHYSICAL,        /* -o physical: don't do logical cds/pwds */
  FPIPEFAIL,        /* -o pipefail: all commands in a pipeline can affect $? */
  FPOSIX,       /* -o posix: be POSIXly correct */
  FPRIVILEGED,      /* -p: use /etc/suid_profile */
  FRESTRICTED,      /* -r: restricted shell */
  FSH,      /* -o sh: favor sh-like behaviour */
  FSTDIN,       /* -s: (invocation) parse standard input */
  FTRACKALL,        /* -h: create tracked aliases for all commands */
  FVERBOSE,     /* -v: echo input */
#ifdef VI
  FVI,      /* VI-like command editing */
  FVIRAW,       /* always read in raw mode (ignored) */
  FVISHOW8,     /* display chars with 8^th bit set as-is (versus "M-") */
  FVITABCOMPLETE,       /* enable tab as file name completion char */
  FVIESCCOMPLETE,       /* enable escape as file name completion in command mode */
#endif
  FXTRACE,      /* -x: execution trace */
  FTALKING_I,       /* (internal): initial shell was interactive */
  FNFLAGS       /* (place holder: how many flags are there) */
};
```

See also sections 32, 47, 71, 84, 88, 97, 105, 106, 107, 218, 226, 237, 262, 271, 316, 331, 332, 364, 414, 433, 566, 573, 634, 656, 670, 671, 777, 859, 888, 1005, 1024, 1036, 1111, 1160, 1163, 1164, 1165, 1169, 1191, 1199, 1276, 1284, 1296, 1297, 1299, and 1340.

This code is cited in section 897.

This code is used in section 7.

18. Each flag is represented by an **int** in *shell_flags*, its value accessed with *Flag*.

```
#define Flag(f) (shell_flags[(int)(f)])
⟨ Global variables 5 ⟩ +≡
  char shell_flags[FNFLAGS];
```

19.

```
⟨ Externally-linked variables 6 ⟩ +≡
  extern char shell_flags[FNFLAGS];
```

20. Change a flag value and take care of special actions.

```
⟨ misc.c 9 ⟩ +≡
  void change_flag(enum sh_flag f, int what,      /* flag to change */
  int newval)      /* what is changing the flag (command line vs set) */
  {
    int oldval;
    oldval ← Flag(f);
    Flag(f) ← newval;
    if (f ≡ FMONITOR) {
      if (what ≠ OF_CMDLINE ∧ newval ≠ oldval) j_change();
    }
    else ⟨ Emacs or VI flags 21 ⟩
    else if (f ≡ FPRIVILEGED ∧ oldval ∧ ¬newval ∧ issetugid() ∧ ¬dropped_privileges) {
          /* Turning off -p? */
        gid_t gid ← getgid();
        setresgid(gid, gid, gid);
        setgroups(1, &gid);
        setresuid(ksheuid, ksheuid, ksheuid);
        if (pledge("stdio␣rpath␣wpath␣cpath␣fattr␣flock␣getpw␣proc␣""exec␣tty", Λ) ≡ −1)
          bi_errorf("pledge␣fail");
        dropped_privileges ← 1;
      }
      else if (f ≡ FPOSIX ∧ newval) { Flag(FBRACEEXPAND) ← 0; }
    if (f ≡ FTALKING) {      /* Changing interactive flag? */
      if ((what ≡ OF_CMDLINE ∨ what ≡ OF_SET) ∧ procpid ≡ kshpid) Flag(FTALKING_I) ← newval;
    }
  }
```

21. ⟨ Emacs or VI flags 21 ⟩ ≡

```
  if (0
#ifdef VI
   ∨ f ≡ FVI
#endif      /* VI */
#ifdef EMACS
   ∨ f ≡ FEMACS ∨ f ≡ FGMACS
#endif      /* EMACS */
  ) {
    if (newval) {
#ifdef VI
      Flag(FVI) ← 0;
#endif      /* VI */
#ifdef EMACS
      Flag(FEMACS) ← Flag(FGMACS) ← 0;
#endif      /* EMACS */
      Flag(f) ← newval;
    }
  }
```

This code is used in section 20.

22. Some flags are initialiased specially before parsing the command line arguments. The first of these is FNOHUP which in the original ksh was preceeded by this comment, included verbatim, which has been in place since at least as far back as 1996.

⟨ Temporarily enable nohup by default for three decades 22 ⟩ ≡

```
  Flag(FNOHUP) ← 1;      /* Turn on nohup by default for now—will change to off by default once people
      are aware of its existence (AT&T ksh does not have a nohup option—it always sends the hup). */
```

This code is used in section 23.

23. Other flags which are initialised before querying the command line arguments are just how POSIX to be and the interactive editing mode.

FBRACEEXPAND: Turn on brace expansion by default. AT&T ksh's that have alternation always have it on but POSIX doesn't have brace expansion, so set this before setting up FPOSIX (*change_flag*( ) clears FBRACEEXPAND when FPOSIX is set).

FPOSIX: Set just before importing the environment so that it will have exactly the same effect as the POSIXLY_CORRECT environment variable. If this needs to be done sooner to ensure correct POSIX operation, an initial scan of the environment will also have done sooner.

⟨ Set default flag values 23 ⟩ ≡

```
  ⟨ Temporarily enable nohup by default for three decades 22 ⟩
  Flag(FBRACEEXPAND) ← 1;
#ifdef POSIXLY_CORRECT
  change_flag(FPOSIX, OF_SPECIAL, 1);
#endif      /* POSIXLY_CORRECT */
  ⟨ Differentiate between sh and ksh 24 ⟩
#if defined (EMACS)
  change_flag(FEMACS, OF_SPECIAL, 1);
#endif      /* EMACS */
#if defined (VI)
  Flag(FVITABCOMPLETE) ← 1;
#endif      /* VI */
```

This code is used in section 15.

24. Ksh operates subtly differently if started from a binary named plain `sh`. To indicate this it raises the `FSH` flag and adjusts the word pointed to by *version_param* from its default "`KSH_VERSION`" to "`SH_VERSION`".

```
⟨ Differentiate between sh and ksh 24 ⟩ ≡
  if (¬strcmp(kshname, "sh") ∨ ¬strcmp(kshname, "-sh") ∨
        (strlen(kshname) ≥ 3 ∧ ¬strcmp(&kshname[strlen(kshname) − 3], "/sh"))) {
    Flag(FSH) ← 1;
    version_param ← "SH_VERSION";
  }
```

This code is used in section 23.

25. The `FPRIVILEGED` flag (`-p`) is also set before parsing command line arguments if the current user is privileged.

```
⟨ Detect privilege early 25 ⟩ ≡
  Flag(FPRIVILEGED) ← getuid( ) ≠ ksheuid ∨ getgid( ) ≠ getegid( );
```

This code is used in section 15.

26. Other flags are initialised en masse by *parse_args*. Each flag is normally set to 1 if raised so `FMONITOR` is set to 127 first to detect whether it's set explicitely on the command line (`-m`).

```
⟨ Parse command line arguments 26 ⟩ ≡
  Flag(FMONITOR) ← 127;
  argi ← parse_args(argv, OF_CMDLINE, Λ);
  if (argi < 0) exit(1);
  if (Flag(FCOMMAND)) {⟨ Initialise to run a command string 27 ⟩}      /* -c */
  else if (argi < argc ∧ ¬Flag(FSTDIN)) {⟨ Initialise to read from a file 28 ⟩}      /* -s */
  else {⟨ Initialise to read from standard input 29 ⟩}
```

This code is used in section 15.

27.

```
⟨ Initialise to run a command string 27 ⟩ ≡
  s ← pushs(SSTRING, ATEMP);
  if (¬(s→start ← s→str ← argv[argi++])) errorf("-c␣requires␣an␣argument");
  if (argv[argi]) kshname ← argv[argi++];
```

This code is used in section 26.

28.

```
⟨ Initialise to read from a file 28 ⟩ ≡
  s ← pushs(SFILE, ATEMP);
  s→file ← argv[argi++];
  s→u.shf ← shf_open(s→file, O_RDONLY, 0, SHF_MAPHI | SHF_CLEXEC);
  if (s→u.shf ≡ Λ) {
    exstat ← 127;      /* POSIX */
    errorf("%s:␣%s", s→file, strerror(errno));
  }
  kshname ← s→file;
```

This code is used in section 26.

29. ⟨ Initialise to read from standard input 29 ⟩ ≡
```
Flag(FSTDIN) ← 1;
s ← pushs(SSTDIN, ATEMP);
s→file ← "<stdin>";
s→u.shf ← shf_fdopen(0, SHF_RD | can_seek(0), Λ);
if (isatty(0) ∧ isatty(2)) {⟨ Initialise interactivity 30 ⟩}
```
This code is cited in section 815.

This code is used in section 26.

30. ⟨ Initialise interactivity 30 ⟩ ≡
```
Flag(FTALKING) ← Flag(FTALKING_I) ← 1;      /* -i */
  /* The following only if isatty(0): */
s→flags |= SF_TTY;
s→u.shf→flags |= SHF_INTERRUPT;
s→file ← Λ;
```
This code is used in section 29.

31. "This bizarreness is mandated by POSIX". The bizarreness being to, if standard input is a character device and the shell is interactive, clear its non-blocking flag if it's set. Bizarre.

⟨ POSIXise standard input 31 ⟩ ≡
```
{
  struct stat s_stdin;
  if (fstat(0, &s_stdin) ≥ 0 ∧ S_ISCHR(s_stdin.st_mode) ∧ Flag(FTALKING)) reset_nonblock(0);
}
```
This code is used in section 15.

32. Option Parsing.

```
#define GF_ERROR BIT(0) /* call errorf if there is an error */
#define GF_PLUSOPT BIT(1) /* allow "+c" as an option */
#define GF_NONAME BIT(2) /* don't print argv[0] in errors */
#define GI_MINUS BIT(0) /* an option started with "-" */
#define GI_PLUS BIT(1) /* an option started with "+" */
#define GI_MINUSMINUS BIT(2) /* arguments were ended with "--" */
```

⟨ Type definitions 17 ⟩ +≡

```
typedef struct {
  int optind;
  int uoptind; /* what user sees in $OPTIND */
  char *optarg;
  int flags; /* see GF_* */
  int info; /* see GI_* */
  unsigned int p; /* 0 or index into argv[optind − 1] */
  char buf[2]; /* for bad option $OPTARG value */
} Getopt;
```

33. ⟨ Global variables 5 ⟩ +≡

```
Getopt builtin_opt; /* for shell builtin commands */
Getopt user_opt; /* parsing state for getopts builtin command */
```

34. ⟨ Externally-linked variables 6 ⟩ +≡

```
extern Getopt builtin_opt;
extern Getopt user_opt;
```

35. Initialize a Getopt structure.

⟨ misc.c 9 ⟩ +≡

```
void ksh_getopt_reset(Getopt *go, int flags)
{
  go→optind ← 1;
  go→optarg ← Λ;
  go→p ← 0;
  go→flags ← flags;
  go→info ← 0;
  go→buf[1] ← '\0';
}
```

36. It's not clear why this should be in c_ksh.c.

⟨ KSH commands 36 ⟩ ≡

```
void getopts_reset(int val)
{
  if (val ≥ 1) {
    ksh_getopt_reset(&user_opt, GF_NONAME | (Flag(FPOSIX) ? 0 : GF_PLUSOPT));
    user_opt.optind ← user_opt.uoptind ← val;
  }
}
```

See also sections 1230, 1236, 1237, 1243, 1249, 1250, 1260, 1265, 1268, 1269, 1270, 1271, 1277, 1279, and 1280.

This code is used in section 1201.

37. Interface to *getopt* for shell built-in commands, the `getopts` command, and command line options.

A leading "`:`" in options means don't print errors, instead return "`?`" or "`:`" and set `$OPTARG` to the offending option character.

If `GF_ERROR` is set (and option doesn't start with "`:`"), errors result in a call to *bi_errorf*.

Non-standard features:

* "`;`" is like "`:`" in options, except the argument is optional (if it isn't present, `$OPTARG` is set to 0). Used for "`set -o`".

* "`,`" is like "`:`" in options, except the argument always immediately follows the option character (`$OPTARG` is set to the null string if the option is missing). Used for "`read -u2`", "`print -u2`" and "`fc -40`".

* "`#`" is like "`:`" in options, expect that the argument is optional and must start with a digit or be the string "`unlimited`". If the argument doesn't match, it is assumed to be missing and normal option processing continues (`$OPTARG` is set to 0 if the option is missing). Used for "`typeset -LZ4`" and "`ulimit -adunlimited`".

* accepts `+c` as well as `-c` if the `GF_PLUSOPT` flag is present. If an option starting with "`+`" is accepted the `GI_PLUS` flag will be set in *go→info*.

```
⟨ misc.c 9 ⟩ +≡
  int ksh_getopt(char **argv, Getopt *go, const char *options)
  {
    char c;
    char *o;
    if (go→p ≡ 0 ∨ (c ← argv[go→optind − 1][go→p]) ≡ '\0') {⟨ (Re-)start argument parsing 38 ⟩}
    go→p++;
    if (c ≡ '?' ∨ c ≡ ':' ∨ c ≡ ';' ∨ c ≡ ',' ∨ c ≡ '#' ∨ ¬(o ← strchr(options, c)))
    {⟨ Flag is special or invalid 39 ⟩}
    if (*++o ≡ ':' ∨ *o ≡ ';') {⟨ Look for a possibly optional argument 40 ⟩}
    else if (*o ≡ ',') {⟨ Look for an attached argument 41 ⟩}
    else if (*o ≡ '#') {⟨ Look for a numeric argument 42 ⟩}
    return c;
  }
```

38. ⟨ (Re-)start argument parsing 38 ⟩ ≡

```
  char *arg ← argv[go→optind], flag ← arg ? *arg : '\0';
  go→p ← 1;
  if (flag ≡ '-' ∧ arg[1] ≡ '-' ∧ arg[2] ≡ '\0') {      /* arg is exactly "--" */
    go→optind++;
    go→p ← 0;
    go→info |= GI_MINUSMINUS;
    return −1;
  }
  if (arg ≡ Λ ∨
        ((flag ≠ '-') ∧ (¬(go→flags & GF_PLUSOPT) ∨ flag ≠ '+')) ∨
        (c ← arg[1]) ≡ '\0') {      /* nb. always sets c or returns */
    go→p ← 0;
    return −1;      /* not a - (nor a + if allowed) */
  }
  go→optind++;
  go→info &= ∼(GI_MINUS | GI_PLUS);
  go→info |= flag ≡ '-' ? GI_MINUS : GI_PLUS;
```

This code is used in section 37.

39. ⟨ Flag is special or invalid 39 ⟩ ≡

```
if (options[0] ≡ ':') {
  go→buf[0] ← c;
  go→optarg ← go→buf;
}
else {
  warningf(true, "%s%s-%c: unknown option",
      (go→flags & GF_NONAME) ? "" : argv[0],
      (go→flags & GF_NONAME) ? "" : ": ", c);
  if (go→flags & GF_ERROR) bi_errorf(Λ);
}
return '?';
```

This code is used in section 37.

40. : indicates required, ; optional.

⟨ Look for a possibly optional argument 40 ⟩ ≡

```
if (argv[go→optind − 1][go→p]) go→optarg ← argv[go→optind − 1] + go→p;
else if (argv[go→optind]) go→optarg ← argv[go→optind++];
else if (∗o ≡ ';') go→optarg ← Λ;
else {
  if (options[0] ≡ ':') {
    go→buf[0] ← c;
    go→optarg ← go→buf;
    return ':';
  }
  warningf(true, "%s%s-'%c' requires argument",
      (go→flags & GF_NONAME) ? "" : argv[0],
      (go→flags & GF_NONAME) ? "" : ": ", c);
  if (go→flags & GF_ERROR) bi_errorf(Λ);
  return '?';
}
go→p ← 0;
```

This code is used in section 37.

41. ⟨ Look for an attached argument 41 ⟩ ≡

```
go→optarg ← argv[go→optind − 1] + go→p;
go→p ← 0;
```

This code is used in section 37.

42. "`unlimited`" is also acceptable.

```
⟨ Look for a numeric argument 42 ⟩ ≡
  if (argv[go→optind − 1][go→p]) {
    if (digit(argv[go→optind − 1][go→p]) ∨ ¬strcmp(&argv[go→optind − 1][go→p], "unlimited")) {
      go→optarg ← argv[go→optind − 1] + go→p;
      go→p ← 0;
    }
    else go→optarg ← Λ;
  }
  else {
    if (argv[go→optind] ∧ (digit(argv[go→optind][0]) ∨ ¬strcmp(argv[go→optind], "unlimited"))) {
      go→optarg ← argv[go→optind++];
      go→p ← 0;
    }
    else go→optarg ← Λ;
  }
```

This code is used in section 37.

43. Parse command line & `set`-command arguments. Returns the index of non-option arguments or -1 if there is an error.

```
⟨ misc.c 9 ⟩ +≡
  int parse_args(char **argv, int what,       /* OF_CMDLINE or OF_SET */
  int *setargsp)
  {
    static char cmd_opts[NELEM(sh_options) + 3];      /* "o:" (plus '\0') */
    static char set_opts[NELEM(sh_options) + 5];      /* "Ao;s" (plus '\0') */
    char *opts;
    char *array ← Λ;
    Getopt go;
    int i, optc, sortargs ← 0, arrayset ← 0;
    unsigned int ele;
    if (cmd_opts[0] ≡ '\0') {⟨ Build option strings 44 ⟩}
    if (what ≡ OF_CMDLINE) {
      char *p;
      Flag(FLOGIN) ← (argv[0][0] ≡ '-' ∨ ((p ← strrchr(argv[0], '/')) ∧ *++p ≡ '-'));
        /* Set FLOGIN here so the user can clear it with "+l" (ell). */
      opts ← cmd_opts;
    }
    else opts ← set_opts;
    ksh_getopt_reset(&go, GF_ERROR | GF_PLUSOPT);
    ⟨ Scan arguments 45 ⟩
    if (arrayset ∧ (¬*array ∨ *skip_varname(array, false))) {
      bi_errorf("%s: is not an identifier", array);
      return −1;
    }
    if (sortargs) {
      for (i ← go.optind; argv[i]; i++) ;
      qsortp((void **) &argv[go.optind], (size_t)(i − go.optind), xstrcmp);
    }
    if (arrayset) {
      set_array(array, arrayset, argv + go.optind);
      for ( ; argv[go.optind]; go.optind++) ;
    }
    return go.optind;
  }
```

44. ⟨ Build option strings 44 ⟩ ≡

```
char *p, *q;
strlcpy(cmd_opts, "o:", sizeof cmd_opts);
p ← cmd_opts + strlen(cmd_opts);
strlcpy(set_opts, "A:o;s", sizeof set_opts);
q ← set_opts + strlen(set_opts);
for (ele ← 0; ele < NELEM(sh_options); ele++) {
  if (sh_options[ele].c) {
    if (sh_options[ele].flags & OF_CMDLINE) *p++ ← sh_options[ele].c;
    if (sh_options[ele].flags & OF_SET) *q++ ← sh_options[ele].c;
  }
}
*p ← '\0';
*q ← '\0';
```

This code is used in section 43.

45. ⟨ Scan arguments 45 ⟩ ≡

```
while ((optc ← ksh_getopt(argv, &go, opts)) ≠ −1) {
  int set ← (go.info & GI_PLUS) ? 0 : 1;
  switch (optc) {
  case 'A':     /* (re-)set an array's elements */
    arrayset ← set ? 1 : −1;
    array ← go.optarg;
    break;
  case 'o': ⟨ Process a long option and break 46 ⟩
  case '?': return −1;
  default: ⟨ Process a flag option and break 58 ⟩
  }
}
if (¬(go.info & GI_MINUSMINUS) ∧ argv[go.optind] ∧
      (argv[go.optind][0] ≡ '-' ∨ argv[go.optind][0] ≡ '+') ∧
      argv[go.optind][1] ≡ '\0') {
  if (argv[go.optind][0] ≡ '-' ∧ ¬Flag(FPOSIX))     /* lone - clears -v and -x flags */
    Flag(FVERBOSE) ← Flag(FXTRACE) ← 0;
  go.optind++;
}
if (setargsp)     /* "--" sets $#/$*/$@ even if there are no arguments */
  *setargsp ← ¬arrayset ∧ ((go.info & GI_MINUSMINUS) ∨ argv[go.optind]);
```

This code is used in section 43.

46. Long (named) options. Exactly "`set -o`" on its own prints the available options and their state but on the command line `-o` requires an option. A lone `+o` prints the arguments in a format that can be fed back into `set`.

```
⟨ Process a long option and break 46 ⟩ ≡
  if (go.optarg ≡ Λ) {
    printoptions(set);
    break;
  }
  i ← option(go.optarg);
  if (i ≠ −1 ∧ set ≡ Flag(i))
    ;     /* Don't check the context if the flag isn't changing so "set -o interactive" works if we're
         already interactive. */
  else if (i ≠ −1 ∧ (sh_options[i].flags & what)) change_flag((enum sh_flag) i, what, set);
  else {
    bi_errorf("%s: bad option", go.optarg);
    return −1;
  }
  break;
```

This code is used in section 45.

47.

```
#define OF_CMDLINE 0x01     /* command line */
#define OF_SET 0x02     /* set built-in */
#define OF_SPECIAL 0x04     /* a special variable changing */
#define OF_INTERNAL 0x08     /* set internally by shell */
#define OF_ANY (OF_CMDLINE | OF_SET | OF_SPECIAL | OF_INTERNAL)
⟨ Type definitions 17 ⟩ +≡
  struct option {
    const char *name;     /* long name of option */
    char c;     /* character flag (if any) */
    short flags;     /* OF_* */
  };
```

48. In addition to these there are special cases handled in *parse_args*: `-A`, `-o` & `-s`. Options are sorted by the order of their long name and must match the flags in **sh_flag**.

```
⟨ misc.c variables 48 ⟩ ≡
  const struct option sh_options[] ← {
        {"allexport", 'a', OF_ANY},
        {"braceexpand", 0, OF_ANY},      /* non-standard */
        {"bgnice", 0, OF_ANY},
        {Λ, 'c', OF_CMDLINE},
        {"csh-history", 0, OF_ANY},      /* non-standard */
#ifdef EMACS
        {"emacs", 0, OF_ANY},
#endif     /* EMACS */
        {"errexit", 'e', OF_ANY},
#ifdef EMACS
        {"gmacs", 0, OF_ANY},
#endif     /* EMACS */
        {"ignoreeof", 0, OF_ANY},
        {"interactive", 'i', OF_CMDLINE},
        {"keyword", 'k', OF_ANY},
        {"login", 'l', OF_CMDLINE},
        {"markdirs", 'X', OF_ANY},
        {"monitor", 'm', OF_ANY},
        {"noclobber", 'C', OF_ANY},
        {"noexec", 'n', OF_ANY},
        {"noglob", 'f', OF_ANY},
        {"nohup", 0, OF_ANY},
        {"nolog", 0, OF_ANY},      /* no effect */
        {"notify", 'b', OF_ANY},
        {"nounset", 'u', OF_ANY},
        {"physical", 0, OF_ANY},      /* non-standard */
        {"pipefail", 0, OF_ANY},      /* non-standard */
        {"posix", 0, OF_ANY},      /* non-standard */
        {"privileged", 'p', OF_ANY},
        {"restricted", 'r', OF_CMDLINE},
        {"sh", 0, OF_ANY},      /* non-standard */
        {"stdin", 's', OF_CMDLINE},      /* pseudo non-standard */
        {"trackall", 'h', OF_ANY},
        {"verbose", 'v', OF_ANY},
#ifdef VI
        {"vi", 0, OF_ANY},
        {"viraw", 0, OF_ANY},      /* no effect */
        {"vi-show8", 0, OF_ANY},      /* non-standard */
        {"vi-tabcomplete", 0, OF_ANY},      /* non-standard */
        {"vi-esccomplete", 0, OF_ANY},      /* non-standard */
#endif     /* VI */
        {"xtrace", 'x', OF_ANY},
        {Λ, 0, OF_INTERNAL},      /* FTALKING_I (internal use only) */
  };
```

See also section 205.

This code is used in section 9.

49. ⟨ Externally-linked variables 6 ⟩ +≡
```
extern const struct option sh_options[];
```

50. Translate "`-o` ⟨*option*⟩" into `F*` constant (also used for "`test -o option`").

⟨ misc.c 9 ⟩ +≡
```
int option(const char *n)
{
  unsigned int ele;
  for (ele ← 0; ele < NELEM(sh_options); ele++)
    if (sh_options[ele].name ∧ strcmp(sh_options[ele].name, n) ≡ 0) return ele;
  return −1;
}
```

51. ⟨ misc.c 9 ⟩ +≡
```
char *getoptions(void)
{
  unsigned int ele;
  char m[(int) FNFLAGS + 1];
  char *cp ← m;
  for (ele ← 0; ele < NELEM(sh_options); ele++)
    if (sh_options[ele].c ∧ Flag(ele)) *cp++ ← sh_options[ele].c;
  *cp ← 0;
  return str_save(m, ATEMP);
}
```

52. ⟨ misc.c declarations 52 ⟩ ≡
```
static char *options_fmt_entry(void *arg, int i, char *buf, int buflen);
static void printoptions(int verbose);
```
See also section 207.

This code is used in section 9.

53. This cannot be squeezed into the global type definitions—used to ⟨ Print options verbosely 55 ⟩.

⟨ misc.c 9 ⟩ +≡
```
struct options_info {
  int opt_width;
  struct {
    const char *name;
    int flag;
  } opts[NELEM(sh_options)];
};
```

54. ⟨ misc.c 9 ⟩ +≡
```
static void printoptions(int verbose)
{
  unsigned int ele;
  if (verbose) {⟨ Print options verbosely 55 ⟩}
  else {⟨ Print options quietly 57 ⟩}
}
```

55. ⟨ Print options verbosely 55 ⟩ ≡
```
struct options_info oi;
unsigned int n;
int len;
shprintf("Current option settings\n");
for (ele ← n ← oi.opt_width ← 0; ele < NELEM(sh_options); ele++) {
  if (sh_options[ele].name) {
    len ← strlen(sh_options[ele].name);
    oi.opts[n].name ← sh_options[ele].name;
    oi.opts[n++].flag ← ele;
    if (len > oi.opt_width) oi.opt_width ← len;
  }
}
print_columns(shl_stdout, n, options_fmt_entry, &oi, oi.opt_width + 5, 1);
```
This code is cited in section 53.

This code is used in section 54.

56. ⟨ misc.c 9 ⟩ +≡
```
static char *options_fmt_entry(void *arg, int i, char *buf, int buflen)
{
  struct options_info *oi ← (struct options_info *) arg;
  shf_snprintf(buf, buflen, "%-*s %s",
      oi→opt_width, oi→opts[i].name,
      Flag(oi→opts[i].flag) ? "on" : "off");
  return buf;
}
```

57. ⟨ Print options quietly 57 ⟩ ≡
```
shprintf("set");
for (ele ← 0; ele < NELEM(sh_options); ele++) {
  if (sh_options[ele].name) shprintf("  %co %s", Flag(ele) ? '-' : '+', sh_options[ele].name);
}
shprintf("\n");
```
This code is used in section 54.

58. ⟨ Process a flag option and **break** 58 ⟩ ≡
```
if (what ≡ OF_SET ∧ optc ≡ 's') {      /* "-s: sort positional params (XXX AT&T ksh stupidity) */
  sortargs ← 1;
  break;
}
for (ele ← 0; ele < NELEM(sh_options); ele++)
  if (optc ≡ sh_options[ele].c ∧ (what & sh_options[ele].flags)) {
    change_flag((enum sh_flag) ele, what, set);
    break;
  }
if (ele ≡ NELEM(sh_options)) {
  internal_errorf("%s: '%c'", __func__, optc);
  return −1;      /* not reached */
}
```
This code is used in section 45.

59. The *argv* passed to *main* is copied as soon as possible into an array in the permanent area so that changes do not affect the output of the `ps` command.

```
⟨ Create a copy of argv 59 ⟩ ≡
  l ← genv→loc;
  l→argv ← make_argv(argc − (argi − 1), &argv[argi − 1]);
  l→argc ← argc − argi;
  getopts_reset(1);
```

This code is used in section 15.

60.

```
static char **make_argv(int argc, char *argv[])
{
  int i;
  char **nargv;
  nargv ← areallocarray(Λ, argc + 1, sizeof(char *), &aperm);
  nargv[0] ← (char *) kshname;
  for (i ← 1; i < argc; i++) nargv[i] ← argv[i];
  nargv[i] ← Λ;
  return nargv;
}
```

61. Initial Shell Environment.

#define *version_param* (*initcoms*[2])

⟨ Global variables 5 ⟩ +≡

```
static const char *initcoms[] ← {
"typeset","-r","KSH_VERSION",Λ,
"typeset","-x","SHELL","PATH","HOME","PWD","OLDPWD",Λ,
"typeset","-ir","PPID",Λ,
"typeset","-i","OPTIND=1",Λ,
"eval","typeset␣-i␣RANDOM"
    "␣MAILCHECK=\"${MAILCHECK-600}\""
    "␣SECONDS=\"${SECONDS-0}\""
    "␣TMOUT=\"${TMOUT-0}\"",Λ,
"alias",      /* Standard ksh aliases */
    "hash=alias␣-t",      /* not "alias -t --"—"hash -r" needs to work */
    "stop=kill␣-STOP",
    "autoload=typeset␣-fu",
    "functions=typeset␣-f",
    "history=fc␣-l",
    "integer=typeset␣-i",
    "nohup=nohup␣",
    "local=typeset",
    "r=fc␣-s",
        /* Aliases that are built-in commands in AT&T: */
"login=exec␣login",Λ,
"alias","-tU",      /* This is what AT&T ksh seems to track, with the addition of emacs: */
    "cat","cc","chmod","cp","date","ed","emacs","grep","ls",
        "mail","make","mv","pr","rm","sed","sh","vi","who",Λ,
Λ
};
```

62. Execute the initialisation statements held in *initcoms* as a list of Λ-separated commands which are made up of C-strings of individual `shell` tokens.

⟨ Execute initialisation statements 62 ⟩ ≡

```
for (wp ← (char **) initcoms; *wp ≠ Λ; wp++) {
  shcomexec(wp);
  for ( ; *wp ≠ Λ; wp++)
      /* move to the next statement */ ;
}
```

This code is used in section 15.

63. This must be done after *j_init* as *tty_fd* is not initialised until then.

⟨ Initialise interactive editing 63 ⟩ ≡

```
if (Flag(FTALKING)) x_init( );
```

This code is used in section 15.

64. If an interactive shell could not determine its working directory then it emits a warning before reading the profile so that if it causes problems in them, user will know why things broke.

If the shell is a login shell then the system profile is read and if unprivileged then the user's profile in `$HOME/.profile` is also read.

If a non-privileged user is running the shell interactively, whether a login shell or not, then the file pointed to by `$ENV` (or possibly a default one) is read. A privileged shell reads only `/etc/suid_profile` (after the system profile of a login shell).

While reading the profile the `FRESTRICTED` (`-r`) and `FERREXIT` (`-e`) flags are temporarily disabled.

```
⟨ Temporarily disable -re 64 ⟩ ≡
  restricted ← Flag(FRESTRICTED);
  Flag(FRESTRICTED) ← 0;
  errexit ← Flag(FERREXIT);
  Flag(FERREXIT) ← 0;
```

This code is used in section 65.

65. `#define KSH_SYSTEM_PROFILE "/etc/profile"`

```
⟨ Read the profile file(s) 65 ⟩ ≡
  ⟨ Temporarily disable -re 64 ⟩
  if (¬current_wd[0] ∧ Flag(FTALKING))
    warningf(false, "Cannot␣determine␣current␣working␣directory");
  if (Flag(FLOGIN)) {
    Include(KSH_SYSTEM_PROFILE, 0, Λ, 1);
    if (¬Flag(FPRIVILEGED)) Include(substitute("$HOME/.profile", 0), 0, Λ, 1);
  }
  if (Flag(FPRIVILEGED)) Include("/etc/suid_profile", 0, Λ, 1);
  else if (Flag(FTALKING)) {
    char *env_file;
    env_file ← str_val(global("ENV"));      /* include $ENV */
#ifdef DEFAULT_ENV
    if (env_file ≡ null) env_file ← DEFAULT_ENV;      /* If env isn't set, include default environment */
#endif      /* DEFAULT_ENV */
    env_file ← substitute(env_file, DOTILDE);
    if (*env_file ≠ '\0') Include(env_file, 0, Λ, 1);
  }
  ⟨ Enable and/or re-enable FRESTRICTED & FERREXIT 67 ⟩
```

This code is used in section 15.

66.

```
⟨ Initialise (non-)interactivity 66 ⟩ ≡
  if (Flag(FTALKING)) {
    hist_init(s);
    alarm_init( );
  }
  else Flag(FTRACKALL) ← 1;      /* set after $ENV */
```

This code is used in section 15.

67. FERREXIT is simply re-applied. FRESTRICTED is reapplied but also set unconditionally if the shell is restricted other than by the -r switch.

```
⟨ Enable and/or re-enable FRESTRICTED & FERREXIT 67 ⟩ ≡
  if (is_restricted(argv[0]) ∨ is_restricted(str_val(global("SHELL")))) restricted ← 1;
  if (restricted) {
    static const char *const restr_com[] ← {
        "typeset", "-r", "PATH", "ENV", "SHELL",
        Λ
    };
    shcomexec((char **) restr_com);
    Flag(FRESTRICTED) ← 1;      /* After typeset command... */
  }
  if (errexit) Flag(FERREXIT) ← 1;
```

This code is used in section 65.

68. In addition to the -r option a shell is restricted if its full name matches one of a set of special names.

```
static int is_restricted(char *name)
{
  char *p;
  if ((p ← strrchr(name, '/'))) name ← p + 1;
  if (strcmp(name, "rsh") ∧
        strcmp(name, "rksh") ∧
        strcmp(name, "rpdksh") ∧
        strcmp(name, "pdrksh"))
          return (0);
  else return (1);
}
```

69. Memory. Memory allocation is by *malloc* with wrappers that allow it to be managed dynamically.

```
⟨ alloc.c 69 ⟩ ≡
#include <stdint.h>
#include <stdlib.h>
#include "sh.h"
```

See also sections 74, 75, 76, 77, 78, and 79.

70.
```
⟨ Shared function declarations 4 ⟩ +≡
  Area *ainit(Area *);
  void afreeall(Area *);
  void *alloc(size_t, Area *);
  void *areallocarray(void *, size_t, size_t, Area *);
  void *aresize(void *, size_t, Area *);
  void afree(void *, Area *);
```

71. Each pointer returned by *malloc* will be stored in an **Area** along with metadata to allow it to be *free*d later.

```
⟨ Type definitions 17 ⟩ +≡
  struct link {
    struct link *prev;
    struct link *next;
  };
  typedef struct Area {
    struct link *freelist;
  } Area;
```

72. One **Area** is held globally in *aperm* and initialised first before any other work can proceed. Another pointer to a temporary **Area** is declared to be an element of the current environment *genv* (see "Variable Storage & The Environment").

```
#define APERM &aperm
#define ATEMP &genv→area
⟨ Global variables 5 ⟩ +≡
  Area aperm;
```

73.
```
⟨ Externally-linked variables 6 ⟩ +≡
  extern Area aperm;      /* permanent object space */
```

74. An **Area**, including the permanent area, is initialised by setting its *freelist* to an empty list.

```
⟨ alloc.c 69 ⟩ +≡
  Area *ainit(Area *ap)
  {
    ap→freelist ← Λ;
    return ap;
  }
```

75. Memory is allocated with extra space to store the **link** object which points to the **Area** it's in. The L2P & P2L macros toggle between the address returned by *malloc* and the **link** address.

```
#define L2P(l) ((void *)(((char *)(l)) + sizeof(struct link)))
#define P2L(p) ((struct link *)(((char *)(p)) - sizeof(struct link)))
```

```
⟨ alloc.c 69 ⟩ +≡
  void *alloc(size_t size, Area *ap)
  {
    struct link *l;
    if (size > SIZE_MAX - sizeof(struct link))
        /* ensure that we don't overflow by allocating space for link */
      internal_errorf("unable to allocate memory");
    l ← malloc(sizeof(struct link) + size);
    if (l ≡ Λ) internal_errorf("unable to allocate memory");
    l→next ← ap→freelist;
    l→prev ← Λ;
    if (ap→freelist) ap→freelist→prev ← l;
    ap→freelist ← l;
    return L2P(l);
  }
```

76. Memory is frequently required to hold n items of size m. *areallocarray* does so in a manner which safely protects against multiplication overflow if n and/or m is too large. The conditional logic was cloned from *calloc*[1].

```
#define MUL_NO_OVERFLOW (1UL << (sizeof(size_t) * 4))  /* This is sqrt(SIZE_MAX + 1), as
          s1 × s2 ≤ SIZE_MAX if s1 < MUL_NO_OVERFLOW ∧ s2 < MUL_NO_OVERFLOW */
```

```
⟨ alloc.c 69 ⟩ +≡
  void *areallocarray(void *ptr, size_t nmemb, size_t size, Area *ap)
  {
    if ((nmemb ≥ MUL_NO_OVERFLOW ∨ size ≥ MUL_NO_OVERFLOW) ∧
        nmemb > 0 ∧
        SIZE_MAX/nmemb < size) {
      internal_errorf("unable to allocate memory");
    }
    return aresize(ptr, nmemb * size, ap);
  }
```

[1] http://cvsweb.openbsd.org/src/lib/libc/stdlib/malloc.c

77. An existing allocation can be resized using *aresize*, a straightforward wrapper around *realloc* which then replaces the stale pointer in the **Area**.

```
⟨ alloc.c 69 ⟩ +≡
  void *aresize(void *ptr, size_t size, Area *ap)
  {
    struct link *l, *l2, *lprev, *lnext;
    if (ptr ≡ Λ) return alloc(size, ap);
    if (size > SIZE_MAX − sizeof(struct link))
        /* ensure that we don't overflow by allocating space for link */
      internal_errorf("unable to allocate memory");
    l ← P2L(ptr);
    lprev ← l→prev;
    lnext ← l→next;
    l2 ← realloc(l, sizeof(struct link) + size);
    if (l2 ≡ Λ) internal_errorf("unable to allocate memory");
    if (lprev) lprev→next ← l2;
    else ap→freelist ← l2;
    if (lnext) lnext→prev ← l2;
    return L2P(l2);
  }
```

78. An individual allocation is freed by calling *free* after attaching the allocations either side of it in the **Area** to each other.

```
⟨ alloc.c 69 ⟩ +≡
  void afree(void *ptr, Area *ap)
  {
    struct link *l;
    if (¬ptr) return;
    l ← P2L(ptr);
    if (l→prev) l→prev→next ← l→next;
    else ap→freelist ← l→next;
    if (l→next) l→next→prev ← l→prev;
    free(l);
  }
```

79. The allocations held in an **Area** are freed en masse by walking the list of **link**s and calling *free* with each pointer.

```
⟨ alloc.c 69 ⟩ +≡
  void afreeall(Area *ap)
  {
    struct link *l, *l2;
    for (l ← ap→freelist; l ≠ Λ; l ← l2) {
      l2 ← l→next;
      free(l);
    }
    ap→freelist ← Λ;
  }
```

80. Hash Table. Definitions of variables and commands are stored in a hash table for fast retrieval. The API is defined in `table.h` and implemented in `table.c`.

```
⟨ table.h 80 ⟩ ≡
  unsigned int hash(const char *);
  void ktinit(struct table *, Area *, int);
  struct tbl *ktsearch(struct table *, const char *, unsigned int);
  struct tbl *ktenter(struct table *, const char *, unsigned int);
  void ktdelete(struct tbl *);
  void ktwalk(struct tstate *, struct table *);
  struct tbl *ktnext(struct tstate *);
  struct tbl **ktsort(struct table *);
```

81. `#define INIT_TBLS 8 /* initial table size (power of 2) */`

```
⟨ table.c 81 ⟩ ≡
#include <limits.h>
#include <stddef.h>
#include <string.h>
#include "sh.h"
  char *search_path; /* copy of either $PATH or def_path */
  const char *def_path; /* path to use if $PATH not set */
  char *tmpdir; /* $TMPDIR value */
  const char *prompt;
  int cur_prompt; /* $PS1 or $PS2 */
  int current_lineno; /* $LINENO value */
  static void texpand(struct table *, int);
  static int tnamecmp(const void *, const void *);
```

See also sections 83, 89, 90, 91, 94, 95, 96, 98, 99, 100, and 101.

82.

```
⟨ Externally-linked variables 6 ⟩ +≡
  extern char *search_path;
  extern const char *def_path;
  extern char *tmpdir;
  extern const char *prompt;
  extern int cur_prompt;
  extern int current_lineno;
```

83. No hash table would be complete (or work) without a function to turn a string into a number. How it does so is immaterial (hint: it performs maths).

```
⟨ table.c 81 ⟩ +≡
  unsigned int hash(const char *n)
  {
    unsigned int h ← 0;
    while (*n ≠ '\0') h ← 33 * h + (unsigned char)(*n++);
    return h;
  }
```

84. An object is stored in a hash table in some variant of this **tbl** object. When ksh needs to refer to a variable, function, built-in, etc. it will get or create one of these setting *flag*, *type* and the appropriate part of *val*.

If the hash table entry is referring to a command then *type* indicates its type.

```
#define CNONE 0 /* undefined */
#define CSHELL 1 /* built-in */
#define CFUNC 2 /* function */
#define CEXEC 4 /* executable command */
#define CALIAS 5 /* alias */
#define CKEYWD 6 /* keyword */
#define CTALIAS 7 /* tracked alias */
```

⟨ Type definitions 17 ⟩ +≡

```
struct tbl { /* table item */
  int flag; /* flags */
  int type; /* command type, base (if INTEGER), or offset from val.s of value (if EXPORT) */
  Area *areap; /* area to allocate from */
  union {
    char *s; /* string */
    int64_t i; /* integer */
    int (*f)(char **); /* int function */
    struct Op *t; /* "function" tree */
  } val; /* value */
  int index; /* index for an array */
  union {
    int field; /* field width for -L/-R/-Z */
    int errno_; /* CEXEC/CTALIAS */
  } u2;
  union {
    struct tbl *array; /* array values */
    char *fpath; /* temporary path to undef function */
  } u;
  char name[4]; /* name – variable length */
};
```

85. Bits 0-7 are reserved for flags affecting objects held in all hash tables.

```
#define ALLOC BIT(0) /* val.s has been allocated */
#define DEFINED BIT(1) /* is defined in block */
#define ISSET BIT(2) /* has value in s or i in val */
#define EXPORT BIT(3) /* exported variable/function */
#define TRACE BIT(4) /* variable: user-flagged, function: execution tracing */
```

86. These flags are used for objects holding a variable. `USERATTRIB` is attributes that can be set by the user, used to decide if an unset parameter should be reported by set/typeset). Does not include `ARRAY` or `LOCAL`

```
#define SPECIAL BIT(8)    /* $PATH, $IFS, $SECONDS, etc */
#define INTEGER BIT(9)    /* val.i contains integer value */
#define RDONLY BIT(10)    /* read-only variable */
#define LOCAL BIT(11)     /* for local typeset */
#define ARRAY BIT(13)     /* array */
#define LJUST BIT(14)     /* left justify */
#define RJUST BIT(15)     /* right justify */
#define ZEROFIL BIT(16)   /* 0 filled if RJUSTIFY, strip 0s if LJUSTIFY */
#define LCASEV BIT(17)    /* convert to lower case */
#define UCASEV_AL BIT(18) /* convert to upper case / autoload function */
#define INT_U BIT(19)     /* unsigned integer */
#define INT_L BIT(20)     /* long integer (no-op) */
#define IMPORT BIT(21)    /* flag to typeset: no arrays, must have "=" */
#define LOCAL_COPY BIT(22) /* with LOCAL, copy attrs from existing var */
#define EXPRINEVAL BIT(23) /* contents currently being evaluated */
#define EXPRLVALUE BIT(24) /* useable as lvalue (temp flag) */
#define USERATTRIB
        (EXPORT | INTEGER | RDONLY | LJUST | RJUST | ZEROFIL | LCASEV | UCASEV_AL | INT_U | INT_L)
```

87. These flags are used for the other tables *taliases*, *builtins*, *aliases*, *keywords* & *functions*.

```
#define KEEPASN BIT(8)    /* keep command assignments (eg. "var=x cmd") */
#define FINUSE BIT(9)     /* function is being executed */
#define FDELETE BIT(10)   /* function was deleted while it was executing */
#define FKSH BIT(11)      /* function defined with "function x" (vs. "x()") */
#define SPEC_BI BIT(12)   /* a POSIX special built-in */
#define REG_BI BIT(13)    /* a POSIX regular built-in */
```

88. The hash table itself is an array of **tbl** objects and the memory **Area** they're allocated in.

```
⟨ Type definitions 17 ⟩ +≡
  struct table {
    Area *areap;      /* area to allocate entries */
    int size, nfree;  /* hash size (always 2^n), free entries */
    struct tbl **tbls;  /* hashed table items */
  };
```

89. A hash table is initialised with an **Area** and optionally an initial size. The **table** object must have already been allocated.

```
⟨ table.c 81 ⟩ +≡
  void ktinit(struct table *tp, Area *ap, int tsize)
  {
    tp→areap ← ap;
    tp→tbls ← Λ;
    tp→size ← tp→nfree ← 0;
    if (tsize) texpand(tp, tsize);
  }
```

90. To find the location in a hash table which does or should contain an object *ktsearch* first obtains a pointer to the position indicated by the hash. If this location contains the correct object it's returned otherwise subsequent array locations are checked in turn, backwards[1], until found. If any location checked is empty then NULL is returned.

```
⟨ table.c 81 ⟩ +≡
  struct tbl *ktsearch(struct table *tp, const char *n, unsigned int h)
  {
    struct tbl **pp, *p;
    if (tp→size ≡ 0) return Λ;
    for (pp ← &tp→tbls[h & (tp→size − 1)]; (p ← *pp) ≠ Λ; pp −−) {
      if (*p→name ≡ *n ∧ strcmp(p→name, n) ≡ 0 ∧ (p→flag & DEFINED)) return p;
      if (pp ≡ tp→tbls) pp += tp→size;      /* wrap */
    }
    return Λ;
  }
```

91. Inserting a new entry to a hash table uses the same algorithm as *ktsearch* but if an empty slot is found a new **tbl** is created and returned. The table will be expanded if it fills up.

```
⟨ table.c 81 ⟩ +≡
  struct tbl *ktenter(struct table *tp, const char *n, unsigned int h)
  {
    struct tbl **pp, *p;
    int len;
    if (tp→size ≡ 0) texpand(tp, INIT_TBLS);
  Search:
    for (pp ← &tp→tbls[h & (tp→size − 1)]; (p ← *pp) ≠ Λ; pp −−) {
      if (*p→name ≡ *n ∧ strcmp(p→name, n) ≡ 0) return p;      /* found */
      if (pp ≡ tp→tbls) pp += tp→size;      /* wrap */
    }
    ⟨ Expand a hash table if necessary 92 ⟩
    ⟨ Create a new tbl entry in p 93 ⟩
    tp→nfree −−;
    *pp ← p;
    return p;
  }
```

92. ⟨ Expand a hash table if necessary 92 ⟩ ≡

```
  if (tp→nfree ≤ 0) {      /* too full */
    if (tp→size ≤ INT_MAX/2) texpand(tp, 2 * tp→size);
    else internal_errorf("too␣many␣vars");
    goto Search;
  }
```

This code is used in section 91.

[1] supersequent?

93. ⟨ Create a new **tbl** entry in p 93 ⟩ ≡

```
len ← strlen(n) + 1;
p ← alloc(offsetof(struct tbl, name[0]) + len, tp→areap);
p→flag ← 0;
p→type ← 0;
p→areap ← tp→areap;
p→u2.field ← 0;
p→u.array ← Λ;
memcpy(p→name, n, len);
```

This code is used in section 91.

94. If a hash table fills up it's expanded by requesting a larger memory allocation and copying each object from the old table to the new table, again using the same lookup algorithm as *ktsearch* to locate the slot a **tbl** object should occupy.

```
⟨ table.c 81 ⟩ +≡
  static void texpand(struct table *tp, int nsize)
  {
    int i;
    struct tbl *tblp, **p;
    struct tbl **ntblp, **otblp ← tp→tbls;
    int osize ← tp→size;
    ntblp ← areallocarray(Λ, nsize, sizeof(struct tbl *), tp→areap);
    for (i ← 0; i < nsize; i++) ntblp[i] ← Λ;
    tp→size ← nsize;
    tp→nfree ← 7 * nsize/10;      /* table can get 70% full */
    tp→tbls ← ntblp;
    if (otblp ≡ Λ) return;
    for (i ← 0; i < osize; i++)
      if ((tblp ← otblp[i]) ≠ Λ) {
        if ((tblp→flag & DEFINED)) {
          for (p ← &ntblp[hash(tblp→name) & (tp→size − 1)]; *p ≠ Λ; p−−)
            if (p ≡ ntblp) p += tp→size;      /* wrap */
          *p ← tblp;
          tp→nfree−−;
        }
        else if (¬(tblp→flag & FINUSE)) {
          afree(tblp, tp→areap);
        }
      }
    afree(otblp, tp→areap);
  }
```

95. To mark an object to be deleted its flags are cleared renderring the object unusable but resources associated with it are not freed.

```
⟨ table.c 81 ⟩ +≡
  void ktdelete(struct tbl *p)
  {
    p→flag ← 0;
  }
```

96. If PERF_DEBUG is defined then *tprintinfo* is compiled to allow for run-time introspection of hash tables.

```
⟨ table.c 81 ⟩ +≡
#ifdef PERF_DEBUG    /* performance debugging */
  void tprintinfo(struct table *tp);
  void tprintinfo(struct table *tp)
  {
    struct tbl *te;
    char *n;
    unsigned int h;
    int ncmp;
    int totncmp ← 0, maxncmp ← 0;
    int nentries ← 0;
    struct tstate ts;
    shellf("table␣size␣%d,␣nfree␣%d\n", tp→size, tp→nfree);
    shellf("␣␣␣␣Ncmp␣name\n");
    ktwalk(&ts, tp);
    while ((te ← ktnext(&ts))) {
      struct tbl **pp, *p;
      h ← hash(n ← te→name);
      ncmp ← 0;
      for (pp ← &tp→tbls[h & (tp→size − 1)]; (p ← *pp); pp−−) {
        ncmp++;    /* counter added to algorithm from ktsearch */
        if (*p→name ≡ *n ∧ strcmp(p→name, n) ≡ 0 ∧ (p→flag & DEFINED)) break;    /* was return p; */
        if (pp ≡ tp→tbls) pp += tp→size;    /* wrap */
      }
      shellf("␣␣␣␣%4d␣%s\n", ncmp, n);
      totncmp += ncmp;
      nentries++;
      if (ncmp > maxncmp) maxncmp ← ncmp;
    }
    if (nentries) shellf("␣␣%d␣entries,␣worst␣ncmp␣%d,␣avg␣ncmp␣%d.%02d\n",
        nentries, maxncmp, totncmp/nentries, (totncmp % nentries) * 100/nentries);
  }
#endif    /* PERF_DEBUG */
```

97. Iteration. When iterating through the allocated objects this structure holds the current state of the iteration.

```
⟨ Type definitions 17 ⟩ +≡
  struct tstate {
    int left;
    struct tbl **next;
  };
```

98. To visit every object *ktwalk* sets *left* to the number of live entries and *next* to the first element.

```
⟨ table.c 81 ⟩ +≡
  void ktwalk(struct tstate *ts, struct table *tp)
  {
    ts→left ← tp→size;
    ts→next ← tp→tbls;
  }
```

99. *ktnext* then looks for and returns another extant and defined object, decreasing *left* as it searches.

```
⟨ table.c 81 ⟩ +≡
  struct tbl *ktnext(struct tstate *ts)
  {
    while (--ts→left ≥ 0) {
      struct tbl *p ← *ts→next++;
      if (p ≠ Λ ∧ (p→flag & DEFINED)) return p;
    }
    return Λ;
  }
```

100. Sorting. A hashed array of objects is sorted using this comparison function.

```
⟨ table.c 81 ⟩ +≡
  static int tnamecmp(const void *p1, const void *p2)
  {
    char *name1 ← (*(struct tbl **) p1)→name;
    char *name2 ← (*(struct tbl **) p2)→name;
    return strcmp(name1, name2);
  }
```

101. *ktsort* returns a new (non-hashed) array of the **struct tbl** objects in *tp*.

```
⟨ table.c 81 ⟩ +≡
  struct tbl **ktsort(struct table *tp)
  {
    int i;
    struct tbl **p, **sp, **dp;
    p ← areallocarray(Λ, tp→size + 1, sizeof(struct tbl *), ATEMP);
    sp ← tp→tbls;      /* source */
    dp ← p;      /* dest */
    for (i ← 0; i < tp→size; i++)
      if ((*dp ← *sp++) ≠ Λ ∧ (((*dp)→flag & DEFINED) ∨ ((*dp)→flag & ARRAY))) dp++;
    i ← dp − p;
    qsortp((void **) p, (size_t) i, tnamecmp);
    p[i] ← Λ;
    return p;
  }
```

102. This wrapper around *qsort* is used in a few other places although it doesn't do anything a preprocessor **define** couldn't.

```
⟨ misc.c 9 ⟩ +≡
  void qsortp(void **base,      /* base address */
  size_t n,      /* elements */
  int(*f)(const void *, const void *))      /* compare function */
  {
    qsort(base, n, sizeof(char *), f);
  }
```

103. Variables & The Environment. The original `vars.c` began with a warning that the code was unreadable and needs a rewrite. Hopefully we can do something about the former here and, establishing ksh's in-built laziness, do away with the need for the latter.

A reminder from the hash table code above about how to reach the data in a variable:

* If (*flag* & `INTEGER`), *val*.*i* contains an integer value, and type contains its base, otherwise, *val*.*s* + *type* points to the string value.

* If (*flag* & `EXPORT`), *val*.*s* contains "`name=value`" for easy exporting.

⟨ `var.c` 103 ⟩ ≡

```
#include <sys/stat.h>
#include <sys/time.h>
#include <ctype.h>
#include <errno.h>
#include <inttypes.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#ifndef SMALL
#include <term.h>
#include <curses.h>
#endif
#include "sh.h"
  static struct tbl vtemp;
  static struct table specials;
  static char *formatstr(struct tbl *, const char *);
  static void export(struct tbl *, const char *);
  static int special(const char *);
  static void unspecial(const char *);
  static void getspec(struct tbl *);
  static void setspec(struct tbl *);
  static void unsetspec(struct tbl *);
  static struct tbl *arraysearch(struct tbl *, int);
  static const char *array_index_calc(const char *, bool *, int *);
```

See also sections 112, 113, 116, 119, 121, 131, 132, 133, 136, 140, 142, 143, 144, 148, 150, 151, 154, 155, 160, 163, 165, 171, 172, 173, 178, 182, 198, 199, 201, 202, and 203.

104. ⟨ Shared function declarations 4 ⟩ +≡

```
void newblock(void);
void popblock(void);
void initvar(void);
struct tbl *global(const char *);
struct tbl *local(const char *, bool);
char *str_val(struct tbl *);
int64_t intval(struct tbl *);
int setstr(struct tbl *, const char *, int);
struct tbl *setint_v(struct tbl *, struct tbl *, bool);
void setint(struct tbl *, int64_t);
int getint(struct tbl *, int64_t *, bool);
struct tbl *typeset(const char *, int, int, int, int);
void unset(struct tbl *, int);
char *skip_varname(const char *, int);
char *skip_wdvarname(const char *, int);
int is_wdvarname(const char *, int);
int is_wdvarassign(const char *);
char **makenv(void);
void change_random(void);
int array_ref_len(const char *);
char *arrayname(const char *);
void set_array(const char *, int, char **);
```

105. Variables and functions are stored in a **Block** object in the hash tables *vars* and *funs* respectively. A new **Block** is allocated every time a function is called which constitutes its local environment (A **Block** is not itself "an environment" but part of one—see below).

```
#define BF_DOGETOPTS BIT(0)      /* save/restore getopts state */
```

⟨ Type definitions 17 ⟩ +≡

```
struct Block {
  Area area;      /* area to allocate things */
/*struct arg_info argi;*/
  char **argv;
  int argc;
  int flags;      /* BF_* */
  struct table vars;      /* local variables */
  struct table funs;      /* local functions */
  Getopt getopts_state;
#if 1
  char *error;      /* error handler */
  char *exit;      /* exit handler */
#else
  Trap error, exit;
#endif
  struct Block *next;      /* enclosing block */
};
```

106. This is **arg_info**, which is not used anywhere except that comment.

```
#define AF_ARGV_ALLOC 0x1 /* argv array allocated */
#define AF_ARGS_ALLOCED 0x2 /* argument strings allocated */
#define AI_ARGV(a,i) ((i) ≡ 0 ? (a).argv[0] : (a).argv[(i) − (a).skip])
#define AI_ARGC(a) ((a).argc_ − (a).skip)
```

⟨ Type definitions 17 ⟩ +≡

```
struct arg_info { /* Argument info. Used for $#, $* for shell, functions, includes, etc. */
  int flags; /* AF_* */
  char **argv;
  int argc_;
  int skip; /* first arg is argv[0], second is argv[1 + skip] */
};
```

107. A new **Block** cannot be created except in an environment (**Env**) object. In addition it contains the file descriptor mapping in place when it was created, a reference to the previous environment from which it was created and a longjump buffer back to it, and finally a list of any temporary files held open.

```
#define EF_FUNC_PARSE BIT(0) /* function being parsed */
#define EF_BRKCONT_PASS BIT(1) /* set if E_LOOP must pass break/continue on */
#define EF_FAKE_SIGDIE BIT(2) /* hack to get info from unwind to quitenv */
```

⟨ Type definitions 17 ⟩ +≡

```
struct Env {
  short type; /* environment type—see below */
  short flags; /* EF_* */
  Area area; /* temporary allocation area */
  struct Block *loc; /* local variables and functions */
  short *savefd; /* original redirected fd's */
  struct Env *oenv; /* link to previous environment */
  sigjmp_buf jbuf; /* long jump back to env creator */
  struct Temp *temps; /* temp files */
};
```

108. Environments are used for more than a function's local namespace (`E_FUNC`) but also these activities. The # mark indicates that the environment has a valid longjump buffer in *jbuf* (for **unwind**).

```
#define E_NONE 0 /* dummy environment */
#define E_PARSE 1 /* # parsing a command */
#define E_FUNC 2 /* # executing a function */
#define E_INCL 3 /* # including a file via "." */
#define E_EXEC 4 /* executing a command tree */
#define E_LOOP 5 /* # executing for/while */
#define E_ERRH 6 /* # general error handler */
```

109. The "current" environment is held in the global variable *genv*.

⟨ Global variables 5 ⟩ +≡

```
struct Env *genv;
```

110. ⟨ Externally-linked variables 6 ⟩ +≡

```
extern struct Env *genv;
```

111. A new environment is created by copying the **Block** pointer from the current environment and replacing *genv*.

```
void newenv(int type)
{
  struct Env *ep;
  ep ← alloc(sizeof (*ep), ATEMP);
  ep→type ← type;
  ep→flags ← 0;
  ainit(&ep→area);
  ep→loc ← genv→loc;
  ep→savefd ← Λ;
  ep→oenv ← genv;
  ep→temps ← Λ;
  genv ← ep;
}
```

112. When an environment is created to call a defined function (`E_FUNC`) a new **Block** is created by copying the existing **Block**'s *argc*/*argv* and pointing *next* to the current **Block**.

⟨ `var.c` 103 ⟩ +≡

```
void newblock(void)
{
  struct Block *l;
  static char *const empty[] ← {null};
  l ← alloc(sizeof(struct Block), ATEMP);
  l→flags ← 0;
  ainit(&l→area);      /* TODO: could use genv→area (l→area => l→areap) */
  if (¬genv→loc) {
    l→argc ← 0;
    l→argv ← (char **) empty;
  }
  else {
    l→argc ← genv→loc→argc;
    l→argv ← genv→loc→argv;
  }
  l→exit ← l→error ← Λ;
  ktinit(&l→vars, &l→area, 0);
  ktinit(&l→funs, &l→area, 0);
  l→next ← genv→loc;
  genv→loc ← l;
}
```

113. To pop a **Block** any special variables (see below) which are present have their magic (un-)applied and the getopts state is restored.

```
⟨ var.c 103 ⟩ +≡
  void popblock(void)
  {
    struct Block *l ← genv→loc;
    struct tbl *vp, **vpp ← l→vars.tbls, *vq;
    int i;
    genv→loc ← l→next;      /* pop block */
    for (i ← l→vars.size; --i ≥ 0; )
      if ((vp ← *vpp++) ≠ Λ ∧ (vp→flag & SPECIAL)) {
        if ((vq ← global(vp→name))→flag & ISSET) setspec(vq);
        else unsetspec(vq);
      }
    if (l→flags & BF_DOGETOPTS) user_opt ← l→getopts_state;
    afreeall(&l→area);
    afree(l, ATEMP);
  }
```

114. After an environment is finished with, *quitenv* frees its resources and otherwise cleans up. If the **Block** stack was extended it is popped back first.

```
  void quitenv(struct Shf *shf)
  {
    struct Env *ep ← genv;
    int fd;
    if (ep→oenv ∧ ep→oenv→loc ≠ ep→loc) popblock( );
    if (ep→savefd ≠ Λ) {
      for (fd ← 0; fd < NUFILE; fd++)
        if (ep→savefd[fd]) restfd(fd, ep→savefd[fd]);      /* if ep→savefd[fd] < 0, means fd was closed */
      if (ep→savefd[2]) shf_reopen(2, SHF_WR, shl_out);      /* Clear any write errors */
    }
    if (ep→oenv ≡ Λ) {⟨ Quit the primary environment and exit 115 ⟩}
    if (shf) shf_close(shf);
    reclaim( );
    genv ← genv→oenv;
    afree(ep, ATEMP);
  }
```

115. If there's no environment to return to then either the main shell is exiting or *cleanup_parents_env* was called after *fork*. It's unclear why `EF_FAKE_SIGDIE` is useful.

```
⟨ Quit the primary environment and exit 115 ⟩ ≡
  if (ep¬type ≡ E_NONE) {      /* Main shell exiting? */
    if (Flag(FTALKING)) hist_finish();
    j_exit();
    if (ep¬flags & EF_FAKE_SIGDIE) {
      int sig ← exstat − 128;
      if ((sig ≡ SIGINT ∨ sig ≡ SIGTERM) ∧ getpgrp() ≡ kshpid) {      /* ham up our death a bit (AT&T
          ksh only seems to do this for SIGTERM); don't do it for SIGQUIT, since we'd dump a core... */
        setsig(&sigtraps[sig], SIG_DFL, SS_RESTORE_CURR | SS_FORCE);
        kill(0, sig);
      }
    }
  }
  if (shf) shf_close(shf);
  reclaim();
  exit(exstat);
```

This code is used in section 114.

116. A local variable is one which can be changed by the current function (and any which it calls) without affecting other instances of the same variable. To localise a variable a new slot is allocated in the current environment's **Block** if not already present. If not present (or not `DEFINED`) then the variable is looked up as normal and copied into the new allocation. Punctuation variables cannot be localised.

```
⟨ var.c 103 ⟩ +≡
  struct tbl *local(const char *n, bool copy)
  {
    struct Block *l ← genv¬loc;
    struct tbl *vp;
    unsigned int h;
    bool array;
    int val;
    n ← array_index_calc(n, &array, &val);      /* Check to see if this is an array */
    h ← hash(n);
    if (¬letter(*n)) {⟨ return a temporary read-only variable 118 ⟩}
    vp ← ktenter(&l¬vars, n, h);
    if (copy ∧ ¬(vp¬flag & DEFINED)) {⟨ Copy a variable into the local Block 117 ⟩}
    if (array) vp ← arraysearch(vp, val);
    vp¬flag |= DEFINED;
    if (special(n)) vp¬flag |= SPECIAL;
    return vp;
  }
```

117. If a variable already exists in the greater environment then its flags and type are copied into the new local variable.

```
⟨ Copy a variable into the local Block 117 ⟩ ≡
  struct Block *ll ← l;
  struct tbl *vq ← Λ;
  while ((ll ← ll→next) ∧ ¬(vq ← ktsearch(&ll→vars, n, h)))
        /* search for the variable in the rest of the variable stack */ ;
  if (vq) {
    vp→flag |= vq→flag &       /* nb. this set is exactly USERATTRIB */
      (EXPORT | INTEGER | RDONLY | LJUST | RJUST | ZEROFIL | LCASEV | UCASEV_AL | INT_U | INT_L);
    if (vq→flag & INTEGER) vp→type ← vq→type;
    vp→u2.field ← vq→u2.field;
  }
```

This code is used in section 116.

118. Punctuation and numeric variables don't require permanent storage but use *vtemp* for their short duration.

```
⟨ return a temporary read-only variable 118 ⟩ ≡
  vp ← &vtemp;
  vp→flag ← DEFINED | RDONLY;
  vp→type ← 0;
  vp→areap ← ATEMP;
  return vp;
```

This code is used in section 116.

119. Instead of storage the magic of punctuation/numeric variables is performed by *global* directly when a variable's value is looked up. Ordinary variables are looked up in the usual way or created if they do not already exist.

```
⟨ var.c 103 ⟩ +≡
  struct tbl *global(const char *n)
  {
    struct Block *l ← genv→loc;
    struct tbl *vp;
    long num;
    int c;
    unsigned int h;
    bool array;
    int val;
    n ← array_index_calc(n, &array, &val);      /* Check to see if this is an array */
    h ← hash(n);
    c ← (unsigned char) n[0];
    if (¬letter(c)) {
      if (array) errorf("bad substitution");
      ⟨ return the value of non-alphabetic variables 145 ⟩
    }
    ⟨ return a variable already in the environment 120 ⟩
    vp ← ktenter(&l→vars, n, h);
    if (array) vp ← arraysearch(vp, val);
    vp→flag |= DEFINED;
    if (special(n)) vp→flag |= SPECIAL;
    return vp;
  }
```

120. ⟨ return a variable already in the environment 120 ⟩ ≡

```
  for (l ← genv→loc; ; l ← l→next) {
    vp ← ktsearch(&l→vars, n, h);
    if (vp ≠ Λ) {
      if (array) return arraysearch(vp, val);
      else return vp;
    }
    if (l→next ≡ Λ) break;
  }
```

This code is used in section 119.

121. Look up a variable and set its attributes (see ⟨ Set/clear variable flags 127 ⟩) and its value if an assignment.

As near as I can tell the line "*vpbase* ← (*vp*→*flag* & ARRAY) ? *global*(*arrayname*(*tvar*)) : *vp*" may as well read "*vpbase* ← *vp*" because *global* should find the same object in *genv*→*loc* that the previous *global* call did or which *local* found or created.

```
⟨ var.c 103 ⟩ +≡
  struct tbl *typeset(const char *var, int set, int clr, int field, int base)
  {
    struct tbl *vp;
    struct tbl *vpbase, *t;
    char *tvar;
    const char *val;
    ⟨ Check that the variable name is valid 122 ⟩
    ⟨ Locate the value in an assignment 123 ⟩
    ⟨ Deny localising $PATH, $ENV or $SHELL if FRESTRICTED 124 ⟩
    vp ← (set & LOCAL)
        ? local(tvar, (set & LOCAL_COPY) ? true : false)
        : global(tvar);
    set &= ~(LOCAL | LOCAL_COPY);
    vpbase ← (vp→flag & ARRAY) ? global(arrayname(tvar)) : vp;
    ⟨ Validate arguments/flags if the variable is read-only 125 ⟩
    if (val) afree(tvar, ATEMP);
    if (set | clr) {⟨ Set/clear variable flags 127 ⟩}
    if (val ≠ Λ) {      /* these setstr calls can't fail—the readonly check is already done: */
      if (vp→flag & INTEGER) {      /* do not zero base before assignment */
        setstr(vp, val, KSH_UNWIND_ERROR | KSH_IGNORE_RDONLY);
        if (base > 0) vp→type ← base;      /* Done after assignment to override default */
      }
      else setstr(vp, val, KSH_RETURN_ERROR | KSH_IGNORE_RDONLY);
    }
    ⟨ Prepare a variable for exporting 126 ⟩
    return vp;
  }
```

122. Set *val* to the byte after the variable name in *var*. If it's an open-bracket ("[") then also skip over the index. If IMPORT is raised in *set* then the shell is starting and only simple (ie. numeric) indexes are permitted so that substitution is not carried out on the index value, which would be a major security hole.

```
⟨ Check that the variable name is valid 122 ⟩ ≡
  val ← skip_varname(var, false);
  if (val ≡ var) return Λ;
  if (*val ≡ '[') {
    int len;
    len ← array_ref_len(val);
    if (len ≡ 0) return Λ;
    if (set & IMPORT) {
      int i;
      for (i ← 1; i < len − 1; i++)
        if (¬digit(val[i])) return Λ;
    }
    val += len;
  }
```

This code is used in section 121.

123. The variable name is saved in *tvar*. If the call to *typeset* is an assignment (*var* contains an "=") then *tvar* is a copy of the variable-name part and *val* points to the new value, otherwise *val* is cleared and *tvar* set to *var*. When importing from the system environment the "=" is required.

```
⟨ Locate the value in an assignment 123 ⟩ ≡
  if (*val ≡ '=') tvar ← str_nsave(var, val++ − var, ATEMP);
  else {
    if (set & IMPORT) return Λ;
    tvar ← (char *) var;
    val ← Λ;
  }
```

This code is used in section 121.

124. ⟨ Deny localising $PATH, $ENV or $SHELL if FRESTRICTED 124 ⟩ ≡

```
  if (Flag(FRESTRICTED) ∧ (strcmp(tvar, "PATH") ≡ 0 ∨
        strcmp(tvar, "ENV") ≡ 0 ∨
        strcmp(tvar, "SHELL") ≡ 0))
    errorf("%s: restricted", tvar);
```

This code is used in section 121.

125. Only allow the export flag to be set if the variable is read-only. AT&T ksh allows any attribute to be changed, which means it can be truncated or modified (-L/-R/-Z/-i).

```
⟨ Validate arguments/flags if the variable is read-only 125 ⟩ ≡
  if ((vpbase→flag & RDONLY) ∧ (val ∨ clr ∨ (set & ∼EXPORT)))
      /* XXX check calls - is error here ok by POSIX? */
    errorf("%s: is read only", tvar);
```

This code is used in section 121.

126. Only the first element of an array is exported so *vpbase* is used.

```
⟨ Prepare a variable for exporting 126 ⟩ ≡
  if ((vpbase→flag & EXPORT) ∧
        ¬(vpbase→flag & INTEGER) ∧
        vpbase→type ≡ 0)
    export(vpbase, (vpbase→flag & ISSET) ? vpbase→val.s : null);
```

This code is used in section 121.

127. Set the attributes `INTEGER`, `RDONLY`, `EXPORT`, `TRACE`, `LJUST`, `RJUST`, `ZEROFIL`, `LCASEV` and `UCASEV_AL` as appropriate. To set the attributes on an array entry all entries must have their flags set (TODO: why?) and in addition the first element must be set for variable lookup to work. TODO: need to have one copy of attributes for arrays.

It might be better to call *fake_assign early_assign* as it's purpose is not to pretend to assign but to perform the assignment before changing the flags in *t* that it needs to know how to do it.

```
⟨ Set/clear variable flags 127 ⟩ ≡
  int ok ← 1;
  for (t ← vpbase; t; t ← t→u.array) {
    int fake_assign;
    int error_ok ← KSH_RETURN_ERROR;
    char *s ← Λ;
    char *free_me ← Λ;
    ⟨ Is the variable being assigned to? 128 ⟩
    if (fake_assign) {⟨ Save the value in a local buffer 129 ⟩}
    if (¬(t→flag & INTEGER) ∧ (set & INTEGER)) {
      t→type ← 0;
      t→flag &= ∼ALLOC;
    }
    if (¬(t→flag & RDONLY) ∧ (set & RDONLY))      /* Allow read-only variables to set an initial value */
      error_ok |= KSH_IGNORE_RDONLY;
    t→flag ← (t→flag | set) & ∼clr;
    if ((set & INTEGER) ∧ base > 0 ∧ (¬val ∨ t ≠ vp))
        /* Don't change base if assignment is to be done, in case assignment fails. */
      t→type ← base;
    if (set & (LJUST | RJUST | ZEROFIL)) t→u2.field ← field;
    if (fake_assign) {⟨ Save the local buffer in the fresh variable 130 ⟩}
  }
  if (¬ok) errorf(Λ);
```

This code is cited in section 121.

This code is used in section 121.

128. Is set, is not current or is not an assignment, and:

* the new flags involve string formatting (TODO: what about *clr*?),
* or is an integer being stringed,
* or a string being integered.

```
⟨ Is the variable being assigned to? 128 ⟩ ≡
  fake_assign ← (t→flag & ISSET) ∧ (¬val ∨ t ≠ vp) ∧
      ((set & (UCASEV_AL | LCASEV | LJUST | RJUST | ZEROFIL)) ∨
        ((t→flag & INTEGER) ∧ (clr & INTEGER)) ∨
        (¬(t→flag & INTEGER) ∧ (set & INTEGER)));
```

This code is used in section 127.

129. If the variable is not an integer then its value is copied to s (and marked to be freed later) otherwise the string representation of the integer is saved.

```
⟨ Save the value in a local buffer 129 ⟩ ≡
  if (t→flag & INTEGER) { s ← str_val(t); free_me ← Λ; }
  else { s ← t→val.s + t→type; free_me ← (t→flag & ALLOC) ? t→val.s : Λ; }
  t→flag &= ∼ALLOC;
```

This code is used in section 127.

130. *setstr* will take care of formatting the value according to the (new) flags. If *setstr* fails then the variable's contents are cleared but the flags remain.

```
⟨ Save the local buffer in the fresh variable 130 ⟩ ≡
  if (¬setstr(t, s, error_ok)) {
    ok ← 0;
    if (t→flag & INTEGER) t→flag &= ∼ISSET;
    else {
      if (t→flag & ALLOC) afree(t→val.s, t→areap);
      t→flag &= ∼(ISSET | ALLOC);
      t→type ← 0;
    }
  }
  afree(free_me, t→areap);
```

This code is used in section 127.

131. When a variable is unset it allocation is freed (if any) and its flags cleared. If the first element of an array (see below) is being unset then the rest of the array is not, so the DEFINED (and ARRAY) flags remain set although ISSET et al. will not be.

```
⟨ var.c 103 ⟩ +≡
  void unset(struct tbl *vp, int array_ref)
  {
    if (vp→flag & ALLOC) afree(vp→val.s, vp→areap);
    if ((vp→flag & ARRAY) ∧ ¬array_ref) {⟨ Free up an entire array 141 ⟩}
    vp→flag &= SPECIAL | (array_ref ? ARRAY | DEFINED : 0);
    if (vp→flag & SPECIAL) unsetspec(vp);
  }
```

132. Exporting variables. A variable which is being exported has its value is set to the full export string "`name=value`" and the offset within that string to the value is recorded. *op* here is a variable who's symbol shadows that of **struct Op**—`CWEB` does not understand this so *op* is incorrectly in **bold** rather than *italic*.

```
⟨ var.c 103 ⟩ +≡
  static void export(struct tbl *vp, const char *val)
  {
    char *xp;
    char *op ← (vp→flag & ALLOC) ? vp→val.s : Λ;
    int namelen ← strlen(vp→name);
    int vallen ← strlen(val) + 1;
    vp→flag |= ALLOC;
    xp ← alloc(namelen + 1 + vallen, vp→areap);
    memcpy(vp→val.s ← xp, vp→name, namelen);
    xp += namelen;
    *xp++ ← '=';
    vp→type ← xp − vp→val.s;      /* offset to value */
    memcpy(xp, val, vallen);
    afree(op, vp→areap);
  }
```

133. To export the environment to a subprocess *makenv* checks every variable in every layer of the environment for those with `ISSET` and `EXPORT` flags and appends the value to the growing *env*.

```
⟨ var.c 103 ⟩ +≡
  char **makenv(void)
  {
    struct Block *l;
    XPtrV env;
    struct tbl *vp, **vpp;
    int i;
    XPinit(env, 64);
    for (l ← genv→loc; l ≠ Λ; l ← l→next)
      for (vpp ← l→vars.tbls, i ← l→vars.size; −−i ≥ 0; )
        if ((vp ← *vpp++) ≠ Λ ∧ (vp→flag & (ISSET | EXPORT)) ≡ (ISSET | EXPORT)) {
          struct Block *l2;
          struct tbl *vp2;
          unsigned int h ← hash(vp→name);
          ⟨ Unexport any redefined instances 134 ⟩
          if ((vp→flag & INTEGER)) {⟨ Convert an integer to a string 135 ⟩}
          XPput(env, vp→val.s);
        }
    XPput(env, Λ);
    return (char **) XPclose(env);
  }
```

134. When a variable is exported any variables it has shadowed have their EXPORT flag lowerd.

```
⟨ Unexport any redefined instances 134 ⟩ ≡
  for (l2 ← l→next; l2 ≠ Λ; l2 ← l2→next) {
    vp2 ← ktsearch(&l2→vars, vp→name, h);
    if (vp2 ≠ Λ) vp2→flag &= ∼EXPORT;
  }
```

This code is used in section 133.

135. ⟨ Convert an integer to a string 135 ⟩ ≡

```
char *val;
val ← str_val(vp);
vp→flag &= ∼(INTEGER | RDONLY);
setstr(vp, val, KSH_RETURN_ERROR);      /* setstr can't fail here */
```

This code is used in section 133.

136. Arrays. The shell mimicks sparse arrays using a linked list, which makes lookup, insertion and deletion "slow" (O(n) rather than O(1)) in exchange for minimising wasted storage. In other circumstances this would be horribly inefficient but arrays in the shell are not expected to grow to the sort of size where it would become noticable.

An array variable has the `ARRAY` flag set. The array entry with index 0 is that variable in the hash table; the link to the next entry is in each entry's *t→u.array*.

In order to look up an array entry *arraysearch* walks along the linked list for an entry with the desired index value or one which is larger, in which case a new entry object is created and inserted into the list.

```
⟨ var.c 103 ⟩ +≡
  static struct tbl *arraysearch(struct tbl *vp, int val)
  {
    struct tbl *prev, *curr, *new;
    size_t namelen ← strlen(vp→name) + 1;
    vp→flag |= ARRAY | DEFINED;
    vp→index ← 0;
    if (val ≡ 0) return vp;      /* The table entry is always 0. */
    ⟨ Locate or create an array entry 137 ⟩
    ⟨ Fill in an (unset & undefined) array entry 139 ⟩
    if (curr ≠ new) {⟨ Insert a new entry into an array 138 ⟩}
    return new;
  }
```

137. It seems reasonable to merge the next section into the final (outer) **else** clause.

```
⟨ Locate or create an array entry 137 ⟩ ≡
  prev ← vp;
  curr ← vp→u.array;
  while (curr ∧ curr→index < val) {
    prev ← curr;
    curr ← curr→u.array;
  }
  if (curr ∧ curr→index ≡ val) {
    if (curr→flag & ISSET) return curr;
    else new ← curr;
  }
  else new ← alloc(sizeof(struct tbl) + namelen, vp→areap);
```

This code is used in section 136.

138.

```
⟨ Insert a new entry into an array 138 ⟩ ≡
  prev→u.array ← new;
  new→u.array ← curr;
```

This code is used in section 136.

139. Values are copied into the new entry from the table header *vp*.

```
⟨ Fill in an (unset & undefined) array entry 139 ⟩ ≡
  strlcpy(new→name, vp→name, namelen);
  new→flag ← vp→flag & ∼(ALLOC | DEFINED | ISSET | SPECIAL);
  new→type ← vp→type;
  new→areap ← vp→areap;
  new→u2.field ← vp→u2.field;
  new→index ← val;
```

This code is used in section 136.

140. To set the contents of an array repeatedly calls *arraysearch* until all entries have been assigned. As noted this is of course "quite non-optimal" (nearly $O(n^2)$ after possibly deallocating every element one by one) but, again, array are expected to be small.

AT&T ksh allows "`set -A`" but not "`set +A`" (merge new contents) of a read-only variable.

It would be nice for assignment to completely succeed or completely fail. This only really effects integer arrays where evaluation of some values may fail.

```
⟨ var.c 103 ⟩ +≡
  void set_array(const char *var, int reset, char **vals)
  {
    struct tbl *vp, *vq;
    int i;
    vp ← global(var);
    if ((vp→flag & RDONLY)) errorf("%s: is read only", var);
    if (reset > 0) unset(vp, 0);      /* trash existing values and attributes */
    for (i ← 0; vals[i]; i++) {
      vq ← arraysearch(vp, i);
      setstr(vq, vals[i], KSH_RETURN_ERROR);      /* may fail—TODO: deal with errors here */
    }
  }
```

141. To free an array entirely (in *unset*) each entry and its allocation is freed in turn and the header's *u.array* pointer is cleared.

```
⟨ Free up an entire array 141 ⟩ ≡
  struct tbl *a, *tmp;
  for (a ← vp→u.array; a; ) {
    tmp ← a;
    a ← a→u.array;
    if (tmp→flag & ALLOC) afree(tmp→val.s, tmp→areap);
    afree(tmp, tmp→areap);
  }
  vp→u.array ← Λ;
```

This code is used in section 131.

142. *arrayname* returns a copy of the name part of an array, ie. the first part of a string up to (and not including) the first "`[`". If one isn't found then the whole string is returned but as noted this shouldn't happen so "why worry?"

```
⟨ var.c 103 ⟩ +≡
  char *arrayname(const char *str)
  {
    const char *p;
    if ((p ← strchr(str, '[')) ≡ 0) return (char *) str;
    return str_nsave(str, p − str, ATEMP);
  }
```

143. Return the string length of the array subscript (eg. in "`[1+2]`" the length is 3). *cp* is assumed to point to the open bracket. Returns 0 if there is no matching closing bracket.

```
⟨ var.c 103 ⟩ +≡
  int array_ref_len(const char *cp)
  {
    const char *s ← cp;
    int c;
    int depth ← 0;
    while ((c ← *s++) ∧ (c ≠ ']' ∨ --depth))
      if (c ≡ '[') depth++;
    if (¬c) return 0;
    return s − cp;
  }
```

144. To resolve a subscript expression to the index value *array_index_calc* (after determining if an expression is even an array lookup, returned via *arrayp*) calls *substitute* to expand embedded variables etc. and then *evaluate* to evaluate the mathematical expression. The index value is returned via *valp*.

```
⟨ var.c 103 ⟩ +≡
  static const char *array_index_calc(const char *n, bool *arrayp, int *valp)
  {
    const char *p;
    int len;
    *arrayp ← false;
    p ← skip_varname(n, false);
    if (p ≠ n ∧ *p ≡ '[' ∧ (len ← array_ref_len(p))) {
      char *sub, *tmp;
      int64_t rval;      /* Calculate the value of the subscript */
      *arrayp ← true;
      tmp ← str_nsave(p + 1, len − 2, ATEMP);
      sub ← substitute(tmp, 0);
      afree(tmp, ATEMP);
      n ← str_nsave(n, p − n, ATEMP);
      evaluate(sub, &rval, KSH_UNWIND_ERROR, true);
      if (rval < 0 ∨ rval > INT_MAX) errorf("%s: subscript %" PRIi64 " out of range", n, rval);
      *valp ← rval;
      afree(sub, ATEMP);
    }
    return n;
  }
```

145. Special Variables. The special variables handled directly by *global* are the punctuation variables `$$`, `$!`, `$?`, `$#` & `$-` and the numeric positional parameters `$0`, `$1`, `$2`, &c.

```
⟨ return the value of non-alphabetic variables 145 ⟩ ≡
  vp ← &vtemp;
  vp→flag ← DEFINED;
  vp→type ← 0;
  vp→areap ← ATEMP;
  *vp→name ← c;
  if (digit(c)) {⟨ return the value of positional parameters 146 ⟩}
  vp→flag |= RDONLY;
  if (n[1] ≠ '\0') return vp;      /* more than one character */
  vp→flag |= ISSET | INTEGER;
  ⟨ return the value of punctuation variables 147 ⟩
  return vp;
```

This code is used in section 119.

146. Numeric variables (`$0` – `$9`) have the value copied from the environment's *argv* array.

```
⟨ return the value of positional parameters 146 ⟩ ≡
  errno ← 0;
  num ← strtol(n, Λ, 10);
  if (errno ≡ 0 ∧ num ≤ l→argc)
        setstr(vp, l→argv[num], KSH_RETURN_ERROR);      /* setstr can't fail here */
  vp→flag |= RDONLY;
  return vp;
```

This code is used in section 145.

147. The punctuation variable values come from various places.

```
⟨ return the value of punctuation variables 147 ⟩ ≡
  switch (c) {
  case '$': vp→val.i ← kshpid; break;
  case '?': vp→val.i ← exstat; break;
  case '#': vp→val.i ← l→argc; break;
  case '!':      /* If there is no job expand to nothing */
    if ((vp→val.i ← j_async( )) ≡ 0)
      vp→flag &= ~(ISSET | INTEGER);
    break;
  case '-': vp→flag &= ~INTEGER;
    vp→val.s ← getoptions( );
    break;
  default: vp→flag &= ~(ISSET | INTEGER);
  }
```

This code is used in section 145.

148. Other special variables have normal names like `$PATH`, `$RANDOM` and `$TERM`. To identify when a variable should be treated specially they are named in the global hash table *specials*. Although they are marked `ISSET` they are not given a value.

```
#define V_NONE 0
#define V_PATH 1
#define V_IFS 2
#define V_SECONDS 3
#define V_OPTIND 4
#define V_MAIL 5
#define V_MAILPATH 6
#define V_MAILCHECK 7
#define V_RANDOM 8
#define V_HISTCONTROL 9
#define V_HISTSIZE 10
#define V_HISTFILE 11
#define V_VISUAL 12
#define V_EDITOR 13
#define V_COLUMNS 14
#define V_POSIXLY_CORRECT 15
#define V_TMOUT 16
#define V_TMPDIR 17
#define V_LINENO 18
#define V_TERM 19
```

⟨ var.c 103 ⟩ +≡

```
void initvar(void)
{
  static const struct {
    const char *name;
    int v;
  } names[] ← {⟨ Names of special variables 149 ⟩};
  int i;
  struct tbl *tp;
  ktinit(&specials, APERM, 32);  /* must be 2^n (currently 19 specials, table extends at 22) */
  for (i ← 0; names[i].name; i++) {
    tp ← ktenter(&specials, names[i].name, hash(names[i].name));
    tp→flag ← DEFINED | ISSET;
    tp→type ← names[i].v;
  }
}
```

149. TODO: Find a way to align code in columns.

```
⟨ Names of special variables 149 ⟩ ≡
  {"COLUMNS", V_COLUMNS},
  {"IFS", V_IFS},
  {"OPTIND", V_OPTIND},
  {"PATH", V_PATH},
  {"POSIXLY_CORRECT", V_POSIXLY_CORRECT},
  {"TMPDIR", V_TMPDIR},
  {"HISTCONTROL", V_HISTCONTROL},
  {"HISTFILE", V_HISTFILE},
  {"HISTSIZE", V_HISTSIZE},
  {"EDITOR", V_EDITOR},
  {"VISUAL", V_VISUAL},
  {"MAIL", V_MAIL},
  {"MAILCHECK", V_MAILCHECK},
  {"MAILPATH", V_MAILPATH},
  {"RANDOM", V_RANDOM},
  {"SECONDS", V_SECONDS},
  {"TMOUT", V_TMOUT},
  {"LINENO", V_LINENO},
  {"TERM", V_TERM},
  {Λ, 0}
```

This code is used in section 148.

150. *special* searches *specials* to see if a given name is one of these special names.

```
⟨ var.c 103 ⟩ +≡
  static int special(const char *name)
  {
    struct tbl *tp;
    tp ← ktsearch(&specials, name, hash(name));
    return tp ∧ (tp→flag & ISSET) ? tp→type : V_NONE;
  }
```

151. For historical reasons some variables can lose their special status.

```
⟨ var.c 103 ⟩ +≡
  static void unspecial(const char *name)
  {
    struct tbl *tp;
    tp ← ktsearch(&specials, name, hash(name));
    if (tp) ktdelete(tp);
  }
```

152. Notes on particular special variables. `$IFS` is used frequently when scanning strings so when its set the `C_IFS` character class is updated. The first character of `$IFS` is also used to separate generated lists of words (eg. "`$*`") and is saved in *ifs0* whenever `$IFS` is updated.

```
⟨ Global variables 5 ⟩ +≡
  int ifs0 ← ' ';
```

153.

```
⟨ Externally-linked variables 6 ⟩ +≡
  extern int ifs0;
```

154. Some special variables are special because the value is dynamic. *getspec* updates *vp* to hold its current value. The variable's SPECIAL flag is disabled while updating the value.

```
⟨ var.c 103 ⟩ +≡
  static struct timespec seconds;      /* time $SECONDS last set */
  static int user_lineno;      /* what user set $LINENO to */
  static void getspec(struct tbl *vp)
  {
    switch (special(vp->name)) {
    case V_SECONDS: vp->flag &= ~SPECIAL;
      if (vp->flag & ISSET) {      /* On start up the value of $SECONDS is used before seconds has been
            set—don't do anything in this case (see initcoms). */
        struct timespec difference, now;
        clock_gettime(CLOCK_MONOTONIC, &now);
        timespecsub(&now, &seconds, &difference);
        setint(vp, (int64_t) difference.tv_sec);
      }
      vp->flag |= SPECIAL;
      break;
    case V_RANDOM: vp->flag &= ~SPECIAL;
      setint(vp, (int64_t)(rand( ) & 0x7fff));
      vp->flag |= SPECIAL;
      break;
    case V_HISTSIZE: vp->flag &= ~SPECIAL;
      setint(vp, (int64_t) histsize);
      vp->flag |= SPECIAL;
      break;
    case V_OPTIND: vp->flag &= ~SPECIAL;
      setint(vp, (int64_t) user_opt.uoptind);
      vp->flag |= SPECIAL;
      break;
    case V_LINENO: vp->flag &= ~SPECIAL;
      setint(vp, (int64_t) current_lineno + user_lineno);
      vp->flag |= SPECIAL;
      break;
    }
  }
```

155. Setting the value of a special variable has many different magical actions which are performed by *setspec*. The behaviours are grouped into broadly-related categories but no meaning should be inferred from this—it is only to better break the large **switch** statement over multiple pages.

```
⟨ var.c 103 ⟩ +≡
  static void setspec(struct tbl *vp)
  {
    char *s;
    switch (special(vp->name)) {
      ⟨ Set special I/O, getopt & POSIX variables 156 ⟩
      ⟨ Set special history & interaction variables 157 ⟩
      ⟨ Set special variables to do with mail 158 ⟩
      ⟨ Set special variables that need no introduction 159 ⟩
    }
  }
```

156. These variables are the most commonly used. *tmpdir* is only set if the value is an absolute, writable, searchable path and is a directory otherwise it is cleared.

```
⟨ Set special I/O, getopt & POSIX variables 156 ⟩ ≡
case V_PATH: afree(search_path, APERM);
  search_path ← str_save(str_val(vp), APERM);
  flushcom(1);      /* clear tracked aliases */
  break;
case V_IFS: setctypes(s ← str_val(vp), C_IFS);
  ifs0 ← *s;
  break;
case V_OPTIND: vp→flag &= ~SPECIAL;
  getopts_reset((int) intval(vp));
  vp→flag |= SPECIAL;
  break;
case V_POSIXLY_CORRECT: change_flag(FPOSIX, OF_SPECIAL, 1);
  break;
case V_TMPDIR: afree(tmpdir, APERM);
  tmpdir ← Λ;
  {
    struct stat statb;
    s ← str_val(vp);
    if (s[0] ≡ '/' ∧ access(s, W_OK | X_OK) ≡ 0 ∧ stat(s, &statb) ≡ 0 ∧ S_ISDIR(statb.st_mode))
      tmpdir ← str_save(s, APERM);
  }
  break;
```

This code is used in section 155.

157. The history and $EDITOR/$VISUAL variables are also set simply. $COLUMNS' value is clamped to the range MIN_COLS – INT_MAX before assignment to *x_cols*. $TERM is only applied if SMALL is not defined.

```
⟨ Set special history & interaction variables 157 ⟩ ≡
case V_HISTCONTROL: sethistcontrol(str_val(vp)); break;
case V_HISTSIZE: vp→flag &= ∼SPECIAL;
  sethistsize((int) intval(vp));
  vp→flag |= SPECIAL;
  break;
case V_HISTFILE: sethistfile(str_val(vp)); break;
case V_VISUAL: set_editmode(str_val(vp)); break;
case V_EDITOR:
  if (¬(global("VISUAL")→flag & ISSET)) set_editmode(str_val(vp));
  break;
case V_COLUMNS:
  {
    int64_t l;
    if (getint(vp, &l, false) ≡ −1) {
      x_cols ← MIN_COLS;
      break;
    }
    if (l ≤ MIN_COLS ∨ l > INT_MAX) x_cols ← MIN_COLS;
    else x_cols ← l;
  }
  break;
case V_TERM:
#ifndef SMALL
  {
    int ret;
    vp→flag &= ∼SPECIAL;
    if (setupterm(str_val(vp), shl_out→fd, &ret) ≡ ERR) del_curterm(cur_term);
    vp→flag |= SPECIAL;
  }
#endif
  break;
```

This code is used in section 155.

158. The MAIL* variables control whether and how ksh will check for mail.

```
⟨ Set special variables to do with mail 158 ⟩ ≡
case V_MAIL: mbset(str_val(vp)); break;
case V_MAILPATH: mpset(str_val(vp)); break;
case V_MAILCHECK: vp→flag &= ∼SPECIAL;
  mcset(intval(vp));
  vp→flag |= SPECIAL;
  break;
```

This code is used in section 155.

159. The case for V_TMOUT came with the comment "*enforce integer to avoid command execution from initcoms*" which seems to suggest that the call to *intval* was thought to coerce *vp* into an INTEGER, but in fact the string in *vp* is copied to a number which is discarded.

```
⟨ Set special variables that need no introduction 159 ⟩ ≡
case V_RANDOM: vp→flag &= ∼SPECIAL;
  srand_deterministic((unsigned int) intval(vp));
  vp→flag |= SPECIAL;
  break;
case V_SECONDS: vp→flag &= ∼SPECIAL;
  clock_gettime(CLOCK_MONOTONIC, &seconds);
  seconds.tv_sec −= intval(vp);
  vp→flag |= SPECIAL;
  break;
case V_TMOUT: vp→flag &= ∼SPECIAL;
  intval(vp);
  vp→flag |= SPECIAL;
  if (vp→flag & INTEGER)      /* AT&T ksh seems to do this (only listen if integer) */
    ksh_tmout ← vp→val.i ≥ 0 ? vp→val.i : 0;
  break;
case V_LINENO: vp→flag &= ∼SPECIAL;      /* The −1 is because line numbering starts at 1. */
  user_lineno ← (unsigned int) intval(vp) − current_lineno − 1;
  vp→flag |= SPECIAL;
  break;
```

This code is used in section 155.

160. Conversely when a special variable is being unset the magic needs to be removed. This final case which calls *unspecial* included the comment that the AT&T ksh man page says `$OPTIND`, `$OPTARG` and `$_` lose their special meaning but in fact `$OPTARG` does not (it's still set by *getopts*) and `$_` is also still set in various places. Unsetting these in AT&T ksh does not loose their special status: `$IFS`, `$COLUMNS`, `$PATH`, `$TMPDIR`, `$VISUAL`, `$EDITOR` but the author did not know what AT&T does for `$MAIL`, `$MAILPATH`, `$HISTSIZE` or `$HISTFILE`.

Unsetting this ksh addition also has no effect: `$POSIXLY_CORRECT`. Use "`set +o posix`" instead.

```
⟨ var.c 103 ⟩ +≡
  static void unsetspec(struct tbl *vp)
  {
    switch (special(vp→name)) {
    case V_PATH: afree(search_path, APERM);
      search_path ← str_save(def_path, APERM);
      flushcom(1);      /* clear tracked aliases */
      break;
    case V_IFS: setctypes("␣\t\n", C_IFS);
      ifs0 ← '␣';
      break;
    case V_TMPDIR: afree(tmpdir, APERM);
      tmpdir ← Λ;
      break;
    case V_MAIL: mbset(Λ);
      break;
    case V_MAILPATH: mpset(Λ);
      break;
    case V_HISTCONTROL: sethistcontrol(Λ);
      break;
    case V_LINENO:
    case V_MAILCHECK:
    case V_RANDOM:
    case V_SECONDS:
    case V_TMOUT:
      unspecial(vp→name);
      break;
    }
  }
```

161. String & Number Formatting. The empty string is shared as much as possible by using *null*.

```
⟨ Global variables 5 ⟩ +≡
  char null[] ← "";
```

162.

```
⟨ Externally-linked variables 6 ⟩ +≡
  extern char null[];
```

163. Set variable to string value. If the variable holds an integer then the string is expected to hold a mathematical expression which is evaluated to resolve the value to store.

In some cases callers can indicate that an error condition should not cause a failure, such as initial assignment to a read-only variable.

```
⟨ var.c 103 ⟩ +≡
  int setstr(struct tbl *vq, const char *s, int error_ok)
  {
    const char *fs ← Λ;
    int no_ro_check ← error_ok & KSH_IGNORE_RDONLY;
    error_ok &= ∼KSH_IGNORE_RDONLY;
    if ((vq→flag & RDONLY) ∧ ¬no_ro_check) {
      warningf(true, "%s: is read only", vq→name);
      if (¬error_ok) errorf(Λ);
      return 0;
    }
    if (¬(vq→flag & INTEGER)) {⟨ Save a string value in a string variable 164 ⟩}      /* string dest */
    else { if (¬v_evaluate(vq, s, error_ok, true)) return 0; }      /* integer dest */
    vq→flag |= ISSET;
    if ((vq→flag & SPECIAL)) setspec(vq);
    afree((void *) fs, ATEMP);
    return 1;
  }
```

164.

```
⟨ Save a string value in a string variable 164 ⟩ ≡
  if ((vq→flag & ALLOC)) {
    if (s ≥ vq→val.s ∧
          s ≤ vq→val.s + strlen(vq→val.s))
      internal_errorf("%s: %s=%s: assigning to self", __func__, vq→name, s);
    afree(vq→val.s, vq→areap);
  }
  vq→flag &= ∼(ISSET | ALLOC);
  vq→type ← 0;
  if (s ∧ (vq→flag & (UCASEV_AL | LCASEV | LJUST | RJUST))) fs ← s ← formatstr(vq, s);
  if ((vq→flag & EXPORT)) export(vq, s);
  else {
    vq→val.s ← str_save(s, vq→areap);
    vq→flag |= ALLOC;
  }
```

This code is used in section 163.

165. A formatted string is padded with "␣" or "0" and/or made uppper- or lower-case.

```
⟨ var.c 103 ⟩ +≡
  static char *formatstr(struct tbl *vp, const char *s)
  {
    int olen, nlen;
    char *p, *q;
    olen ← strlen(s);
    ⟨ Determine the post-formatted string's length 166 ⟩
    p ← alloc(nlen + 1, ATEMP);
    if (vp→flag & (RJUST | LJUST)) {
      int slen;
      if (vp→flag & RJUST) {⟨ Right justify a string 167 ⟩}
      else {⟨ Left justify a string 168 ⟩}
    }
    else memcpy(p, s, olen + 1);
    if (vp→flag & UCASEV_AL) {⟨ Up-case a string 169 ⟩}
    else if (vp→flag & LCASEV) {⟨ Down-case a string 170 ⟩}
    return p;
  }
```

166. If a formatted string's field length isn't set it becomes the length of the input string.

```
⟨ Determine the post-formatted string's length 166 ⟩ ≡
  if (vp→flag & (RJUST | LJUST)) {
    if (¬vp→u2.field) vp→u2.field ← olen;      /* default field width */
    nlen ← vp→u2.field;
  }
  else nlen ← olen;
```

This code is used in section 165.

167.

```
⟨ Right justify a string 167 ⟩ ≡
  const char *r ← s + olen;
  while (r > s ∧ isspace((unsigned char) r[−1])) −−r;      /* strip trailing spaces */
  slen ← r − s;
  if (slen > vp→u2.field) {
    s += slen − vp→u2.field;
    slen ← vp→u2.field;
  }
  shf_snprintf(p, nlen + 1,
      ((vp→flag & ZEROFIL) ∧ digit(*s)) ? "%0*s%.*s" : "%*s%.*s",
      vp→u2.field − slen, null, slen, s);
```

This code is used in section 165.

168.

```
⟨ Left justify a string 168 ⟩ ≡
  while (isspace((unsigned char) *s)) s++;
  if (vp→flag & ZEROFIL) while (*s ≡ '0') s++;
  shf_snprintf(p, nlen + 1, "%-*.*s", vp→u2.field, vp→u2.field, s);
```

This code is used in section 165.

169. Algorithms to return the upper case or lower case of a string are the same but opposite.

```
⟨ Up-case a string 169 ⟩ ≡
  for (q ← p; *q; q++)
    if (islower((unsigned char) *q)) *q ← toupper((unsigned char) *q);
```

This code is used in section 165.

170.

```
⟨ Down-case a string 170 ⟩ ≡
  for (q ← p; *q; q++)
    if (isupper((unsigned char) *q)) *q ← tolower((unsigned char) *q);
```

This code is used in section 165.

171. If the variable being assigned to is a string then *setint* will assign its argument to a temporary variable and call *setstr* to save the latter in the former. Notably it doesn't use *setint_v*.

```
⟨ var.c 103 ⟩ +≡
  void setint(struct tbl *vq, int64_t n)
  {
    if (¬(vq→flag & INTEGER)) {
      struct tbl *vp ← &vtemp;
      vp→flag ← (ISSET | INTEGER);
      vp→type ← 0;
      vp→areap ← ATEMP;
      vp→val.i ← n;
      setstr(vq, str_val(vp), KSH_RETURN_ERROR);      /* strstr can't fail here */
    }
    else vq→val.i ← n;
    vq→flag |= ISSET;
    if ((vq→flag & SPECIAL)) setspec(vq);
  }
```

172. Conversely *setint_v* will *convert* the variable *vq* into an integer and then store in it the integer value from *vp* (they may be the same).

```
⟨ var.c 103 ⟩ +≡
  struct tbl *setint_v(struct tbl *vq, struct tbl *vp, bool arith)
  {
    int base;
    int64_t num;
    if ((base ← getint(vp, &num, arith)) ≡ −1) return Λ;
    if (¬(vq→flag & INTEGER) ∧ (vq→flag & ALLOC)) {
      vq→flag &= ∼ALLOC;
      afree(vq→val.s, vq→areap);
    }
    vq→val.i ← num;
    if (vq→type ≡ 0) vq→type ← base;      /* default base */
    vq→flag |= ISSET | INTEGER;
    if (vq→flag & SPECIAL) setspec(vq);
    return vq;
  }
```

173. The string representation of an integer is returned by *str_val*. The variable is not converted to the returned string.

```
⟨ var.c 103 ⟩ +≡
  char *str_val(struct tbl *vp)
  {
    char *s;
    if ((vp→flag & SPECIAL)) getspec(vp);
    if (¬(vp→flag & ISSET)) s ← null;
    else if (¬(vp→flag & INTEGER)) s ← vp→val.s + vp→type;      /* string source */
    else {⟨ Format an integer as a string 174 ⟩}      /* integer source */
    return s;
  }
```

174. An integer is converted to a string by building the string backwards into a buffer. The worst case is when *base* ≡ 2 so space is allocated to fit "± ⟨base⟩ # ⟨64 bytes...⟩ \0".

```
⟨ Format an integer as a string 174 ⟩ ≡
  char strbuf[1 + 2 + 1 + BITS(int64_t) + 1];
  const char *digits ← (vp→flag & UCASEV_AL) ?
      "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" :
      "0123456789abcdefghijklmnopqrstuvwxyz";
  uint64_t n;
  unsigned int base;
  s ← strbuf + sizeof (strbuf);
  ⟨ Find n ← |vp→val.i| and base 175 ⟩
  *−−s ← '\0';
  ⟨ Prepend n % base and reduce until n ≡ 0 176 ⟩
  if (base ≠ 10) {⟨ Prepend # and base 177 ⟩}
  if (¬(vp→flag & INT_U) ∧ vp→val.i < 0) *−−s ← '-';      /* Possibly prepend "-" */
  if (vp→flag & (RJUST | LJUST)) s ← formatstr(vp, s);      /* case already dealt with */
  else s ← str_save(s, ATEMP);
```

This code is used in section 173.

175.
```
⟨ Find n ← |vp→val.i| and base 175 ⟩ ≡
  if (vp→flag & INT_U) n ← (uint64_t) vp→val.i;
  else n ← (vp→val.i < 0) ? −vp→val.i : vp→val.i;
  base ← (vp→type ≡ 0) ? 10 : vp→type;
  if (base < 2 ∨ base > strlen(digits)) base ← 10;
```

This code is used in section 174.

176.
```
⟨ Prepend n % base and reduce until n ≡ 0 176 ⟩ ≡
  do {
    *−−s ← digits[n % base];
    n /= base;
  } while (n ≠ 0);
```

This code is used in section 174.

177. If the string representation is not in base 10, the base is prepended—in base 10—followed by "#".

```
⟨ Prepend # and base 177 ⟩ ≡
  *--s ← '#';
  *--s ← digits[base % 10];
  if (base ≥ 10) *--s ← digits[base/10];
```

This code is used in section 174.

178. *getint* goes the other way.

```
⟨ var.c 103 ⟩ +≡
  int getint(struct tbl *vp, int64_t *nump, bool arith)
  {
    char *s;
    int c;
    int base, neg;
    int have_base ← 0;
    int64_t num;
    if (vp→flag & SPECIAL) getspec(vp);      /* XXX is it possible for ISSET to be set and val.s ≡ Λ? */
    if (¬(vp→flag & ISSET) ∨ (¬(vp→flag & INTEGER) ∧ vp→val.s ≡ Λ)) return −1;
    if (vp→flag & INTEGER) {
      *nump ← vp→val.i;
      return vp→type;
    }
    s ← vp→val.s + vp→type;
    if (s ≡ Λ) s ← null;      /* redundant given initial test */
    base ← 10;
    num ← 0;
    neg ← 0;
    if (arith ∧ *s ≡ '0' ∧ *(s + 1)) {⟨ Determine a source expression's inline base 179 ⟩}
    for (c ← (unsigned char) *s++; c; c ← (unsigned char) *s++) {
      if (c ≡ '-') { neg++; }
      else if (c ≡ '#') {⟨ Extract number's base from string 180 ⟩}
      else if (letnum(c)) {⟨ Increase num by the value of the next digit 181 ⟩}
      else return −1;
    }
    if (neg) num ← −num;
    *nump ← num;
    return base;
  }
```

179.

```
⟨ Determine a source expression's inline base 179 ⟩ ≡
  s++;
  if (*s ≡ 'x' ∨ *s ≡ 'X') {
    s++;
    base ← 16;
  }
  else if (vp→flag & ZEROFIL) {
    while (*s ≡ '0') s++;
  }
  else base ← 8;
  have_base++;
```

This code is used in section 178.

180. ⟨ Extract number's base from string 180 ⟩ ≡

```
base ← (int) num;
if (have_base ∨ base < 2 ∨ base > 36) return −1;
num ← 0;
have_base ← 1;
```

This code is used in section 178.

181. ⟨ Increase *num* by the value of the next digit 181 ⟩ ≡

```
if (isdigit(c)) c −= '0';
else if (islower(c)) c −= 'a' − 10;      /* todo: assumes ascii */
else if (isupper(c)) c −= 'A' − 10;      /* todo: assumes ascii */
else c ← −1;      /* _: force error */
if (c < 0 ∨ c ≥ base) return −1;
num ← num * base + c;
```

This code is used in section 178.

182. *getint* returns -1 if an error occurs which *intval* upgrades to a bona fide error.

⟨ var.c 103 ⟩ +≡

```
int64_t intval(struct tbl *vp)
{
  int64_t num;
  int base;

  base ← getint(vp, &num, false);
  if (base ≡ −1) errorf("%s: bad number", str_val(vp));
      /* XXX check calls - is error here ok by POSIX? */
  return num;
}
```

183. Initialisation. Now the global environment can be initialised. Aliases, keywords, etc. are stored in these global variables.

```
⟨ Global variables 5 ⟩ +≡
  struct table taliases;      /* tracked aliases */
  struct table builtins;      /* built-in commands */
  struct table aliases;       /* aliases */
  struct table keywords;      /* keywords */
  struct table homedirs;      /* homedir cache */
```

184.
```
⟨ Externally-linked variables 6 ⟩ +≡
  extern struct table taliases;      /* tracked aliases */
  extern struct table builtins;      /* built-in commands */
  extern struct table aliases;       /* aliases */
  extern struct table keywords;      /* keywords */
  extern struct table homedirs;      /* homedir cache */
```

185. Recall that *env* is a variable declared at the top of *main*.

```
⟨ Set up the base environment 185 ⟩ ≡
  memset(&env, 0, sizeof (env));
  env.type ← E_NONE;
  ainit(&env.area);
  genv ← &env;
  newblock( );      /* set up global genv→vars and genv→funs */
```

This code is used in section 15.

186.
```
⟨ Set up variable and command dictionaries 186 ⟩ ≡
  ktinit(&taliases, APERM, 0);
  ktinit(&aliases, APERM, 0);
  ktinit(&homedirs, APERM, 0);
```

This code is used in section 15.

187.
```
⟨ Define built-in commands 187 ⟩ ≡
  ktinit(&builtins, APERM, 64);      /* must be 2^n (currently 40 built-ins, table extends at 44) */
  for (i ← 0; shbuiltins[i].name ≠ Λ; i++) builtin(shbuiltins[i].name, shbuiltins[i].func);
  for (i ← 0; kshbuiltins[i].name ≠ Λ; i++) builtin(kshbuiltins[i].name, kshbuiltins[i].func);
```

This code is used in section 15.

188. `$IFS` is set early, immediately after importing the caller's environment, so that it overrides any value which may have been imported. The variables in *initsubs* (`$PS2`, `$PS3` & `$PS4`) are also given values but only if they were *not* imported—this allows the calling process to override these sub-prompts.

The space character is harmless because *initifs* is treated as though the whole expression is quoted. Recall also that the backslashes here escape the next character *to the* C *compiler*, ie. *initifs* is exactly 8 bytes long including the trailing `'\0'`.

```
⟨ Global variables 5 ⟩ +≡
  static const char initifs[ ] ← "IFS=␣\t\n";
  static const char initsubs[ ] ← "${PS2=>␣}␣${PS3=#?␣}␣${PS4=+␣}";
```

189. It's not clear whether the value returned by *substitute*, which has been allocated somewhere in the depths of *expand*, can be discarded without being freed. On the other hand it should probably be *null*.

```
⟨ Import the calling process' environment 189 ⟩ ≡
  if (environ ≠ Λ)
    for (wp ← environ; *wp ≠ Λ; wp++) typeset(*wp, IMPORT | EXPORT, 0, 0, 0);
  kshpid ← procpid ← getpid( );
  typeset(initifs, 0, 0, 0, 0);      /* for security */
  substitute(initsubs, 0);
```

This code is used in section 15.

190. `$PS1` is set separately from the other prompt values to differentiate between the privileged root account and user accounts. If `$PS1` was explicitely imported then it is set to that instead.

```
⟨ Set $USER and the main prompt 190 ⟩ ≡
  ksheuid ← geteuid( );
  init_username( );
  safe_prompt ← ksheuid ? "$␣" : "#␣";
  {
    struct tbl *vp ← global("PS1");
    if (¬(vp¬flag & ISSET)) {      /* Set $PS1 if it isn't set */
      setstr(vp, "\\h\\$␣", KSH_RETURN_ERROR);      /* setstr can't fail here */
    }
  }
```

This code is used in section 15.

191. If the current value of `$PWD` is invalid it is ignored, otherwise it's stripped of excess `.` and `..` components. `$PWD` is set to the simplified value if it's valid or the bogus value if not.

```
⟨ Figure out $PWD 191 ⟩ ≡
  {
    struct stat s_pwd, s_dot;
    struct tbl *pwd_v ← global("PWD");
    char *pwd ← str_val(pwd_v);
    char *pwdx ← pwd;
    if (pwd[0] ≠ '/' ∨
        stat(pwd, &s_pwd) ≡ −1 ∨
        stat(".", &s_dot) ≡ −1 ∨
        s_pwd.st_dev ≠ s_dot.st_dev ∨
        s_pwd.st_ino ≠ s_dot.st_ino)
      /* Only try to use existing PWD if it is valid */
      pwdx ← Λ;
    set_current_wd(pwdx);      /* calls getcwd (eventually) if pwdx ≡ Λ */
    if (current_wd[0]) simplify_path(current_wd);
    if (current_wd[0] ∨ pwd ≠ null)
      /* Only set pwd if we know where we are or if it had a bogus value */
      setstr(pwd_v, current_wd, KSH_RETURN_ERROR);      /* setstr can't fail here */
  }
```

This code is used in section 15.

192. ⟨ Store initial $PPID & $(K)SH_VERSION 192 ⟩ ≡

```
ppid ← getppid( );
setint(global("PPID"), (int64_t) ppid);
setstr(global(version_param), ksh_version, KSH_RETURN_ERROR);      /* setstr can't fail here */
```

This code is used in section 15.

193. Set $PATH to *def_path* (will set the path global variable). (import of environment below will probably change this setting).

⟨ Set the global $PATH variable 193 ⟩ ≡

```
def_path ← _PATH_DEFPATH;
{
  size_t len ← confstr(_CS_PATH, Λ, 0);
  char *new;
  if (len > 0) {
    confstr(_CS_PATH, new ← alloc(len + 1, APERM), len + 1);
    def_path ← new;
  }
}
{
  struct tbl *vp ← global("PATH");
  setstr(vp, def_path, KSH_RETURN_ERROR);      /* setstr can't fail here */
}
```

This code is used in section 15.

194. On systems where the superuser is not called "root" this routine will incorrectly set *username* (which is not the same as the $USER variable) to "root" when the current user is privileged and $USER was imported.

```
static void init_username(void)
{
  char *p;
  struct tbl *vp ← global("USER");
  if (vp¬flag & ISSET) p ← ksheuid ≡ 0 ? "root" : str_val(vp);
  else p ← getlogin( );
  strlcpy(username, p ≠ Λ ? p : "?", sizeof username);
}
```

195. Miscellanea. *cleanup_parents_env* is called in the new process after a *fork* to close unwanted file descriptors. Other memory can't be easily reclaimed however such subprocesses are generally short-lived so this should not be a big problem.

```
void cleanup_parents_env(void)
{
  struct Env *ep;
  int fd;
  for (ep ← genv; ep; ep ← ep→oenv) {
    if (ep→savefd) {
      for (fd ← 0; fd < NUFILE; fd++)
        if (ep→savefd[fd] > 0) close(ep→savefd[fd]);
      afree(ep→savefd, &ep→area);
      ep→savefd ← Λ;
    }
  }
  genv→oenv ← Λ;
}
```

196. Similarly, when a process is about to call *execve* to run another program *cleanup_proc_env* removes any temporary files that were open.

```
void cleanup_proc_env(void)
{
  struct Env *ep;
  for (ep ← genv; ep; ep ← ep→oenv) remove_temps(ep→temps);
}
```

197. This routine removes temporary files that the current environment holds and frees its temporary **Area**.

```
static void reclaim(void)
{
  remove_temps(genv→temps);
  genv→temps ← Λ;
  afreeall(&genv→area);
}
```

198. Return a pointer to the first char past a legal variable name or the argument if there is no legal name. If the whole string is valid the pointer returned will be Λ.

```
⟨ var.c 103 ⟩ +≡
char *skip_varname(const char *s, int aok)
{
  int alen;
  if (s ∧ letter(*s)) {
    while (*++s ∧ letnum(*s)) ;
    if (aok ∧ *s ≡ '[' ∧ (alen ← array_ref_len(s))) s += alen;
  }
  return (char *) s;
}
```

199. The lexical analyser will return strings where each character is preceeded by a token indicating its type which these `wd*` variant functions handle. *skip_wdvarname* performs the same task as *skip_varname*.

```
⟨ var.c 103 ⟩ +≡
  char *skip_wdvarname(const char *s, int aok)      /* skip array de-reference? */
  {
    if (s[0] ≡ CHAR ∧ letter(s[1])) {
      do {
        s += 2;
      } while (s[0] ≡ CHAR ∧ letnum(s[1]));
      if (aok ∧ s[0] ≡ CHAR ∧ s[1] ≡ '[') {⟨ Skip possible array de-reference 200 ⟩}
    }
    return (char *) s;
  }
```

200. ⟨ Skip possible array de-reference 200 ⟩ ≡

```
  const char *p ← s;
  char c;
  int depth ← 0;
  while (1) {
    if (p[0] ≠ CHAR) break;
    c ← p[1];
    p += 2;
    if (c ≡ '[') depth++;
    else if (c ≡ ']' ∧ --depth ≡ 0) {
      s ← p;
      break;
    }
  }
```

This code is used in section 199.

201. Check if an analysed string is a variable name.

```
⟨ var.c 103 ⟩ +≡
  int is_wdvarname(const char *s, int aok)
  {
    char *p ← skip_wdvarname(s, aok);
    return p ≠ s ∧ p[0] ≡ EOS;
  }
```

202. Check if an analysed string is a variable assignment.

```
⟨ var.c 103 ⟩ +≡
  int is_wdvarassign(const char *s)
  {
    char *p ← skip_wdvarname(s, true);
    return p ≠ s ∧ p[0] ≡ CHAR ∧ p[1] ≡ '=';
  }
```

203. Called after a *fork* to bump the random number generator so that subprocess will not get the same random number sequence even if the caller does not use `$RANDOM`.

```
⟨ var.c 103 ⟩ +≡
  void change_random(void)
  {
    rand( );
  }
```

204. Character Data (Strings). `charclass.h` is a header written by Todd C. Miller for OpenBSD's *libc* and is also in the public domain. It provides POSIX character class support for *fnmatch* and *glob*.

#define NCCLASSES (**sizeof** (*cclasses*)/**sizeof** (*cclasses*[0]) − 1)

⟨ charclass.h 204 ⟩ ≡

```
static const struct CClass
{
  const char *name;
  int (*isctype)(int);
}
cclasses[] ← {
    {"alnum", isalnum},
    {"alpha", isalpha},
    {"blank", isblank},
    {"cntrl", iscntrl},
    {"digit", isdigit},
    {"graph", isgraph},
    {"lower", islower},
    {"print", isprint},
    {"punct", ispunct},
    {"space", isspace},
    {"upper", isupper},
    {"xdigit", isxdigit},
    {Λ, Λ}
};
```

205. An array of UCHAR_MAX + 1 flags is allocated in *ctypes* to quickly determine how to interpret any byte.

#define *ctype*(*c*, *t*) ¬¬(*ctypes*[(**unsigned char**)(*c*)] & (*t*))
#define *letter*(*c*) *ctype*(*c*, C_ALPHA)
#define *digit*(*c*) *isdigit*((**unsigned char**)(*c*))
#define *letnum*(*c*) (*ctype*(*c*, C_ALPHA) ∨ *isdigit*((**unsigned char**)(*c*)))

⟨ misc.c variables 48 ⟩ +≡

```
short ctypes[UCHAR_MAX + 1];      /* type bits for unsigned char */
```

206. ⟨ Externally-linked variables 6 ⟩ +≡

```
extern short ctypes[];
```

207. ⟨ misc.c declarations 52 ⟩ +≡

```
static const unsigned char *cclass(const unsigned char *, int);
```

208. More use could be made of these constants.

```
#define OPAREN  '('
#define CPAREN  ')'
#define OBRACK  '['
#define CBRACK  ']'
#define OBRACE  '{'
#define CBRACE  '}'
```

209. C_DIGIT was BIT(1). There are now only 8 flags so *ctypes* could be changed from **short** to **char**.

```
#define C_ALPHA BIT(0)    /* [a-z_A-Z] */
#define C_LEX1 BIT(2)     /* ['\0' ␣ TAB \n |&;<>()] */
#define C_VAR1 BIT(3)     /* [*@#!$-?] */
#define C_IFSWS BIT(4)    /* [TAB ␣ \n] (IFS white space—not $IFS) */
#define C_SUBOP1 BIT(5)   /* [=-+?] */
#define C_SUBOP2 BIT(6)   /* [#%] */
#define C_IFS BIT(7)      /* $IFS */
#define C_QUOTE BIT(8)    /* [␣ \n TAB "#$&'()*;<>?[\`|] (characters which need quoting) */
```

```
⟨ misc.c 9 ⟩ +≡
  void initctypes(void)
  {
    int c;
    for (c ← 'a'; c ≤ 'z'; c++) ctypes[c] |= C_ALPHA;
    for (c ← 'A'; c ≤ 'Z'; c++) ctypes[c] |= C_ALPHA;
    ctypes['_'] |= C_ALPHA;
    setctypes("␣\t\n|&;<>()", C_LEX1);
    setctypes("*@#!$-?", C_VAR1);
    setctypes("␣\t\n", C_IFSWS);
    setctypes("=-+?", C_SUBOP1);
    setctypes("#%", C_SUBOP2);
    setctypes("␣\n\t\"#$&'()*;<>?[\\`|", C_QUOTE);
  }
```

210.

```
⟨ misc.c 9 ⟩ +≡
  void setctypes(const char *s, int t)
  {
    int i;
    if (t & C_IFS) {
      for (i ← 0; i < UCHAR_MAX + 1; i++) ctypes[i] &= ~C_IFS;    /* clear C_IFS from all characters */
      ctypes[0] |= C_IFS;    /* ... except '\0' */
    }
    while (*s ≠ 0) ctypes[(unsigned char) *s++] |= t;
  }
```

211. There is a wrapper around *strcmp* which doesn't affect its arguments at all except to permit **const** C-string pointers.

```
⟨ misc.c 9 ⟩ +≡
  int xstrcmp(const void *p1, const void *p2)
  {
    return (strcmp(*(char **) p1, *(char **) p2));
  }
```

212. A string can be copied into dynamic storage.

```
⟨ misc.c 9 ⟩ +≡
  char *str_save(const char *s, Area *ap)
  {
    size_t len;
    char *p;
    if (¬s) return Λ;
    len ← strlen(s) + 1;
    p ← alloc(len, ap);
    strlcpy(p, s, len);
    return (p);
  }
```

213. The same but for an undelimited string who's maximum length is known (the real length may be shorter if there's a `'\0'` embedded).

```
⟨ misc.c 9 ⟩ +≡
  char *str_nsave(const char *s, int n, Area *ap)
  {
    char *ns;
    if (n < 0) return 0;
    ns ← alloc(n + 1, ap);
    ns[0] ← '\0';
    return strncat(ns, s, n);
  }
```

214. A (positive) number is converted to a string. Note that the argument is 64 bits to cope with the absolute value of `INT_MIN` without having to rely on signed arithmetic (and its inherent undefined behaviour).

```
⟨ misc.c 9 ⟩ +≡
  char *u64ton(uint64_t n, int base)
  {
    char *p;
    static char buf[20];
    p ← &buf[sizeof (buf)];
    *--p ← '\0';
    do {
      *--p ← "0123456789ABCDEF"[n % base];
      n /= base;
    } while (n ≠ 0);
    return p;
  }
```

215. A string is converted to a number, again calculated in a wider representation than is used to hold the number which is clamped to INT_MIN–INT_MAX.

```
⟨ misc.c 9 ⟩ +≡
  int getn(const char *as, int *ai)
  {
    char *p;
    long n;
    n ← strtol(as, &p, 10);
    if (¬*as ∨ *p ∨ INT_MIN ≥ n ∨ n ≥ INT_MAX) return 0;
    *ai ← (int) n;
    return 1;
  }
```

216. Errors from *getn* are normally harmless.

```
⟨ misc.c 9 ⟩ +≡
  int bi_getn(const char *as, int *ai)
  {
    int rv ← getn(as, ai);
    if (¬rv) bi_errorf("%s: bad number", as);
    return rv;
  }
```

217. Expandable String Macros. The file expanding strings are implemented in, `expand.h` begins with a demonstration of its usage and defines a single function. The rest of the implementation uses C macros.

```
⟨ expand.h 217 ⟩ ≡
#if 0
  XString xs;
  char *xp;
  Xinit(xs, xp, 128, ATEMP);      /* allocate initial string */
  while ((c ← generate()))
  {
    Xcheck(xs, xp);      /* expand string if necessary */
    Xput(xs, xp, c);      /* add character */
  }
  return Xclose(xs, xp);      /* resize string */
#endif      /* 0 */
  char *Xcheck_grow_(XString *xsp, char *xp, size_t more);
```

218. `#define Xstring(xs, xp) ((xs).beg)`

```
⟨ Type definitions 17 ⟩ +≡
  typedef struct XString {
    char *end, *beg;      /* end, begin of string */
    size_t len;      /* length */
    Area *areap;      /* Area to allocate/free from */
  } XString;
  typedef char *XStringP;
```

219. The implementation of *Xcheck_grow_* is in `misc.c`.

```
⟨ misc.c 9 ⟩ +≡
  char *Xcheck_grow_(XString *xsp, char *xp, size_t more)
  {
    char *old_beg ← xsp→beg;
    xsp→len += more > xsp→len ? more : xsp→len;
    xsp→beg ← aresize(xsp→beg, xsp→len + 8, xsp→areap);
    xsp→end ← xsp→beg + xsp→len;
    return xsp→beg + (xp − old_beg);
  }
```

220. The *Xcheck* and *Xinit* macros have a magic +`X_EXTRA` in the lengths. This is so that you can put up to `X_EXTRA` characters in a **XString** before calling *Xcheck*. (See *yylex* in `lex.c`)

```
#define X_EXTRA 8      /* this many extra bytes in XString */
```

221. **#define** *Xinit*(*xs*, *xp*, *length*, *area*) **do**
{
(*xs*).*len* ← *length*;
(*xs*).*areap* ← (*area*);
(*xs*).*beg* ← *alloc*((*xs*).*len* + `X_EXTRA`, (*xs*).*areap*);
(*xs*).*end* ← (*xs*).*beg* + (*xs*).*len*;
xp ← (*xs*).*beg*;
}
while (0)
#define *Xfree*(*xs*, *xp*) *afree*((*xs*).*beg*, (*xs*).*areap*)

222. Append a **char** to a string.
#define *Xput*(*xs*, *xp*, *c*) (∗*xp*++ ← (*c*))

223. Ensure there are at least *n* (or 1) bytes left and expand the **XString** to fit if necessary.
#define *XcheckN*(*xs*, *xp*, *n*) **do**
{
ptrdiff_t *more* ← ((*xp*) + (*n*)) − (*xs*).*end*;
if (*more* > 0) *xp* ← *Xcheck_grow_*(&*xs*, *xp*, *more*);
}
while (0)
#define *Xcheck*(*xs*, *xp*) *XcheckN*(*xs*, *xp*, 1)

224. "Close" an **XString** to the bytes that are in use and return a pointer to the C-string.
#define *Xclose*(*xs*, *xp*) *aresize*((*xs*).*beg*, ((*xp*) − (*xs*).*beg*), (*xs*).*areap*)

225. Other obvious **XString** accessors.
#define *Xnleft*(*xs*, *xp*) ((*xs*).*end* − (*xp*)) /∗ may be less than 0 ∗/
#define *Xlength*(*xs*, *xp*) ((*xp*) − (*xs*).*beg*)
#define *Xsize*(*xs*, *xp*) ((*xs*).*end* − (*xs*).*beg*)
#define *Xsavepos*(*xs*, *xp*) ((*xp*) − (*xs*).*beg*) /∗ ie. address to offset ∗/
#define *Xrestpos*(*xs*, *xp*, *n*) ((*xs*).*beg* + (*n*)) /∗ ie. offset to address ∗/

226. A simpler API is implemented for a vector of generic pointers.

```
#define XPinit(x, n) do
        {
          void **vp__;
          vp__ ← areallocarray(Λ, n, sizeof(void *), ATEMP);
          (x).cur ← (x).beg ← vp__;
          (x).end ← vp__ + n;
        }
        while (0)
#define XPput(x, p) do
        {
          if ((x).cur ≥ (x).end) {
            int n ← XPsize(x);
            (x).beg ← areallocarray((x).beg, n, 2 * sizeof(void *), ATEMP);
            (x).cur ← (x).beg + n;
            (x).end ← (x).cur + n;
          }
          *(x).cur++ ← (p);
        }
        while (0)
#define XPptrv(x) ((x).beg)
#define XPsize(x) ((x).cur - (x).beg)
#define XPclose(x) areallocarray((x).beg, XPsize(x), sizeof(void *), ATEMP)
#define XPfree(x) afree((x).beg, ATEMP)
```

⟨ Type definitions 17 ⟩ +≡

```
  typedef struct XPtrV {
    void **cur;      /* next available pointer */
    void **beg, **end;      /* begin, end of vector */
  } XPtrV;
```

227. Scripts & Core Loop. The *command* function will execute a single command by putting it in a **Source** object and calling *shell*.

```
int command(const char *comm, int line)
{
  Source *s;
  s ← pushs(SSTRING, ATEMP);
  s→start ← s→str ← comm;
  s→line ← line;
  return shell(s, false);
}
```

228. To interpret a script the *Include* function accepts a filename and arguments. *intr_ok* is a flag raised when reading the user profile so that users can interrupt the startup process and still get a shell prompt. After setting up error handling and function arguments this similarly calls *shell* with a prepared **Source** object.

```
int Include(const char *name, int argc, char **argv, int intr_ok)
{
  Source *volatile s ← Λ;
  struct Shf *shf;
  char **volatile old_argv;
  volatile int old_argc;
  int i;
  shf ← shf_open(name, O_RDONLY, 0, SHF_MAPHI | SHF_CLEXEC);
  if (shf ≡ Λ) return −1;
  if (argv) {
    old_argv ← genv→loc→argv;
    old_argc ← genv→loc→argc;
  }
  else {
    old_argv ← Λ;
    old_argc ← 0;
  }
  newenv(E_INCL);
  ⟨ Establish an error handler (Include) 229 ⟩
  if (argv) {
    genv→loc→argv ← argv;
    genv→loc→argc ← argc;
  }
  ⟨ Evaluate the included file's contents 230 ⟩
  if (old_argv) {
    genv→loc→argv ← old_argv;
    genv→loc→argc ← old_argc;
  }
  return i & 0xff;      /* &0xff to ensure value not -1 */
}
```

229. Yay! Jump buffers! TODO: Describe jump buffers.

```
#define LRETURN 1     /* return statement */
#define LEXIT 2       /* exit statement */
#define LERROR 3      /* errorf called */
#define LLEAVE 4      /* untrappable exit/error */
#define LINTR 5       /* Ctrl-C noticed */
#define LBREAK 6      /* break statement */
#define LCONTIN 7     /* continue statement */
#define LSHELL 8      /* return to interactive shell */
#define LAEXPR 9      /* error in arithmetic expression */
```

⟨ Establish an error handler (*Include*) 229 ⟩ ≡

```
i ← sigsetjmp(genv→jbuf, 0);
if (i) {
  quitenv(s ? s→u.shf : Λ);
  if (old_argv) {
    genv→loc→argv ← old_argv;
    genv→loc→argc ← old_argc;
  }
  switch (i) {
  case LRETURN: case LERROR: return exstat & 0xff;     /* &0xff to ensure value not -1 */
  case LINTR:
    if (intr_ok ∧ (exstat − 128) ≠ SIGTERM) return 1;
  case LEXIT: case LLEAVE: case LSHELL:     /* else FALLTHROUGH */
    unwind (i);
  default: internal_errorf("%s: %d", __func__, i);
  }
}
```

This code is used in section 228.

230. The file code is evaluated by creating a new **Source** object pointing to the file and passing it to *shell*.

⟨ Evaluate the included file's contents 230 ⟩ ≡

```
s ← pushs(SFILE, ATEMP);
s→u.shf ← shf;
s→file ← str_save(name, ATEMP);
i ← shell(s, false);
quitenv(s→u.shf);
```

This code is used in section 228.

231. After a **Source** has been prepared *shell* will evaluate it. A jump buffer is retained on each invocation so that shell can handle errors when called recursively.

```
int shell(Source *volatile s, volatile int toplevel)
{
  struct Op *t;
  volatile int wastty ← s→flags & SF_TTY;
  volatile int attempts ← 13;
  volatile int interactive ← Flag(FTALKING) ∧ toplevel;
  Source *volatile old_source ← source;
  int i;

  newenv(E_PARSE);
  if (interactive) really_exit ← 0;
  ⟨ Establish an error handler (shell) 232 ⟩
  while (1) {⟨ Compile and interpret an expression 233 ⟩}
  quitenv(Λ);
  source ← old_source;
  return exstat;
}
```

232. In case of interruption by LINTER, LERROR or LSHELL any file or string input consumed so far is discarded. This is "[u]sed by exit command to get back to top level shell. Kind of strange since interactive is set if we are reading from a tty, but to have stopped jobs, one only needs FMONITOR set (not FTALKING/SF_TTY)..."

```
⟨ Establish an error handler (shell) 232 ⟩ ≡
  i ← sigsetjmp(genv→jbuf, 0);
  if (i) {
    switch (i) {
    case LINTR:     /* we get here if SIGINT is not caught or ignored */
    case LERROR: case LSHELL:
      if (interactive) {
        c_fc_reset( );
        if (i ≡ LINTR) shellf("\n");
        if (Flag(FIGNOREEOF) ∧ s→type ≡ SEOF ∧ wastty)
          s→type ← SSTDIN;      /* Reset any eof that was read as part of a multiline command. */
        s→start ← s→str ← null;      /* toss any input we have so far */
        break;
      }
    case LEXIT: case LLEAVE: case LRETURN:      /* else FALLTHROUGH */
      source ← old_source;
      quitenv(Λ);
      unwind (i);      /* keep on unwinding */
    default:
      source ← old_source;
      quitenv(Λ);
      internal_errorf("%s: %d", __func__, i);
    }
  }
```

This code is used in section 231.

233. ⟨ Compile and interpret an expression 233 ⟩ ≡

```
⟨ Check for traps &c. 234 ⟩
t ← compile(s);
if (t ≠ Λ ∧ t→type ≡ TEOF) {⟨ Handle EOF 235 ⟩}
if (t ∧ (¬Flag(FNOEXEC) ∨ (s→flags & SF_TTY))) exstat ← execute(t, 0, Λ);
if (t ≠ Λ ∧ t→type ≠ TEOF ∧ interactive ∧ really_exit) really_exit ← 0;
reclaim( );
```

This code is cited in section 463.

This code is used in section 231.

234. ⟨ Check for traps &c. 234 ⟩ ≡

```
if (trap) runtraps(0);
if (s→next ≡ Λ) {
  if (Flag(FVERBOSE)) s→flags |= SF_ECHO;
  else s→flags &= ∼SF_ECHO;
}
if (interactive) {
  got_sigwinch ← 1;
  j_notify( );
  mcheck( );
  set_prompt(PS1);
}
```

This code is used in section 233.

235. ⟨ Handle EOF 235 ⟩ ≡

```
if (wastty ∧ Flag(FIGNOREEOF) ∧ −−attempts > 0) {
  shellf("Use␣‘exit’␣to␣leave␣ksh\n");
  s→type ← SSTDIN;
}
else if (wastty ∧ ¬really_exit ∧ j_stopped_running( )) {
  really_exit ← 1;
  s→type ← SSTDIN;
}
else {    /* this for POSIX, which says EXIT traps shall be taken in the environment immediately
       after the last command executed. */
  if (toplevel) unwind (LEXIT);
  break;
}
```

This code is used in section 233.

236. Handling Errors. ksh maintains using jump buffers a stack of partial interpretations which it can return –to in disgrace– an error to. If an error handler has been established by *shell* or **include** it will be jumped back to otherwise the shell will exit.

Before looking for an error handler any `ERR` or `EXIT` traps are run. The "ordering [in the test] for EXIT vs ERR is a bit odd (this is what AT&T ksh does)".

```
void unwind (int i)
{
  if (i ≡ LEXIT ∨
        (Flag(FERREXIT) ∧ (i ≡ LERROR ∨ i ≡ LINTR) ∧ sigtraps[SIGEXIT_].trap)) {
    if (trap) runtraps(0);
    runtrap(&sigtraps[SIGEXIT_]);
    i ← LLEAVE;
  }
  else if (Flag(FERREXIT) ∧ (i ≡ LERROR ∨ i ≡ LINTR)) {
    if (trap) runtraps(0);
    runtrap(&sigtraps[SIGERR_]);
    i ← LLEAVE;
  }
  while (1) {
    switch (genv→type) {
    case E_PARSE: case E_FUNC: case E_INCL: case E_LOOP: case E_ERRH:
      siglongjmp (genv→jbuf, i);      /* return to a previous frame */
    case E_NONE:
      if (i ≡ LINTR) genv→flags |= EF_FAKE_SIGDIE;
    default:      /* else FALLTHROUGH */
      quitenv(Λ);
      source ← Λ;      /* quitenv may have reclaimed the memory used by source which will end badly
          when we jump to a function that expects it to be valid */
    }
  }
}
```

237. Input & Output. ksh defines its own file I/O layer atop a minimal subset of `stdio`.

```
#define SHF_BSIZE 512
⟨ Type definitions 17 ⟩ +≡
  struct Shf {
    int flags;      /* see SHF_* */
    unsigned char *rp;      /* read: current position in buffer */
    int rbsize;      /* size of buffer (1 if SHF_UNBUF) */
    int rnleft;      /* read: how much data left in buffer */
    unsigned char *wp;      /* write: current position in buffer */
    int wbsize;      /* size of buffer (0 if SHF_UNBUF) */
    int wnleft;      /* write: how much space left in buffer */
    unsigned char *buf;      /* buffer */
    int fd;      /* file descriptor */
    int errno_;      /* saved value of errno after error */
    int bsize;      /* actual size of buf */
    Area *areap;      /* area shf/buf were allocated in */
  };
```

238. `shf.h` came with include guards.

```
⟨ shf.h 238 ⟩ ≡
#ifndef SHF_H
#define SHF_H
  struct Shf *shf_open(const char *, int, int, int);
  struct Shf *shf_fdopen(int, int, struct Shf *);
  struct Shf *shf_reopen(int, int, struct Shf *);
  struct Shf *shf_sopen(char *, int, int, struct Shf *);
  int shf_close(struct Shf *);
  int shf_fdclose(struct Shf *);
  char *shf_sclose(struct Shf *);
  int shf_flush(struct Shf *);
  int shf_read(char *, int, struct Shf *);
  char *shf_getse(char *, int, struct Shf *);
  int shf_getchar(struct Shf *s);
  int shf_ungetc(int, struct Shf *);
  int shf_putchar(int, struct Shf *);
  int shf_puts(const char *, struct Shf *);
  int shf_write(const char *, int, struct Shf *);
  int shf_fprintf(struct Shf *, const char *, ...);
  int shf_snprintf(char *, int, const char *, ...);
  char *shf_smprintf(const char *, ...);
  int shf_vfprintf(struct Shf *, const char *, va_list);
#endif      /* SHF_H */
```

239. ⟨ shf.c 239 ⟩ ≡

```
#include <sys/stat.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
  static int shf_fillbuf(struct Shf *);
  static int shf_emptybuf(struct Shf *, int);
```

See also sections 245, 247, 249, 251, 252, 253, 254, 274, 275, 281, 282, 283, 284, 285, 286, 287, 288, 289, 294, 295, and 296.

240. ⟨ io.c 240 ⟩ ≡

```
#include <sys/stat.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
  static int initio_done;
  struct Shf shf_iob[3];
#ifdef KSH_DEBUG
  static struct Shf *kshdebug_shf;
#endif    /* KSH_DEBUG */
```

See also sections 250, 255, 256, 257, 258, 259, 260, 261, 265, 266, 267, 268, 269, 270, 272, 297, 298, 299, 300, 301, 302, 303, 304, and 305.

241. ⟨ Global variables 5 ⟩ +≡

```
  int shl_stdout_ok;
```

242. ⟨ Externally-linked variables 6 ⟩ +≡

```
  extern struct Shf shf_iob[];
  extern int shl_stdout_ok;
```

243. ⟨ Shared function declarations 4 ⟩ +≡

```
void errorf(const char *, ...)__attribute__((__noreturn__,__format__(printf,1,2)));
void warningf(bool,const char *, ...)__attribute__((__format__(printf,2,3)));
void bi_errorf(const char *, ...)__attribute__((__format__(printf,1,2)));
void internal_errorf(const char *, ...)__attribute__((__noreturn__,__format__(printf,1,2)));
void internal_warningf(const char *, ...)__attribute__((__format__(printf,1,2)));
void error_prefix(int);
void shellf(const char *, ...)__attribute__((__format__(printf,1,2)));
void shprintf(const char *, ...)__attribute__((__format__(printf,1,2)));
#ifdef KSH_DEBUG
void kshdebug_init_(void);
void kshdebug_printf_(const char *, ...)__attribute__((__format__(printf,1,2)));
void kshdebug_dump_(const char *,const void *,int);
#endif      /* KSH_DEBUG */
int can_seek(int);
void initio(void);
int ksh_dup2(int,int,int);
int savefd(int);
void restfd(int,int);
void openpipe(int *);
void closepipe(int *);
int check_fd(char *,int,const char **);
void coproc_init(void);
void coproc_read_close(int);
void coproc_readw_close(int);
void coproc_write_close(int);
int coproc_getfd(int,const char **);
void coproc_cleanup(int);
struct Temp *maketemp(Area *, Temp_type,struct Temp **);
```

244. Macro interfaces to the functions below.

```
#define shf_getc(shf)  ((shf)→rnleft > 0
                 ? (shf)→rnleft --, *(shf)→rp ++
                 : shf_getchar(shf))
#define shf_putc(c,shf)  ((shf)→wnleft ≡ 0
                 ? shf_putchar((c),(shf))
                 : ((shf)→wnleft --, *(shf)→wp ++ ← (c)))
#define shf_eof(shf)  ((shf)→flags & SHF_EOF)
#define shf_error(shf)  ((shf)→flags & SHF_ERROR)
#define shf_clearerr(shf)  ((shf)→flags &= ∼(SHF_EOF | SHF_ERROR))
```

245. Opening a file is the same as `stdio` with some extra flags. Memory is allocated first so the file descriptor won't be lost if it fails then *open* is passed the first three arguments. The access mode flags `SHF_RD` & `SHF_WR` are ignored by *shf_open* in favour of the equivalent *open* flags in *oflags*.

struct Shf has been renamed **struct Shf** for cosmetic reasons.

```
#define SHF_RD 0x0001
#define SHF_WR 0x0002
#define SHF_RDWR (SHF_RD | SHF_WR)
#define SHF_ACCMODE 0x0003 /* mask */
#define SHF_GETFL 0x0004 /* use fcntl to figure RD/WR flags */
#define SHF_UNBUF 0x0008 /* unbuffered I/O */
#define SHF_CLEXEC 0x0010 /* set close-on-exec flag */
#define SHF_MAPHI 0x0020 /* make fd > FDBASE (and close orig; for shf_open only */
#define SHF_DYNAMIC 0x0040 /* string: increase buffer as needed */
#define SHF_INTERRUPT 0x0080 /* EINTR in read/write causes error */
 /* The remaining flags are used internally */
#define SHF_STRING 0x0100 /* a string, not a file */
#define SHF_ALLOCS 0x0200 /* shf and shf→buf were alloced */
#define SHF_ALLOCB 0x0400 /* shf→buf was alloced */
#define SHF_ERROR 0x0800 /* read/write error */
#define SHF_EOF 0x1000 /* read eof (sticky) */
#define SHF_READING 0x2000 /* currently reading: rnleft & rp are valid */
#define SHF_WRITING 0x4000 /* currently writing: wnleft & wp are valid */
```

⟨ shf.c 239 ⟩ +≡

```
struct Shf *shf_open(const char *name, int oflags, int mode, int sflags)
{
  struct Shf *shf;
  int bsize ← sflags & SHF_UNBUF ? (sflags & SHF_RD ? 1 : 0) : SHF_BSIZE;
  int fd;
  shf ← alloc(sizeof(struct Shf) + bsize, ATEMP);
  shf→areap ← ATEMP;
  shf→buf ← (unsigned char *) &shf[1];
  shf→bsize ← bsize;
  shf→flags ← SHF_ALLOCS; /* The rest of shf is filled in by reopen. */
  fd ← open(name, oflags, mode);
  if (fd ≡ −1) {
    afree(shf, shf→areap);
    return Λ;
  }
  if ((sflags & SHF_MAPHI) ∧ fd < FDBASE) {⟨ Move the file descriptor to FDBASE or above 246 ⟩}
  sflags &= ∼SHF_ACCMODE;
  sflags |= (oflags & O_ACCMODE) ≡ O_RDONLY ? SHF_RD :
      ((oflags & O_ACCMODE) ≡ O_WRONLY ? SHF_WR : SHF_RDWR);
  return shf_reopen(fd, sflags, shf);
}
```

246. ⟨ Move the file descriptor to FDBASE or above 246 ⟩ ≡

```
int nfd;
nfd ← fcntl(fd, F_DUPFD, FDBASE);
close(fd);
if (nfd ≡ −1) {
  afree(shf, shf→areap);
  return Λ;
}
fd ← nfd;
```

This code is used in section 245.

247. Despite its name *shf_reopen* doesn't open a file but adjusts the flags and buffers of an existing **Shf** object. It assumes *shf→buf* and *shf→bsize* are already set up.

⟨ shf.c 239 ⟩ +≡

```
struct Shf *shf_reopen(int fd, int sflags, struct Shf *shf)
{
  int bsize ← sflags & SHF_UNBUF ? (sflags & SHF_RD ? 1 : 0) : SHF_BSIZE;
  if (sflags & SHF_GETFL) {⟨ Use fcntl to figure out the read/write flags 248 ⟩}
  if (¬(sflags & (SHF_RD | SHF_WR))) internal_errorf("%s: missing read/write", __func__);
  if (¬shf ∨ ¬shf→buf ∨ shf→bsize < bsize) internal_errorf("%s: bad shf/buf/bsize", __func__);
  shf→fd ← fd;
  shf→rp ← shf→wp ← shf→buf;
  shf→rnleft ← 0;
  shf→rbsize ← bsize;
  shf→wnleft ← 0;      /* force call to shf_emptybuf */
  shf→wbsize ← sflags & SHF_UNBUF ? 0 : bsize;
  shf→flags ← (shf→flags & (SHF_ALLOCS | SHF_ALLOCB)) | sflags;
  shf→errno_ ← 0;
  if (sflags & SHF_CLEXEC) fcntl(fd, F_SETFD, FD_CLOEXEC);
  return shf;
}
```

248. ⟨ Use *fcntl* to figure out the read/write flags 248 ⟩ ≡

```
int flags ← fcntl(fd, F_GETFL);
if (flags ≡ −1) sflags |= SHF_RDWR;      /* will get an error on first read/write */
else {
  switch (flags & O_ACCMODE) {
  case O_RDONLY: sflags |= SHF_RD;
    break;
  case O_WRONLY: sflags |= SHF_WR;
    break;
  case O_RDWR: sflags |= SHF_RDWR;
    break;
  }
}
```

This code is used in sections 247 and 249.

249. An open file descriptor can be contained in an **Shf** object with *shf_fdopen*.

```
⟨ shf.c 239 ⟩ +≡
  struct Shf *shf_fdopen(int fd, int sflags, struct Shf *shf)
  {
    int bsize ← sflags & SHF_UNBUF ? (sflags & SHF_RD ? 1 : 0) : SHF_BSIZE;
    if (sflags & SHF_GETFL) {⟨ Use fcntl to figure out the read/write flags 248 ⟩}
    if (¬(sflags & (SHF_RD | SHF_WR))) internal_errorf("%s: missing read/write", __func__);
    if (shf) {
      if (bsize) {
        shf→buf ← alloc(bsize, ATEMP);
        sflags |= SHF_ALLOCB;
      }
      else shf→buf ← Λ;
    }
    else {
      shf ← alloc(sizeof(struct Shf) + bsize, ATEMP);
      shf→buf ← (unsigned char *) &shf[1];
      sflags |= SHF_ALLOCS;
    }
    shf→areap ← ATEMP;
    shf→fd ← fd;
    shf→rp ← shf→wp ← shf→buf;
    shf→rnleft ← 0;
    shf→rbsize ← bsize;
    shf→wnleft ← 0;      /* force call to shf_emptybuf */
    shf→wbsize ← sflags & SHF_UNBUF ? 0 : bsize;
    shf→flags ← sflags;
    shf→errno_ ← 0;
    shf→bsize ← bsize;
    if (sflags & SHF_CLEXEC) fcntl(fd, F_SETFD, FD_CLOEXEC);
    return shf;
  }
```

250. This is used to initialise the I/O routines by allocating buffers for standard output and error.

```
#define shl_spare  (&shf_iob[0])      /* for c_read/c_print */
#define shl_stdout  (&shf_iob[1])
#define shl_out  (&shf_iob[2])
⟨ io.c 240 ⟩ +≡
  void initio(void)
  {
    shf_fdopen(1, SHF_WR, shl_stdout);
    shf_fdopen(2, SHF_WR, shl_out);
    shf_fdopen(2, SHF_WR, shl_spare);
    initio_done ← 1;
    kshdebug_init( );
  }
```

251. A string can be read or written to (but not both) as an **Shf** object. If reading, *bsize* is the number of bytes that can be read. If writing, the maximum number of bytes that can be written. If *shf* is Λ a new **Shf** is allocated and marked as such.

When writing an extra byte is reserved for a trailing `'\0'`—see *shf_sclose*. Additionally if *buf* is Λ and `SHF_DYNAMIC` is set, the buffer is allocated (if *bsize* > 0 it is used for the initial size).

```
⟨ shf.c 239 ⟩ +≡
  struct Shf *shf_sopen(char *buf, int bsize, int sflags, struct Shf *shf)
  {
    if (¬(sflags & (SHF_RD | SHF_WR)) ∨ (sflags & (SHF_RD | SHF_WR)) ≡ (SHF_RD | SHF_WR))
        /* can't have a read&write string */
      internal_errorf("%s: flags 0x%x", __func__, sflags);
    if (¬shf) {
      shf ← alloc(sizeof(struct Shf), ATEMP);
      sflags |= SHF_ALLOCS;
    }
    shf→areap ← ATEMP;
    if (¬buf ∧ (sflags & SHF_WR) ∧ (sflags & SHF_DYNAMIC)) {
      if (bsize ≤ 0) bsize ← 64;
      sflags |= SHF_ALLOCB;
      buf ← alloc(bsize, shf→areap);
    }
    shf→fd ← −1;
    shf→buf ← shf→rp ← shf→wp ← (unsigned char *) buf;
    shf→rnleft ← bsize;
    shf→rbsize ← bsize;
    shf→wnleft ← bsize − 1;      /* space for a '\0' */
    shf→wbsize ← bsize;
    shf→flags ← sflags | SHF_STRING;
    shf→errno_ ← 0;
    shf→bsize ← bsize;
    return shf;
  }
```

252. A **Shf** object is freed and the underlying file descriptor closed with *shf_close*. First any pending buffers are flushed.

```
⟨ shf.c 239 ⟩ +≡
  int shf_close(struct Shf *shf)
  {
    int ret ← 0;
    if (shf→fd ≥ 0) {
      ret ← shf_flush(shf);
      if (close(shf→fd) ≡ −1) ret ← EOF;
    }
    if (shf→flags & SHF_ALLOCS) afree(shf, shf→areap);
    else if (shf→flags & SHF_ALLOCB) afree(shf→buf, shf→areap);
    return ret;
  }
```

253. Alternatively this closes the file descriptor without freeing the **Shf** object.

```
⟨ shf.c 239 ⟩ +≡
  int shf_fdclose(struct Shf *shf)
  {
    int ret ← 0;
    if (shf→fd ≥ 0) {
      ret ← shf_flush(shf);
      if (close(shf→fd) ≡ −1) ret ← EOF;
      shf→rnleft ← 0;
      shf→rp ← shf→buf;
      shf→wnleft ← 0;
      shf→fd ← −1;
    }
    return ret;
  }
```

254. "Closing" a string will not free the string buffer unless it was allocated by *shf_sopen*. It will be terminated with `'\0` if it was for writing. A pointer to the string is returned.

```
⟨ shf.c 239 ⟩ +≡
  char *shf_sclose(struct Shf *shf)
  {
    unsigned char *s ← shf→buf;      /* null terminate */
    if (shf→flags & SHF_WR) {
      shf→wnleft++;
      shf_putc('\0', shf);
    }
    if (shf→flags & SHF_ALLOCS) afree(shf, shf→areap);
    return (char *) s;
  }
```

255. File Descriptors. This routine tests if seeking backwards is possible.

```
⟨ io.c 240 ⟩ +≡
  int can_seek(int fd)
  {
    struct stat statb;
    return fstat(fd, &statb) ≡ 0 ∧ ¬S_ISREG(statb.st_mode) ? SHF_UNBUF : 0;
  }
```

256. A wrapper around *dup2* with error checking.

```
⟨ io.c 240 ⟩ +≡
  int ksh_dup2(int ofd, int nfd, int errok)
  {
    int ret ← dup2(ofd, nfd);
    if (ret ≡ −1 ∧ errno ≠ EBADF ∧ ¬errok) errorf("too␣many␣files␣open␣in␣shell");
    return ret;
  }
```

257. Moves a file descriptor from "user space" ($0 \le fd < 10$) to "shell space" ($fd \ge 10$) and sets the close-on-*exec* flag.

```
⟨ io.c 240 ⟩ +≡
  int savefd(int fd)
  {
    int nfd;
    if (fd < FDBASE) {
      nfd ← fcntl(fd, F_DUPFD_CLOEXEC, FDBASE);
      if (nfd ≡ −1) {
        if (errno ≡ EBADF) return −1;
        else errorf("too␣many␣files␣open␣in␣shell");
      }
    }
    else {
      nfd ← fd;
      fcntl(nfd, F_SETFD, FD_CLOEXEC);
    }
    return nfd;
  }
```

258. 'Restore'; undoes the effect of *savefd*.

```
⟨ io.c 240 ⟩ +≡
  void restfd(int fd, int ofd)
  {
    if (fd ≡ 2) shf_flush(&shf_iob[fd]);
    if (ofd < 0) close(fd);      /* original fd closed */
    else if (fd ≠ ofd) {
      ksh_dup2(ofd, fd, true);      /* XXX: what to do if this fails? */
      close(ofd);
    }
  }
```

259. Open a pipe and put its file descriptors—moved to shell space—in *pv*[ ].

```
⟨ io.c 240 ⟩ +≡
  void openpipe(int *pv)
  {
    int lpv[2];
    if (pipe(lpv) ≡ −1) errorf("can't␣create␣pipe␣-␣try␣again");
    pv[0] ← savefd(lpv[0]);
    if (pv[0] ≠ lpv[0]) close(lpv[0]);
    pv[1] ← savefd(lpv[1]);
    if (pv[1] ≠ lpv[1]) close(lpv[1]);
  }
```

260.
```
⟨ io.c 240 ⟩ +≡
  void closepipe(int *pv)
  {
    close(pv[0]);
    close(pv[1]);
  }
```

261. Called to set up I/O redirection (`Y>&X`) or read/write from/to a file descriptor (`-uX`) by turning the string `X` into a file descriptor.

Contrary to its usual definition `X_OK` is (mis-)used by ksh here to skip testing a file descriptor's mode bits "for dups (`x<&1`)". The author of this function also notes that historical shells never did this check and was unaware what POSIX said on the matter.

```
⟨ io.c 240 ⟩ +≡
  int check_fd(char *name, int mode, const char **emsgp)
  {
    int fd, fl;
    if (isdigit((unsigned char) name[0]) ∧ ¬name[1]) {
      fd ← name[0] − '0';
      if ((fl ← fcntl(fd, F_GETFL)) ≡ −1) {
        if (emsgp) *emsgp ← "bad␣file␣descriptor";
        return −1;
      }
      fl &= O_ACCMODE;
      if (¬(mode & X_OK) ∧
            fl ≠ O_RDWR ∧
            (((mode & R_OK) ∧ fl ≠ O_RDONLY) ∨ ((mode & W_OK) ∧ fl ≠ O_WRONLY))) {
        if (emsgp)
          *emsgp ← (fl ≡ O_WRONLY) ? "fd␣not␣open␣for␣reading" : "fd␣not␣open␣for␣writing";
        return −1;
      }
      return fd;
    }
    else if (name[0] ≡ 'p' ∧ ¬name[1]) return coproc_getfd(mode, emsgp);
    if (emsgp) *emsgp ← "illegal␣file␣descriptor␣name";
    return −1;
  }
```

262. Co-Process File Descriptors. Co-processess, described later, are two processes which communicate with each other bi-directionally over a pair of pipes. The routines here handle the file descriptors in each process. First initialisation from *main*.

```
⟨ Type definitions 17 ⟩ +≡
  typedef int Coproc_id;      /* something that won't (realistically) wrap */
  struct Coproc {
    int read;      /* pipe from co-process's stdout */
    int readw;     /* other side of read (saved temporarily) */
    int write;     /* pipe to co-process's stdin */
    Coproc_id id;      /* id of current output pipe */
    int njobs;     /* number of live jobs using output pipe */
    void *job;     /* 0 or job of co-process using input pipe */
  };
```

263.
```
⟨ Global variables 5 ⟩ +≡
  struct Coproc coproc;
```

264.
```
⟨ Externally-linked variables 6 ⟩ +≡
  extern struct Coproc coproc;
```

265.
```
⟨ io.c 240 ⟩ +≡
  void coproc_init(void)
  {
    coproc.read ← coproc.readw ← coproc.write ← −1;
    coproc.njobs ← 0;
    coproc.id ← 0;
  }
```

266. When *c_read* encounters EOF it calls this to close the file descriptor if it is associated with the co-process. See the description in *c_read* for more details.

```
⟨ io.c 240 ⟩ +≡
  void coproc_read_close(int fd)
  {
    if (coproc.read ≥ 0 ∧ fd ≡ coproc.read) {
      coproc_readw_close(fd);
      close(coproc.read);
      coproc.read ← −1;
    }
  }
```

267. Closes the other side of the read pipe so that reads will terminate.

```
⟨ io.c 240 ⟩ +≡
  void coproc_readw_close(int fd)
  {
    if (coproc.readw ≥ 0 ∧ coproc.read ≥ 0 ∧ fd ≡ coproc.read) {
      close(coproc.readw);
      coproc.readw ← −1;
    }
  }
```

268. Closes the write pipe.

```
⟨ io.c 240 ⟩ +≡
  void coproc_write_close(int fd)
  {
    if (coproc.write ≥ 0 ∧ fd ≡ coproc.write) {
      close(coproc.write);
      coproc.write ← −1;
    }
  }
```

269. Gets (and thus checks its validity for *check_fd*) the co-process file descriptor.

```
⟨ io.c 240 ⟩ +≡
  int coproc_getfd(int mode, const char **emsgp)
  {
    int fd ← (mode & R_OK) ? coproc.read : coproc.write;
    if (fd ≥ 0) return fd;
    if (emsgp) *emsgp ← "no␣coprocess";
    return −1;
  }
```

270. Cleans up all file descriptors associated with co-processes, if any.

```
⟨ io.c 240 ⟩ +≡
  void coproc_cleanup(int reuse)
  {    /* This to allow co-processes to share output pipe */
    if (¬reuse ∨ coproc.readw < 0 ∨ coproc.read < 0) {
      if (coproc.read ≥ 0) {
        close(coproc.read);
        coproc.read ← −1;
      }
      if (coproc.readw ≥ 0) {
        close(coproc.readw);
        coproc.readw ← −1;
      }
    }
    if (coproc.write ≥ 0) {
      close(coproc.write);
      coproc.write ← −1;
    }
  }
```

271. Temporary Files. Used also for heredocs temporary files are removed when the object is freed.

```
⟨ Type definitions 17 ⟩ +≡
  enum temp_type {
    TT_HEREDOC_EXP,      /* expanded heredoc */
    TT_HIST_EDIT      /* used for history editing (fc -e) */
  };
  typedef enum temp_type Temp_type;
  struct Temp {
    struct Temp *next;
    struct Shf *shf;
    int pid;      /* pid of process parsed here-doc */
    Temp_type type;
    char *name;
  };
```

272.

```
⟨ io.c 240 ⟩ +≡
  struct Temp *maketemp(Area *ap, Temp_type type, struct Temp **tlist)
  {
    struct Temp *tp;
    int len;
    int fd;
    char *path;
    const char *dir;
    dir ← tmpdir ? tmpdir : "/tmp";
    len ← strlen(dir) + 3 + 20 + 20 + 1;      /* The 20 + 20 is a paranoid worst case for pid/inc */
    tp ← alloc(sizeof(struct Temp) + len, ap);
    tp→name ← path ← (char *) &tp[1];
    tp→shf ← Λ;
    tp→type ← type;
    shf_snprintf(path, len, "%s/shXXXXXXXX", dir);
    fd ← mkstemp(path);
    if (fd ≥ 0) tp→shf ← shf_fdopen(fd, SHF_WR, Λ);
    tp→pid ← procpid;
    tp→next ← *tlist;
    *tlist ← tp;
    return tp;
  }
```

273.

```
  static void remove_temps(struct Temp *tp)
  {
    for ( ; tp ≠ Λ; tp ← tp→next)
      if (tp→pid ≡ procpid) {
        unlink(tp→name);
      }
  }
```

274. Buffers. Every file object (optionally) has a single buffer which is used for either reading or writing, which determined contextually and by the values *rnleft*/*wdleft*.

A buffer that's being used for reading is flushed (after checking for error states) by clearing it and seeking back the appropriate length in the file. The reading (and EOF) flag is turned off.

```
⟨ shf.c 239 ⟩ +≡
  int shf_flush(struct Shf *shf)
  {
    if (shf→flags & SHF_STRING) return (shf→flags & SHF_WR) ? EOF : 0;
    if (shf→fd < 0) internal_errorf("%s: no fd", __func__);
    if (shf→flags & SHF_ERROR) {
      errno ← shf→errno_;
      return EOF;
    }
    if (shf→flags & SHF_READING) {
      shf→flags &= ∼(SHF_EOF | SHF_READING);
      if (shf→rnleft > 0) {
        lseek(shf→fd, (off_t) − shf→rnleft, SEEK_CUR);
        shf→rnleft ← 0;
        shf→rp ← shf→buf;
      }
      return 0;
    }
    else if (shf→flags & SHF_WRITING) return shf_emptybuf(shf, 0);
    return 0;
  }
```

275. Write buffers instead fall back to the more generic *shf_emptybuf*. Since *shf_flush* calls this if the buffer is being used for writing and this calls *shf_flush* if the buffer is used for writing they can possibly be used interchangably depending on exactly how the buffer interacts with SHF_READING & SHF_WRITING.

```
#define EB_READSW 0x01    /* about to switch to reading */
#define EB_GROW 0x02      /* grow buffer if necessary (STRING&DYNAMIC) */
```

```
⟨ shf.c 239 ⟩ +≡
  static int shf_emptybuf(struct Shf *shf, int flags)
  {
    int ret ← 0;
    if (¬(shf→flags & SHF_STRING) ∧ shf→fd < 0) internal_errorf("%s: no fd", __func__);
    if (shf→flags & SHF_ERROR) {
      errno ← shf→errno_;
      return EOF;
    }
    if (shf→flags & SHF_READING) {
      if (flags & EB_READSW) return 0;      /* doesn't happen */
      ret ← shf_flush(shf);
      shf→flags &= ∼SHF_READING;
    }
    if (shf→flags & SHF_STRING) {⟨ Flush a string's read/write buffers 276 ⟩}
    else {⟨ Flush a file's read/write buffers 277 ⟩}
    shf→flags |= SHF_WRITING;
    return ret;
  }
```

276. A string **Shf** doesn't really have a buffer—what would be considered the buffer is the string itself. It is "flushed" by enlarging the buffer to twice *wbsize*. It's up to the caller to understand the context of the pointers.

```
⟨ Flush a string's read/write buffers 276 ⟩ ≡
  unsigned char *nbuf;
  if (¬(flags & EB_GROW) ∨ ¬(shf→flags & SHF_DYNAMIC) ∨ ¬(shf→flags & SHF_ALLOCB)) return EOF;
  nbuf ← areallocarray(shf→buf, 2, shf→wbsize, shf→areap);
  shf→rp ← nbuf + (shf→rp − shf→buf);
  shf→wp ← nbuf + (shf→wp − shf→buf);
  shf→rbsize += shf→wbsize;
  shf→wnleft += shf→wbsize;
  shf→wbsize *= 2;
  shf→buf ← nbuf;
```

This code is used in section 275.

277. *write* is called repeatedly until it fails or there are no bytes left in the buffer. If the caller indicates that the file object is about to be used for reading (EB_READSW) then SHF_WRITING is turned off and the write buffer cleared. Otherwise *wnleft* indicates that the buffer is empty.

```
⟨ Flush a file's read/write buffers 277 ⟩ ≡
  if (shf→flags & SHF_WRITING) {
    int ntowrite ← shf→wp − shf→buf;
    unsigned char *buf ← shf→buf;
    int n;
    while (ntowrite > 0) {
      n ← write(shf→fd, buf, ntowrite);
      if (n ≡ −1) {⟨ Possibly re-attempt an interrupted flush, or return 278 ⟩}
      buf += n;
      ntowrite −= n;
    }
    if (flags & EB_READSW) {
      shf→wp ← shf→buf;
      shf→wnleft ← 0;
      shf→flags &= ∼SHF_WRITING;
      return 0;
    }
  }
  shf→wp ← shf→buf;
  shf→wnleft ← shf→wbsize;
```

This code is used in section 275.

278. If the *write* system call was interrupted it's retried unless the SHF_INTERRUPT flag is set. Other errors are saved in *errno_*. If at least one *write* call has succeeded then the appropriate amount is removed from the buffer.

```
⟨ Possibly re-attempt an interrupted flush, or return 278 ⟩ ≡
  if (errno ≡ EINTR ∧ ¬(shf→flags & SHF_INTERRUPT)) continue;
  shf→flags |= SHF_ERROR;
  shf→errno_ ← errno;
  shf→wnleft ← 0;
  if (buf ≠ shf→buf) {      /* allow a second flush to work */
    memmove(shf→buf, buf, ntowrite);
    shf→wp ← shf→buf + ntowrite;
  }
  return EOF;
```

This code is used in section 277.

279. Also used by the advanced edit modes *blocking_read* will attempt to *read* and, if it was set to non-blocking mode and no data were available, non-blocking mode is disabled and *read* retried.

```
⟨ misc.c 9 ⟩ +≡
  int blocking_read(int fd, char *buf, int nbytes)
  {
    int ret;
    int tried_reset ← 0;
    while ((ret ← read(fd, buf, nbytes)) ≡ −1) {
      if (¬tried_reset ∧ errno ≡ EAGAIN) {
        int oerrno ← errno;
        if (reset_nonblock(fd) > 0) {
          tried_reset ← 1;
          continue;
        }
        errno ← oerrno;
      }
      break;
    }
    return ret;
  }
```

280. If non-blocking mode was enabled it is turned off and 1 returned. POSIX also requires this routine during initialisation (but not the return value).

```
⟨ misc.c 9 ⟩ +≡
  int reset_nonblock(int fd)
  {
    int flags;
    if ((flags ← fcntl(fd, F_GETFL)) ≡ −1) return −1;
    if (¬(flags & O_NONBLOCK)) return 0;
    flags &= ∼O_NONBLOCK;
    if (fcntl(fd, F_SETFL, flags) ≡ −1) return −1;
    return 1;
  }
```

281. To fill a buffer for reading from it's first flushed if it's currently being used for writing then the underlying file read from and errors/EOF detected. If the read system call was interrupted it's retried.

```
⟨ shf.c 239 ⟩ +≡
  static int shf_fillbuf(struct Shf *shf)
  {
    if (shf→flags & SHF_STRING) return 0;
    if (shf→fd < 0) internal_errorf("%s: no fd", __func__);
    if (shf→flags & (SHF_EOF | SHF_ERROR)) {
      if (shf→flags & SHF_ERROR) errno ← shf→errno_;
      return EOF;
    }
    if ((shf→flags & SHF_WRITING) ∧ shf_emptybuf(shf, EB_READSW) ≡ EOF) return EOF;
    shf→flags |= SHF_READING;
    shf→rp ← shf→buf;
    while (1) {
      shf→rnleft ← blocking_read(shf→fd, (char *) shf→buf, shf→rbsize);
      if (shf→rnleft < 0 ∧ errno ≡ EINTR ∧ ¬(shf→flags & SHF_INTERRUPT)) continue;
      break;
    }
    if (shf→rnleft ≤ 0) {
      if (shf→rnleft < 0) {
        shf→flags |= SHF_ERROR;
        shf→errno_ ← errno;
        shf→rnleft ← 0;
        shf→rp ← shf→buf;
        return EOF;
      }
      shf→flags |= SHF_EOF;
    }
    return 0;
  }
```

282. Reading & Writing. *shf_read* repeatedly calls *shf_fillbuf* and copies the result into *buf* until *bsize* bytes have been copied, there are none left to copy or there was an error.

Note that *fread*(3) returns 0 for errors and this does not.

```
⟨ shf.c 239 ⟩ +≡
  int shf_read(char *buf, int bsize, struct Shf *shf)
  {
    int orig_bsize ← bsize;
    int ncopy;
    if (¬(shf→flags & SHF_RD)) internal_errorf("%s: flags %x", __func__, shf→flags);
    if (bsize ≤ 0) internal_errorf("%s: bsize %d", __func__, bsize);
    while (bsize > 0) {
      if (shf→rnleft ≡ 0 ∧ (shf_fillbuf(shf) ≡ EOF ∨ shf→rnleft ≡ 0)) break;
      ncopy ← shf→rnleft;
      if (ncopy > bsize) ncopy ← bsize;
      memcpy(buf, shf→rp, ncopy);
      buf += ncopy;
      bsize -= ncopy;
      shf→rp += ncopy;
      shf→rnleft -= ncopy;
    }
    return orig_bsize ≡ bsize ? (shf_error(shf) ? EOF : 0) : orig_bsize - bsize;
  }
```

283. Similarly *shf_getse* fills *buf* with a single line including the terminating newline character (unless EOF was reached) and `'\0'` (so $bsize - 2$ bytes are available in *buf*).

A pointer to the terminating `'\0'` is returned.

```
⟨ shf.c 239 ⟩ +≡
  char *shf_getse(char *buf, int bsize, struct Shf *shf)
  {
    unsigned char *end;
    int ncopy;
    char *orig_buf ← buf;
    if (¬(shf→flags & SHF_RD)) internal_errorf("%s: flags %x", __func__, shf→flags);
    if (bsize ≤ 0) return Λ;
    −−bsize;      /* save room for '\0' */
    do {
      if (shf→rnleft ≡ 0) {
        if (shf_fillbuf(shf) ≡ EOF) return Λ;
        if (shf→rnleft ≡ 0) {
          *buf ← '\0';
          return buf ≡ orig_buf ? Λ : buf;
        }
      }
      end ← (unsigned char *) memchr((char *) shf→rp, '\n', shf→rnleft);
      ncopy ← end ? end − shf→rp + 1 : shf→rnleft;
      if (ncopy > bsize) ncopy ← bsize;
      memcpy(buf, (char *) shf→rp, ncopy);
      shf→rp += ncopy;
      shf→rnleft −= ncopy;
      buf += ncopy;
      bsize −= ncopy;
    } while (¬end ∧ bsize);
    *buf ← '\0';
    return buf;
  }
```

284. Finally a single character is fetched by removing it from the read buffer.

```
⟨ shf.c 239 ⟩ +≡
  int shf_getchar(struct Shf *shf)
  {
    if (¬(shf→flags & SHF_RD)) internal_errorf("%s: flags %x", __func__, shf→flags);
    if (shf→rnleft ≡ 0 ∧ (shf_fillbuf(shf) ≡ EOF ∨ shf→rnleft ≡ 0)) return EOF;
    −−shf→rnleft;
    return *shf→rp++;
  }
```

285. Put a character back in the input stream. Returns the character if successful or EOF if there is no room or the underlying SHF_STRING has changed.

```
⟨ shf.c 239 ⟩ +≡
  int shf_ungetc(int c, struct Shf *shf)
  {
    if (¬(shf→flags & SHF_RD)) internal_errorf("%s: flags %x", __func__, shf→flags);
    if ((shf→flags & SHF_ERROR) ∨ c ≡ EOF ∨ (shf→rp ≡ shf→buf ∧ shf→rnleft)) return EOF;
    if ((shf→flags & SHF_WRITING) ∧ shf_emptybuf(shf, EB_READSW) ≡ EOF) return EOF;
    if (shf→rp ≡ shf→buf) shf→rp ← shf→buf + shf→rbsize;
    if (shf→flags & SHF_STRING) {
      if (shf→rp[−1] ≠ c) return EOF;      /* Can unget what was read, but not something different—we
            don't want to modify a string. */
      shf→flags &= ∼SHF_EOF;
      shf→rp−−;
      shf→rnleft++;
      return c;
    }
    shf→flags &= ∼SHF_EOF;
    *−−(shf→rp) ← c;
    shf→rnleft++;
    return c;
  }
```

286. Write a character. Returns the character if successful, EOF if the char could not be written.

```
⟨ shf.c 239 ⟩ +≡
  int shf_putchar(int c, struct Shf *shf)
  {
    if (¬(shf→flags & SHF_WR)) internal_errorf("%s: flags %x", __func__, shf→flags);
    if (c ≡ EOF) return EOF;
    if (shf→flags & SHF_UNBUF) {
      char cc ← c;
      int n;
      if (shf→fd < 0) internal_errorf("%s: no fd", __func__);
      if (shf→flags & SHF_ERROR) {
        errno ← shf→errno_;
        return EOF;
      }
      while ((n ← write(shf→fd, &cc, 1)) ≠ 1)
        if (n ≡ −1) {
          if (errno ≡ EINTR ∧ ¬(shf→flags & SHF_INTERRUPT)) continue;
          shf→flags |= SHF_ERROR;
          shf→errno_ ← errno;
          return EOF;
        }
    }
    else {
      if (shf→wnleft ≡ 0 ∧ shf_emptybuf(shf, EB_GROW) ≡ EOF)
        return EOF;      /* Flush (shf_emptybuf) deals with strings and sticky errors */
      shf→wnleft −−;
      *shf→wp ++ ← c;
    }
    return c;
  }
```

287. Write a buffer. Returns nbytes if successful, EOF if there is an error.
Note that *fwrite*(3) returns 0 for errors and this does not.

```
⟨ shf.c 239 ⟩ +≡
  int shf_write(const char *buf, int nbytes, struct Shf *shf)
  {
    int orig_nbytes ← nbytes;
    int n;
    int ncopy;
    if (¬(shf→flags & SHF_WR)) internal_errorf("%s: flags %x", __func__, shf→flags);
    if (nbytes < 0) internal_errorf("%s: nbytes %d", __func__, nbytes);
        /* Don't buffer if buffer is empty and we're writting a large amount. */
    if ((ncopy ← shf→wnleft) ∧ (shf→wp ≠ shf→buf ∨ nbytes < shf→wnleft)) {
      if (ncopy > nbytes) ncopy ← nbytes;
      memcpy(shf→wp, buf, ncopy);
      nbytes −= ncopy;
      buf += ncopy;
      shf→wp += ncopy;
      shf→wnleft −= ncopy;
    }
    if (nbytes > 0) {
      if (shf_emptybuf(shf, EB_GROW) ≡ EOF) return EOF;      /* Flush deals with strings and errors */
      if (nbytes > shf→wbsize) {
        ncopy ← nbytes;
        if (shf→wbsize) ncopy −= nbytes % shf→wbsize;
        nbytes −= ncopy;
        while (ncopy > 0) {
          n ← write(shf→fd, buf, ncopy);
          if (n ≡ −1) {
            if (errno ≡ EINTR ∧ ¬(shf→flags & SHF_INTERRUPT)) continue;
            shf→flags |= SHF_ERROR;
            shf→errno_ ← errno;
            shf→wnleft ← 0;
            return EOF;
          }
          buf += n;
          ncopy −= n;
        }
      }
      if (nbytes > 0) {
        memcpy(shf→wp, buf, nbytes);
        shf→wp += nbytes;
        shf→wnleft −= nbytes;
      }
    }
    return orig_nbytes;
  }
```

288. Write a string. Returns the length of the string if successful, EOF if the string could not be written.

```
⟨ shf.c 239 ⟩ +≡
  int shf_puts(const char *s, struct Shf *shf)
  {
    if (¬s) return EOF;
    return shf_write(s, strlen(s), shf);
  }
```

289. Formatted Output. Because who doesn't want a custom printf?

```
#define FL_HASH 0x001 /* '#' seen */
#define FL_PLUS 0x002 /* '+' seen */
#define FL_RIGHT 0x004 /* '-' seen */
#define FL_BLANK 0x008 /* ' ' seen */
#define FL_SHORT 0x010 /* 'h' seen */
#define FL_LONG 0x020 /* 'l' seen */
#define FL_LLONG 0x040 /* 'll' seen */
#define FL_ZERO 0x080 /* '0' seen */
#define FL_DOT 0x100 /* '.' seen */
#define FL_UPPER 0x200 /* format character was uppercase */
#define FL_NUMBER 0x400 /* a number was formatted (one of defgoux) */
```

```
⟨ shf.c 239 ⟩ +≡
  int shf_vfprintf(struct Shf *shf, const char *fmt, va_list args)
  {
    char c, *s;
    int tmp ← 0;
    int field, precision;
    int len;
    int flags;
    unsigned long long llnum;
    char numbuf[(BITS(long long) + 2)/3 + 1];  /* "%#o" produces the longest output */
    int nwritten ← 0;  /* for dealing with the buffer */
    if (¬fmt) return 0;
    while ((c ← *fmt++)) {
      if (c ≠ '%') {
        shf_putc(c, shf);
        nwritten++;
        continue;
      }
      ⟨ Scan the format string for flags 290 ⟩
      if (¬c) break;  /* nasty format */
      if (c ≥ 'A' ∧ c ≤ 'Z') {
        flags |= FL_UPPER;
        c ← c − 'A' + 'a';
      }
      switch (c) {⟨ Read the next variable into a string 291 ⟩}
      ⟨ Append the formatted string 293 ⟩
    }
    return shf_error(shf) ? EOF : nwritten;
  }
```

290. Consider each character in turn (after %) until a non-flag, non-digit character is encountered.

This will accept flags/fields in any order—not just the order specified in *printf(3)*, but this is the way "_doprnt() seems to work (on bsd and sysV)". The only restriction is that the format character must come last.

```
⟨ Scan the format string for flags 290 ⟩ ≡
  flags ← field ← precision ← 0;
  for ( ; (c ← *fmt++); ) {
    switch (c) {
    case '#': flags |= FL_HASH; continue;
    case '+': flags |= FL_PLUS; continue;
    case '-': flags |= FL_RIGHT; continue;
    case '␣': flags |= FL_BLANK; continue;
    case 'h': flags |= FL_SHORT; continue;
    case '0': if (¬(flags & FL_DOT)) flags |= FL_ZERO; continue;
    case '.': flags |= FL_DOT;
      precision ← 0;
      continue;
    case '*': tmp ← va_arg(args, int);
      if (flags & FL_DOT) precision ← tmp;
      else if ((field ← tmp) < 0) {
        field ← −field;
        flags |= FL_RIGHT;
      }
      continue;
    case 'l':
      if (*fmt ≡ 'l') {
        fmt++;
        flags |= FL_LLONG;
      }
      else flags |= FL_LONG;
      continue;
    }
    if (digit(c)) {
      tmp ← c − '0';
      while (c ← *fmt++, digit(c)) tmp ← tmp * 10 + c − '0';
      −−fmt;
      if (tmp < 0)      /* overflow? */
        tmp ← 0;
      if (flags & FL_DOT) precision ← tmp;
      else field ← tmp;
      continue;
    }
    break;
  }
  if (precision < 0) precision ← 0;
```

This code is used in section 289.

291. The character found after the flags and field width determine what type of argument to expect next. It sets *len* and *precision* to the length of the formatted string in *numbuf* and, if a number, its precision.

```
⟨ Read the next variable into a string 291 ⟩ ≡
case 'p':      /* pointer */
  flags &= ~(FL_LLONG | FL_SHORT);
  flags |= FL_LONG;      /* FALLTHROUGH */
case 'd': case 'i': case 'o': case 'u': case 'x':
  flags |= FL_NUMBER;
  s ← &numbuf[sizeof (numbuf)];
  if (flags & FL_LLONG) llnum ← va_arg(args, unsigned long long);
  else if (flags & FL_LONG) {
    if (c ≡ 'd' ∨ c ≡ 'i') llnum ← va_arg(args, long);
    else llnum ← va_arg(args, unsigned long);
  }
  else {
    if (c ≡ 'd' ∨ c ≡ 'i') llnum ← va_arg(args, int);
    else llnum ← va_arg(args, unsigned int);
  }
  ⟨ Stringify a number 292 ⟩
  len ← &numbuf[sizeof (numbuf)] − s;
  if (flags & FL_DOT) {
    if (precision > len) {
      field ← precision;
      flags |= FL_ZERO;
    }
    else precision ← len;      /* no loss */
  }
  break;
case 's':
  if (¬(s ← va_arg(args, char *))) s ← "(null␣%s)";
  len ← strlen(s);
  break;
case 'c':
  flags &= ~FL_DOT;
  numbuf[0] ← va_arg(args, int);
  s ← numbuf;
  len ← 1;
  break;
case '%':
default:
  numbuf[0] ← c;
  s ← numbuf;
  len ← 1;
  break;
```

This code is used in section 289.

292. Prepend a digit (et al.) at a time reducing *llnum* until it's 0.

```
⟨ Stringify a number 292 ⟩ ≡
  switch (c) {
  case 'd':
  case 'i':
    if (0 > (long long) llnum) llnum ← −(long long) llnum, tmp ← 1;
    else tmp ← 0;
        /* FALLTHROUGH */
  case 'u': do {
      *−−s ← llnum % 10 + '0';
      llnum /= 10;
    } while (llnum);
    if (c ≠ 'u') {
      if (tmp) *−−s ← '-';
      else if (flags & FL_PLUS) *−−s ← '+';
      else if (flags & FL_BLANK) *−−s ← '␣';
    }
    break;
  case 'o': do {
      *−−s ← (llnum & 0x7) + '0';
      llnum ≫= 3;
    } while (llnum);
    if ((flags & FL_HASH) ∧ *s ≠ '0') *−−s ← '0';
    break;
  case 'p':
  case 'x':
    {
      const char *digits ← (flags & FL_UPPER) ? "0123456789ABCDEF" : "0123456789abcdef";
      do {
        *−−s ← digits[llnum & 0xf];
        llnum ≫= 4;
      } while (llnum);
      if (flags & FL_HASH) {
        *−−s ← (flags & FL_UPPER) ? 'X' : 'x';
        *−−s ← '0';
      }
    }
  }
```

This code is used in section 291.

293. At this point *s* should point to a string that is to be formatted, and *len* should be the length of the string. This copies it into *shf* with the appropriate padding.

```
⟨ Append the formatted string 293 ⟩ ≡
  if (¬(flags & FL_DOT) ∨ len < precision) precision ← len;
  if (field > precision) {      /* TODO: This and the next two ifs would be clearer if inverted */
    field −= precision;
    if (¬(flags & FL_RIGHT)) {
      field ← −field;
      if ((flags & FL_ZERO) ∧ (flags & FL_NUMBER)) {      /* skip past sign or "0x" when padding with 0 */
        if (*s ≡ '+' ∨ *s ≡ '-' ∨ *s ≡ '␣') {
          shf_putc(*s, shf);
          s++;
          precision−−;
          nwritten++;
        }
        else if (*s ≡ '0') {
          shf_putc(*s, shf);
          s++;
          nwritten++;
          if (−−precision > 0 ∧ (*s | 0x20) ≡ 'x') {
            shf_putc(*s, shf);
            s++;
            precision−−;
            nwritten++;
          }
        }
        c ← '0';
      }
      else c ← flags & FL_ZERO ? '0' : '␣';
      if (field < 0) {
        nwritten += −field;
        for ( ; field < 0; field++) shf_putc(c, shf);
      }
    }
    else c ← '␣';
  }
  else field ← 0;
  if (precision > 0) {
    nwritten += precision;
    for ( ; precision−− > 0; s++) shf_putc(*s, shf);
  }
  if (field > 0) {
    nwritten += field;
    for ( ; field > 0; −−field) shf_putc(c, shf);
  }
```

This code is used in section 289.

294. These interfaces to *shf_vfprintf* mimic *stdlib*.

```
⟨ shf.c 239 ⟩ +≡
  int shf_fprintf(struct Shf *shf, const char *fmt, ...)
  {
    va_list args;
    int n;
    va_start(args, fmt);
    n ← shf_vfprintf(shf, fmt, args);
    va_end(args);
    return n;
  }
```

295.

```
⟨ shf.c 239 ⟩ +≡
  int shf_snprintf(char *buf, int bsize, const char *fmt, ...)
  {
    struct Shf shf;
    va_list args;
    int n;
    if (¬buf ∨ bsize ≤ 0)
      internal_errorf("%s: buf %lx, bsize %d", __func__, (long) buf, bsize);
    shf_sopen(buf, bsize, SHF_WR, &shf);
    va_start(args, fmt);
    n ← shf_vfprintf(&shf, fmt, args);
    va_end(args);
    shf_sclose(&shf);       /* '\0' terminates */
    return n;
  }
```

296.

```
⟨ shf.c 239 ⟩ +≡
  char *shf_smprintf(const char *fmt, ...)
  {
    struct Shf shf;
    va_list args;
    shf_sopen(Λ, 0, SHF_WR | SHF_DYNAMIC, &shf);
    va_start(args, fmt);
    shf_vfprintf(&shf, fmt, args);
    va_end(args);
    return shf_sclose(&shf);       /* '\0' terminates */
  }
```

297. These routines are then used to implement ksh's own I/O—*printf* to *shl_out* (standard error) with flush.

```
⟨ io.c 240 ⟩ +≡
  void shellf(const char *fmt, ...)
  {
    va_list va;
    if (¬initio_done) return;      /* shl_out may not be set up yet... */
    va_start(va, fmt);
    shf_vfprintf(shl_out, fmt, va);
    va_end(va);
    shf_flush(shl_out);
  }
```

298. *printf* to *shl_stdout* (standard output) without flush.

```
⟨ io.c 240 ⟩ +≡
  void shprintf(const char *fmt, ...)
  {
    va_list va;
    if (¬shl_stdout_ok) internal_errorf("shl_stdout␣not␣valid");
    va_start(va, fmt);
    shf_vfprintf(shl_stdout, fmt, va);
    va_end(va);
  }
```

299. Error & Warning Messages. A shell error occurred (eg. syntax error, etc.).

```
⟨ io.c 240 ⟩ +≡
  void errorf(const char *fmt, ...)
  {
    va_list va;
    shl_stdout_ok ← 0;      /* debugging: note that stdout not valid */
    exstat ← 1;
    if (fmt ≠ Λ ∧ *fmt ≠ '\0') {
      error_prefix(true);
      va_start(va, fmt);
      shf_vfprintf(shl_out, fmt, va);
      va_end(va);
      shf_putchar('\n', shl_out);
    }
    shf_flush(shl_out);
    unwind (LERROR);
  }
```

300. Like *errorf* but **unwind** is not called.

```
⟨ io.c 240 ⟩ +≡
  void warningf(bool show_lineno, const char *fmt, ...)
  {
    va_list va;
    error_prefix(show_lineno);
    va_start(va, fmt);
    shf_vfprintf(shl_out, fmt, va);
    va_end(va);
    shf_putchar('\n', shl_out);
    shf_flush(shl_out);
  }
```

301. Used by built-in utilities to prefix the shell and utility name to a message (also **unwind**s environments for special built-ins so that non-interactive shells will exit).

```
⟨ io.c 240 ⟩ +≡
  void bi_errorf(const char *fmt, ...)
  {
    va_list va;
    shl_stdout_ok ← 0;      /* debugging: note that stdout not valid */
    exstat ← 1;
    if (fmt ≠ Λ ∧ *fmt ≠ '\0') {
      error_prefix(true);
      if (builtin_argv0)      /* not set when main calls parse_args */
        shf_fprintf(shl_out, "%s: ", builtin_argv0);
      va_start(va, fmt);
      shf_vfprintf(shl_out, fmt, va);
      va_end(va);
      shf_putchar('\n', shl_out);
    }
    shf_flush(shl_out);
    if ((builtin_flag & SPEC_BI) ∨ (Flag(FPOSIX) ∧ (builtin_flag & KEEPASN))) {
        /* XXX odd use of KEEPASN; also may not want LERROR here */
      builtin_argv0 ← Λ;
      unwind (LERROR);
    }
  }
```

302.
```
⟨ io.c 240 ⟩ +≡
  static void internal_error_vwarn(const char *fmt, va_list va)
  {
    error_prefix(true);
    shf_fprintf(shl_out, "internal error: ");
    shf_vfprintf(shl_out, fmt, va);
    shf_putchar('\n', shl_out);
    shf_flush(shl_out);
  }
```

303. Warn when something that shouldn't happen does.

```
⟨ io.c 240 ⟩ +≡
  void internal_warningf(const char *fmt, ...)
  {
    va_list va;
    va_start(va, fmt);
    internal_error_vwarn(fmt, va);
    va_end(va);
  }
```

304. Warn and **unwind** when something that shouldn't happen does.

```
⟨ io.c 240 ⟩ +≡
  __dead void internal_errorf(const char *fmt, ...)
  {
    va_list va;
    va_start(va, fmt);
    internal_error_vwarn(fmt, va);
    va_end(va);
    unwind (LERROR);
  }
```

305. Used by reporting functions to print "ksh: .kshrc[n]: " */

```
⟨ io.c 240 ⟩ +≡
  void error_prefix(int fileline)
  {
    if (¬fileline ∨ ¬source ∨ ¬source→file ∨
          strcmp(source→file, kshname) ≠ 0)      /* Avoid "foo: foo[2]: ..." */
      shf_fprintf(shl_out, "%s: ", kshname + (*kshname ≡ '-'));
    if (fileline ∧ source ∧ source→file ≠ Λ) {
      shf_fprintf(shl_out, "%s[%d]: ", source→file, source→errline > 0 ? source→errline : source→line);
      source→errline ← 0;
    }
  }
```

306. Display Routines. Print a variable/alias value using single quotes if necessary (POSIX says they should be suitable for re-entry). No trailing newline is printed.

```
⟨ misc.c 9 ⟩ +≡
  void print_value_quoted(const char *s)
  {
    const char *p;
    int inquote ← 0;
    ⟨ Print as-is if quotation is unnecessary 307 ⟩
    for (p ← s; *p; p++) {
      if (*p ≡ '\'') {
        shprintf(inquote ? "'\\'" : "\\'");
        inquote ← 0;
      }
      else {
        if (¬inquote) {
          shprintf("'");
          inquote ← 1;
        }
        shf_putc(*p, shl_stdout);
      }
    }
    if (inquote) shprintf("'");
  }
```

307. If no characters needing quotes are found then there is no need to treat the string specially.

```
⟨ Print as-is if quotation is unnecessary 307 ⟩ ≡
  for (p ← s; *p; p++)
    if (ctype(*p, C_QUOTE)) break;
  if (¬*p) {
    shprintf("%s", s);
    return;
  }
```

This code is used in section 306.

308. Print things in columns and rows—*func* is called to format the ith element. *max_width* +1 is allocated for each column so that there is at least one space between them although no spaces are printed after the last column to avoid problems with terminals that have auto-wrap.

```
⟨ misc.c 9 ⟩ +≡
  void print_columns(struct Shf *shf, int n, char *(*func)(void *, int, char *, int),
          void *arg, int max_width, int prefcol)
  {
    char *str ← alloc(max_width + 1, ATEMP);
    int i;
    int r, c;
    int rows, cols;
    int nspace;
    int col_width;
    ⟨ Figure out how many rows & columns 309 ⟩
    ⟨ Figure out the column width 310 ⟩
    for (r ← 0; r < rows; r++) {
      for (c ← 0; c < cols; c++) {
        i ← c * rows + r;
        if (i < n) {
          shf_fprintf(shf, "%-*s", col_width, (*func)(arg, i, str, max_width + 1));
          if (c + 1 < cols) shf_fprintf(shf, "%*s", nspace, "");
        }
      }
      shf_putchar('\n', shf);
    }
    afree(str, ATEMP);
  }
```

309. ⟨ Figure out how many rows & columns 309 ⟩ ≡

```
  cols ← x_cols/(max_width + 1);
  if (¬cols) cols ← 1;
  rows ← (n + cols − 1)/cols;
  if (prefcol ∧ n ∧ cols > rows) {
    int tmp ← rows;
    rows ← cols;
    cols ← tmp;
    if (rows > n) rows ← n;
  }
```

This code is used in section 308.

310. ⟨ Figure out the column width 310 ⟩ ≡

```
  col_width ← max_width;
  if (cols ≡ 1) col_width ← 0;      /* Don't pad entries in single column output. */
  nspace ← (x_cols − max_width * cols)/cols;
  if (nspace ≤ 0) nspace ← 1;
```

This code is used in section 308.

311. Print a **timespec** or **timeval** object.

⟨ Bourne commands 311 ⟩ ≡

```
static void p_ts(struct Shf *shf, int posix, struct timespec *ts, int width, char *prefix, char
        *suffix)
{
  if (posix) shf_fprintf(shf, "%s%*lld.%02ld%s", prefix ? prefix : "", width, (long long)
        ts→tv_sec, ts→tv_nsec/10000000, suffix);
  else shf_fprintf(shf, "%s%*lldm%02lld.%02lds%s", prefix ? prefix : "", width, (long long)
        ts→tv_sec/60, (long long) ts→tv_sec % 60, ts→tv_nsec/10000000, suffix);
}
```

See also sections 312, 467, 468, 1205, 1206, 1207, 1212, 1213, 1214, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, and 1229.

This code is used in section 1200.

312. ⟨ Bourne commands 311 ⟩ +≡

```
static void p_tv(struct Shf *shf, int posix, struct timeval *tv, int width, char *prefix, char *suffix)
{
  if (posix) shf_fprintf(shf, "%s%*lld.%02ld%s", prefix ? prefix : "", width, (long long)
        tv→tv_sec, tv→tv_usec/10000, suffix);
  else shf_fprintf(shf, "%s%*lldm%02lld.%02lds%s", prefix ? prefix : "", width, (long long)
        tv→tv_sec/60, (long long) tv→tv_sec % 60, tv→tv_usec/10000, suffix);
}
```

313. Source Input. Code is read in by creating a **Source** object and pointing it to either a file or a C-string buffer or array of buffers. *yylex* examines *type* to see which sort of source it has and scans the appropriate location to break the string up into tokens. A token is not defined by a regular expression which would break the string "`$(size $(whence ksh))`" into 7 tokens representing the individual lexemes "`$(`", "`size`", "`$(`", "`whence`", "`ksh`" and two "`)`"s, instead it is treated as a single token and a stack of partially-complete **Source** objects in *next* keeps track of the state of analysis.

```
#define HERES 10    /* max <<,<<- in one line */
#define IDENT 64
#define STATE_BSIZE 32
```

⟨ lex.h 313 ⟩ ≡

```
int yylex(int);
void yyerror(const char *, ...)__attribute__((__noreturn__,__format__(printf,1,2)));
Source *pushs(int,Area *);
void set_prompt(int);
void pprompt(const char *,int);
```

314. ⟨ lex.c 314 ⟩ ≡

```
#include <ctype.h>
#include <errno.h>
#include <libgen.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
static void readhere(struct ioword *);
static int getsc__(void);
static void getsc_line(Source *);
static int getsc_bn(void);
static char *get_brace_var(XString *,char *);
static int arraysub(char **);
static const char *ungetsc(int);
static void gethere(void);
static Lex_state *push_state_(State_info *,Lex_state *);
static Lex_state *pop_state_(State_info *,Lex_state *);
static char *special_prompt_expand(char *);
static int doprompt(const char *,int,const char **,int);
int promptlen(const char *cp,const char **spp);
static int backslash_skip;
static int ignore_backslash_newline;
Source *source;     /* yyparse/yylex source */
YYSTYPE yylval;     /* result from yylex */
struct ioword *heres[HERES], **herep;
char ident[IDENT+1];
char **history;     /* saved commands */
char **histptr;     /* last history item */
uint32_t histsize;  /* history size */
```

See also sections 319, 320, 321, 325, 333, 334, 335, 338, 368, 372, 373, 374, 375, 377, 378, 379, and 380.

315. ⟨ Externally-linked variables 6 ⟩ +≡
```
extern Source *source;
extern YYSTYPE yylval;
extern struct ioword *heres[HERES], **herep;
extern char ident[IDENT + 1];
extern char **history;
extern char **histptr;
extern uint32_t histsize;
```

316. ⟨ Type definitions 17 ⟩ +≡
```
typedef struct Source Source;
struct Source {      /* renamed for cosmetic reasons */
  const char *str;      /* input pointer */
  int type;      /* input type */
  const char *start;      /* start of current buffer */
  union {
    char **strv;      /* string [] */
    struct Shf *shf;      /* shell file */
    struct tbl *tblp;      /* alias (SALIAS) */
    char *freeme;      /* also for SREREAD */
  } u;
  char ugbuf[2];      /* buffer for ungetsc() (SREREAD) and alias (SALIAS) */
  int line;      /* line number */
  int cmd_offset;      /* line number - command number */
  int errline;      /* line the error occurred on (0 if not set) */
  const char *file;      /* input file name */
  int flags;      /* SF_* */
  Area *areap;
  XString xs;      /* input buffer */
  Source *next;      /* stacked source */
};
```

317. **Source**→*type* values.
```
#define SEOF 0      /* input EOF */
#define SFILE 1      /* file input */
#define SSTDIN 2      /* read stdin */
#define SSTRING 3      /* string */
#define SWSTR 4      /* string without \n */
#define SWORDS 5      /* string[] */
#define SWORDSEP 6      /* string[] separator */
#define SALIAS 7      /* alias expansion */
#define SREREAD 8      /* read ahead to be re-scanned */
```

318. **Source**→*flags* values.
```
#define SF_ECHO BIT(0)      /* echo input to shlout */
#define SF_ALIAS BIT(1)      /* faking space at end of alias */
#define SF_ALIASEND BIT(2)      /* faking space at end of alias */
#define SF_TTY BIT(3)      /* type ≡ SSTDIN & it is a tty */
```

319. The type of lexical analyser implemented here is known as a look-ahead parser because it occasionally needs to know one or two characters more from the input stream to determine the purpose of the character under test, eg. "<" may be redirecting input or the first character of the tokens "<<" or "<<-".

If it turns out the characters looked ahead at are not wanted they're put back into the input so that the next loop will see them as normal. Unfortunately the input may be a read-only stream so a small buffer is maintained in **Source** to hold them.

```
⟨ lex.c 314 ⟩ +≡
  static const char *ungetsc(int c)
  {
    if (backslash_skip) backslash_skip--;
    if (source→str ≡ null ∧ c ≡ '\0') return source→str;      /* Don't unget eof... */
    if (source→str > source→start) source→str--;
    else {
      Source *s;
      s ← pushs(SREREAD, source→areap);
      s→ugbuf[0] ← c;
      s→ugbuf[1] ← '\0';
      s→start ← s→str ← s→ugbuf;
      s→next ← source;
      source ← s;
    }
    return source→str;
  }
```

320. In order to scan a string there needs to be a string. When the analyser requests the next character to test from *getsc*, if none is available it will fall back to *getsc__* in order to find something to put in the buffer. When reading from a file or standard input (ie. also when interactive) this will use *getsc_line* to read a whole line of text into the buffer before returning its first character.

In between *getsc* and *getsc__* is a handful of macros and *getsc_bn*, which handles reading a line which may contain a backslash followed immediate by a newline character unless such a construct is to be "ignored" according to *ignore_backslash_newline* in which case it is returned instead[1].

```
#define getsc() (*source->str ≠ '\0' ∧ *source->str ≠ '\\' ∧ ¬backslash_skip
    ? *source->str++ : getsc_bn())
```

⟨ lex.c 314 ⟩ +≡

```
static int getsc_bn(void)
{
  int c, c2;
  if (ignore_backslash_newline) return getsc_();
  if (backslash_skip ≡ 1) {
    backslash_skip ← 2;
    return getsc_();
  }
  backslash_skip ← 0;
  while (1) {
    c ← getsc_();
    if (c ≡ '\\') {
      if ((c2 ← getsc_()) ≡ '\n') continue;      /* ignore the "\nl"—get the next char... */
      ungetsc(c2);
      backslash_skip ← 1;
    }
    return c;
  }
}
```

[1] Ignored in this sense means that although ordinarily a backslash is an instruction to the lexer to interpret specially the following (newline) character, this time the backslash' instruction should be ignored and the characters returned as-is.

321. The work of finding and returning the actual character is carried out by *getsc__* depending on the type of source that's being read (*s→type*). The source of the source is queried in its manner which puts a pointer to a line of text (or Λ) in *s→str*.

```
#define getsc_() ((*source→str ≠ '\0') ? *source→str++ : getsc__())
```

⟨ lex.c 314 ⟩ +≡

```
static int getsc__(void)
{
  Source *s ← source;
  int c;
  while ((c ← *s→str++) ≡ 0) {
    s→str ← Λ;      /* return 0 for EOF by default */
    switch (s→type) {
      ⟨ Common Source sources 322 ⟩
      ⟨ Reading an alias 324 ⟩
      ⟨ Re-reading source code 323 ⟩
    }
    if (s→str ≡ Λ) {
      s→type ← SEOF;
      s→start ← s→str ← null;
      return '\0';
    }
    if (s→flags & SF_ECHO) {
      shf_puts(s→str, shl_out);
      shf_flush(shl_out);
    }
  }
  return c;
}
```

322. In-memory strings (SSTRING) and arrays of strings (SWSTR) are already in place. If memory serves SWORDS and SWORDSEP are for arrays of words in C-strings.

When reading from standard input or a file a (nother) line needs to be read in.

```
⟨ Common Source sources 322 ⟩ ≡
case SEOF: s→str ← null;
  return 0;
case SSTDIN: case SFILE: getsc_line(s);
  break;
case SWSTR: break;
case SSTRING: break;
case SWORDS: s→start ← s→str ← *s→u.strv ++;
  s→type ← SWORDSEP;
  break;
case SWORDSEP:
  if (*s→u.strv ≡ Λ) {
    s→start ← s→str ← "\n";
    s→type ← SEOF;
  }
  else {
    s→start ← s→str ← "␣";
    s→type ← SWORDS;
  }
  break;
```

This code is used in section 321.

323. ⟨ Re-reading source code 323 ⟩ ≡

```
case SREREAD:
  if (s→start ≠ s→ugbuf)      /* yuck */
    afree(s→u.freeme, ATEMP);
  source ← s ← s→next;
  continue;
```

This code is used in section 321.

324. I still cannot be bothered to figure whatever weird magic aliases do.

/* At this point, we need to keep the current * alias in the source list so recursive * aliases can be detected and we also need * to return the next character. Do this * by temporarily popping the alias to get * the next character and then put it back * in the source list with the SF_ALIASEND * flag set. */

⟨ Reading an alias 324 ⟩ ≡

```
case SALIAS:
  if (s→flags & SF_ALIASEND) {      /* pass on an unused SF_ALIAS flag */
    source ← s→next;
    source→flags |= s→flags & SF_ALIAS;
    s ← source;
  }
  else if (*s→u.tblp→val.s ∧ isspace((unsigned char) strchr(s→u.tblp→val.s, 0)[−1])) {
      /* this alias ended with "␣", enabling alias expansion on the following word. */
    source ← s ← s→next;      /* pop source stack */
    s→flags |= SF_ALIAS;
  }
  else {
    source ← s→next;      /* pop source stack */
    source→flags |= s→flags & SF_ALIAS;
    c ← getsc__( );
    if (c) {
      s→flags |= SF_ALIASEND;
      s→ugbuf [0] ← c;
      s→ugbuf [1] ← '\0';
      s→start ← s→str ← s→ugbuf;
      s→next ← source;
      source ← s;
    }
    else {      /* avoid reading eof twice */
      s ← source;
      s→str ← Λ;
      break;
    }
  }
  continue;
```

This code is used in section 321.

325. First and before possibly waiting for the user the destination string is prepared in case it gets returned early.

```
⟨ lex.c 314 ⟩ +≡
  static void getsc_line(Source *s)
  {
    char *xp ← Xstring(s→xs, xp);
    int interactive ← Flag(FTALKING) ∧ s→type ≡ SSTDIN;
    int have_tty ← interactive ∧ (s→flags & SF_TTY);
    XcheckN(s→xs, xp, LINE);
    *xp ← '\0';
    s→start ← s→str ← xp;
    if (have_tty ∧ ksh_tmout) {      /* Set an alarm for $TMOUT */
      ksh_tmout_state ← TMOUT_READING;
      alarm(ksh_tmout);
    }
    if (have_tty ∧ (0
#ifdef VI
      ∨ Flag(FVI)
#endif      /* VI */
#ifdef EMACS
      ∨ Flag(FEMACS) ∨ Flag(FGMACS)
#endif      /* EMACS */
    )) {⟨ Read a line interactively using the editor 326 ⟩}
    else {⟨ Read a line simply 327 ⟩}
    source ← s;      /* XXX: temporary kludge to restore source after a trap may have been executed. */
    if (have_tty ∧ ksh_tmout) {
      ksh_tmout_state ← TMOUT_EXECUTING;
      alarm(0);
    }
    ⟨ Process a complete line 328 ⟩
    if (interactive) set_prompt(PS2);
  }
```

326. ⟨ Read a line interactively using the editor 326 ⟩ ≡

```
  int nread;
  nread ← x_read(xp, LINE);
  if (nread < 0) nread ← 0;      /* read error */
  xp[nread] ← '\0';
  xp += nread;
```

This code is used in section 325.

327. Flush any unwanted input so other programs/built-ins can read it. Not very optimal, but less error prone than flushing elsewhere, dealing with redirections, etc.

```
⟨ Read a line simply 327 ⟩ ≡
  if (interactive) pprompt(prompt, 0);
  else s→line++;
  while (1) {
    char *p ← shf_getse(xp, Xnleft(s→xs, xp), s→u.shf);
    if (¬p ∧ shf_error(s→u.shf) ∧ s→u.shf→errno_ ≡ EINTR) {
      shf_clearerr(s→u.shf);
      if (trap) runtraps(0);
      continue;
    }
    if (¬p ∨ (xp ← p, xp[−1] ≡ '\n')) break;
    xp++;      /* move past '\0' so doubling works... */
    XcheckN(s→xs, xp, Xlength(s→xs, xp));      /* double buffer size */
    xp−−;      /* ...and move back again */
  }
  if (s→type ≡ SSTDIN) shf_flush(s→u.shf);      /* TODO: reduce size of shf buffer (~128?) if SSTDIN */
```

This code is used in section 325.

328. Save the line in the **Source** object s and if interactive and the line consists of other than whitespace (or whatever `$IFS` directs) it's saved to the history file. If reading from a file and it has ended, close it.

It's not clear where the `'\0'`s might be coming from that *strip_nuls* removes.

```
⟨ Process a complete line 328 ⟩ ≡
  s→start ← s→str ← Xstring(s→xs, xp);
  strip_nuls(Xstring(s→xs, xp), Xlength(s→xs, xp));      /* Note: if input is all '\0's, this is not eof */
  if (Xlength(s→xs, xp) ≡ 0) {      /* EOF */
    if (s→type ≡ SFILE) shf_fdclose(s→u.shf);
    s→str ← Λ;
  }
  else if (interactive) {
    char *p ← Xstring(s→xs, xp);
    if (cur_prompt ≡ PS1)
      while (*p ∧ ctype(*p, C_IFS) ∧ ctype(*p, C_IFSWS)) p++;
    if (*p) {
      s→line++;
      histsave(s→line, s→str, 1);
    }
  }
```

This code is used in section 325.

329. Strip any '\0' bytes from buf and return the new length. This is also used by the history log.

```
⟨ misc.c 9 ⟩ +≡
  int strip_nuls(char *buf, int nbytes)
  {
    char *dst;
    if ((dst ← memchr(buf, '\0', nbytes))) {
      char *end ← buf + nbytes;
      char *p, *q;
      for (p ← dst; p < end; p ← q) {
        while (++p < end ∧ *p ≡ '\0')      /* skip a block of '\0's */ ;
        if (¬(q ← memchr(p, '\0', end − p))) q ← end;      /* find end of non-'\0' block */
        memmove(dst, p, q − p);
        dst += q − p;
      }
      *dst ← '\0';
      return dst − buf;
    }
    return nbytes;
  }
```

330. Lexical Analysis. The lexical analyser reads each character in the input stream one at a time and processes it depending on the analyser's current state, which can be one of the following:

```
#define SINVALID -1     /* invalid state */
#define SBASE 0     /* outside any lexical constructs */
#define SWORD 1     /* implicit quoting for substitute */
#define SLETPAREN 2     /* inside (( ... )), implicit quoting */
#define SSQUOTE 3     /* inside '...' */
#define SDQUOTE 4     /* inside "..." */
#define SBRACE 5     /* inside ${var} */
#define SCSPAREN 6     /* inside $( ... ) */
#define SBQUOTE 7     /* inside ` ... ` */
#define SASPAREN 8     /* inside $(( ... )) */
#define SHEREDELIM 9     /* parsing <</<<- delimiter */
#define SHEREDQUOTE 10     /* parsing " in <</<<- delimiter */
#define SPATTERN 11     /* parsing Ø(...|...) pattern where Ø is one of *+?@! */
#define STBRACE 12     /* parsing ${var[#%]...} */
#define SBRACEQ 13     /* inside "${var}" */
```

331. The name of this object, which is the result of *yylex*, and the other names beginning with "yy" suggest that ksh began its life with tokens represented by regular expressions read by a parser generated by `yacc`. It is the type of the global value *yylval* which stores extra details of the token returned from *yylex*.

```
⟨ Type definitions 17 ⟩ +≡
  typedef union {
    int i;
    char *cp;
    char **wp;
    struct Op *o;
    struct ioword *iop;
  } YYSTYPE;
```

332. The tokens found are not individual lexemes represented or detected by a regular expression but complete shell expressions like "`${var:-${PWD}}`", and "`$(size␣$(whence␣ksh))`". These expressions may themselves contain complete expressions so the analyser is a state machine to keep track of partial work.

For example scanning the expression "`"$(bar)"`" would first encounter `"` and enter a state where it expects to read a double-quoted string (`SDQUOTE`). The next character is `$` indicating `$`-expansion and a peek is taken at the character after *that* to determine which type of `$`-expansion. In this case `(` indicates a command substitution so the `SDQUOTE` state is pushed onto a stack and the analyser enters the `SCSPAREN` state.

The "`b`", "`a`" and "`r`" are read while in this state and when the "`)`" is finally encountered that state is finished with so the stack is popped and the analyser returns to the `SDQUOTE` state. The next time around it sees the second "`"`" and completes the double-quoted-string token it was looking for.

```
#define ls_scsparen ls_info.u_scsparen
#define ls_sasparen ls_info.u_sasparen
#define ls_sletparen ls_info.u_sletparen
#define ls_sbquote ls_info.u_sbquote
```

⟨ Type definitions 17 ⟩ +≡

```
typedef struct lex_state Lex_state;
struct lex_state {
  int ls_state;
  union {
    struct scsparen_info {      /* $( ... ) */
      int nparen;       /* count open parenthesis */
      int csstate;      /* XXX remove */
    } u_scsparen;
    struct sasparen_info {      /* $(( ... )) */
      int nparen;       /* count open parenthesis */
      int start;      /* marks start of $(( in output str */
    } u_sasparen;
    struct sletparen_info {      /* (( ... )) */
      int nparen;       /* count open parenthesis */
    } u_sletparen;
    struct sbquote_info {      /* ` ... ` */
      int indquotes;      /* true if in double quotes: "` ... `" */
    } u_sbquote;
    Lex_state *base;      /* used to point to next state block */
  } ls_info;
};
typedef struct State_info State_info;
struct State_info {
  Lex_state *base;
  Lex_state *end;
};
```

333. `#define PUSH_STATE(s) do {`
`if (++statep ≡ state_info.end) statep ← push_state_(&state_info, statep);`
`state ← statep→ls_state ← (s);`
`} while (0)`

⟨ lex.c 314 ⟩ +≡
```
static Lex_state *push_state_(State_info *si, Lex_state *old_end)
{
  Lex_state *new ← areallocarray(Λ, STATE_BSIZE, sizeof(Lex_state), ATEMP);
  new[0].ls_info.base ← old_end;
  si→base ← &new[0];
  si→end ← &new[STATE_BSIZE];
  return &new[1];
}
```

334. `#define POP_STATE() do {`
`if (--statep ≡ state_info.base) statep ← pop_state_(&state_info, statep);`
`state ← statep→ls_state;`
`} while (0)`

⟨ lex.c 314 ⟩ +≡
```
static Lex_state *pop_state_(State_info *si, Lex_state *old_end)
{
  Lex_state *old_base ← si→base;
  si→base ← old_end→ls_info.base − STATE_BSIZE;
  si→end ← old_end→ls_info.base;
  afree(old_base, ATEMP);
  return si→base + STATE_BSIZE − 1;
}
```

335. Errors encountered while scanning are reported using this function, included here so that the start of lexing proper gets push on to the next page. There is little to be said about it (except note that it's popping the *sources* stack).

⟨ lex.c 314 ⟩ +≡
```
void yyerror(const char *fmt, ...)
{
  va_list va;
  while (source→type ≡ SALIAS ∨ source→type ≡ SREREAD)      /* pop aliases and re-reads */
    source ← source→next;
  source→str ← null;      /* zap pending input */
  error_prefix(true);
  va_start(va, fmt);
  shf_vfprintf(shl_out, fmt, va);
  va_end(va);
  errorf(Λ);
}
```

336. Calling *yylex* consumes characters from the input until a complete token is formed. A token is one of these identifiers possibly followed by a character or a sequence of tokens and a terminator.

```
#define EOS 0     /* end of string */
#define CHAR 1    /* unquoted character */
#define QCHAR 2   /* quoted character */
#define COMSUB 3  /* $() substitution (zero-terminated) */
#define EXPRSUB 4 /* $(()) substitution (zero-terminated) */
#define OQUOTE 5  /* opening " or ' */
#define CQUOTE 6  /* closing " or ' */
#define OSUBST 7  /* opening ${ subst (followed by { or X) */
#define CSUBST 8  /* closing } of above (followed by } or X) */
#define OPAT 9    /* open pattern: *(, @(, etc. */
#define SPAT 10   /* separate pattern: | */
#define CPAT 11   /* close pattern: ) */
```

337. Flags passed to *yylex* slightly adapt its behaviour to suit the context of the parser.

```
#define CONTIN BIT(0)     /* skip new lines to complete command */
#define ONEWORD BIT(1)    /* single word for substitute */
#define ALIAS BIT(2)      /* recognize alias */
#define KEYWORD BIT(3)    /* recognize keywords */
#define LETEXPR BIT(4)    /* get expression inside (( ... )) */
#define VARASN BIT(5)     /* check for var=word */
#define ARRAYVAR BIT(6)   /* parse x[1␣&␣2] as one word */
#define ESACONLY BIT(7)   /* only accept esac keyword */
#define CMDWORD BIT(8)    /* parsing simple command (alias related) */
#define HEREDELIM BIT(9)  /* parsing <</<<- delimiter */
#define HEREDOC BIT(10)   /* parsing heredoc */
#define UNESCAPE BIT(11)  /* remove backslashes */
```

338. Each of the front-ends *token* & *tpeek* obtain a token and saves it in *symbol*. *token* marks the token as accepted (*reject* ← *false*) so that the next call to token will call *yylex* again. Conversely *tpeek* marks the token as unacceptable (*reject* ← *true*) and the next use of either will return that rather than call *yylex* again.

The main loop test is easier to read if it's inverted and the distinction between the two base states is plastered over[1]: *until*(((*c* ← *getsc*( )) ≡ 0) ∨ (*is_base*(*state*) ∧ *ctype*(*c*, C_LEX1)))[2].

```
#define token(cf) ((reject) ? (reject ← false, symbol) : (symbol ← yylex(cf)))
#define tpeek(cf) ((reject) ? (symbol) : (reject ← true, symbol ← yylex(cf)))
```

⟨ lex.c 314 ⟩ +≡

```
int yylex(int cf)
{
  Lex_state states[STATE_BSIZE], *statep;
  State_info state_info;
  int c, state;
  XString ws;      /* expandable output word */
  char *wp;      /* output word pointer */
  char *sp, *dp;
  int c2;
Again: ⟨ Initialise the lexing state machine 339 ⟩
  while (¬((c ← getsc( )) ≡ 0 ∨
      ((state ≡ SBASE ∨ state ≡ SHEREDELIM) ∧ ctype(c, C_LEX1))))
  {⟨ Collect non-special or quoted characters to form word 340 ⟩}
Done: Xcheck(ws, wp);
  if (statep ≠ &states[1]) yyerror("no␣closing␣quote\n");      /* XXX figure out what is missing */
  if (state ≡ SHEREDELIM) state ← SBASE;      /* avoid tests for SHEREDELIM wherever SBASE tested */
  dp ← Xstring(ws, wp);
  if ((c ≡ '<' ∨ c ≡ '>') ∧ state ≡ SBASE ∧
      ((c2 ← Xlength(ws, wp)) ≡ 0 ∨
        (c2 ≡ 2 ∧ dp[0] ≡ CHAR ∧ digit(dp[1])))) {⟨ Prepare redirection token and return 365 ⟩}
  if (wp ≡ dp ∧ state ≡ SBASE) {⟨ Process no token and return or restart 366 ⟩}
  ⟨ Finish the token (may return or restart) 367 ⟩
  return LWORD;
}
```

[1] Such a macro would also make the epilogue (⟨ Finish the token (may **return** or **restart**) 367 ⟩) simpler.
[2] C_LEX1 is any of tab (\t), newline (\n) or [␣|&;<>()].

339. The state machine is (re-)initialised by clearing the stack to a single SINVALID state and preparing the memory in *ws* (& *wp*) in which the token will be built.

The initial state is set to one of SASPAREN, SBASE, SHEREDELIM, SLETPAREN or SWORD depending on the caller's flags (*cf*). In the case of scanning for a normal token whitespace and comments are skipped.

```
⟨ Initialise the lexing state machine 339 ⟩ ≡
  states[0].ls_state ← SINVALID;
  states[0].ls_info.base ← Λ;
  statep ← &states[1];
  state_info.base ← states;
  state_info.end ← &states[STATE_BSIZE];
  Xinit(ws, wp, 64, ATEMP);
  backslash_skip ← 0;
  ignore_backslash_newline ← 0;
  if (cf & ONEWORD) state ← SWORD;      /* ⟨ Lexing SWORD state (single word) 359 ⟩ */
  else if (cf & LETEXPR) {
    *wp++ ← OQUOTE;      /* enclose arguments in (double) quotes */
    state ← SLETPAREN;      /* ⟨ Lexing SLETPAREN state ((( ... ))) 360 ⟩ */
    statep→ls_sletparen.nparen ← 0;
  }
  else {      /* normal lexing */
    state ← (cf & HEREDELIM) ? SHEREDELIM : SBASE;
      /* ⟨ Lexing SHEREDELIM state (<</<<- delimiter) 361 ⟩ */
    while ((c ← getsc()) ≡ '␣' ∨ c ≡ '\t')
        /* skip whitespace */ ;
    if (c ≡ '#') {
      ignore_backslash_newline++;
      while ((c ← getsc()) ≠ '\0' ∧ c ≠ '\n')
          /* skip a comment */ ;
      ignore_backslash_newline--;
    }
    ungetsc(c);
  }
  if (source→flags & SF_ALIAS) {      /* trailing "␣" in alias definition */
    source→flags &= ∼SF_ALIAS;
    if (¬Flag(FPOSIX) ∨ (cf & CMDWORD))
        /* In POSIX mode, a trailing space only counts if we are parsing a simple command */
      cf |= ALIAS;
  }
  statep→ls_state ← state;      /* Initial state—one of */
```

This code is cited in section 359.

This code is used in section 338.

340. Ensure there is space in *ws* then one giant switch depending on the lexing state.

The base state first checks whether it is looking at csh-style history invocation and then whether the partially-scanned token is an array assignment (`foo[n]=...`). Their implementation distracts from the implementation of the lexing state machine so they have been moved to the end. The remainder of the base state is jumped into from the other states.

```
⟨ Collect non-special or quoted characters to form word 340 ⟩ ≡
  Xcheck(ws, wp);
  switch (state) {
  case SBASE:
    if (Flag(FCSHHISTORY) ∧
        (source¬flags & SF_TTY) ∧ c ≡ '!') {⟨ Lexing SBASE handle csh-style history 370 ⟩}
    if (c ≡ '[' ∧ (cf & (VARASN | ARRAYVAR))) {⟨ Detect assignment to an array 371 ⟩}
      /* FALLTHROUGH */
  Sbase1:     /* does include [*+?@!](...|...) pattern */
    ⟨ Detect matching a pattern and switch to SPATTERN 341 ⟩
      /* FALLTHROUGH */
  Sbase2:     /* doesn't include [*+?@!](...|...) pattern */
    ⟨ Unescape \ or detect quotes and switch state 342 ⟩
    break;    /* TODO: misleading; previous ends with switch { ... default: goto Subst; } */
  Subst:
    ⟨ Detect escapes and substitutions 343 ⟩
    break;
  case SSQUOTE: ⟨ Lexing SSQUOTE state ('...') 351 ⟩
  case SDQUOTE: ⟨ Lexing SDQUOTE state ("...") 352 ⟩
  case SCSPAREN: ⟨ Lexing SCSPAREN state ($( ... )) 353 ⟩
  case SASPAREN: ⟨ Lexing SASPAREN state ($(( ... ))) 354 ⟩
  case SBRACEQ: ⟨ Lexing SBRACEQ state ("${⟨var⟩}") 356 ⟩
  case SBRACE: ⟨ Lexing SBRACE state (${⟨var⟩}) 355 ⟩
  case STBRACE: ⟨ Lexing STBRACE state (${var[#%]...}) 357 ⟩
  case SBQUOTE: ⟨ Lexing SBQUOTE state (` ... `) 358 ⟩
  case SWORD: ⟨ Lexing SWORD state (single word) 359 ⟩
  case SLETPAREN: ⟨ Lexing SLETPAREN state ((( ... ))) 360 ⟩
  case SHEREDELIM: ⟨ Lexing SHEREDELIM state (<</<<- delimiter) 361 ⟩
  case SHEREDQUOTE: ⟨ Lexing SHEREDQUOTE state (" in <</<<- delimiter) 362 ⟩
  case SPATTERN: ⟨ Lexing SPATTERN state ([*+?@!](...|...)) 363 ⟩
  }
```

This code is used in section 338.

341. If any of the set [*@+?!] is followed by "(" then append OPAT and the pattern character to the lexing output, remaining in the current state (which may not be SBASE).

```
⟨ Detect matching a pattern and switch to SPATTERN 341 ⟩ ≡
  if (c ≡ '*' ∨ c ≡ '@' ∨ c ≡ '+' ∨ c ≡ '?' ∨ c ≡ '!') {
    c2 ← getsc( );
    if (c2 ≡ '(') {
      *wp++ ← OPAT;
      *wp++ ← c;
      PUSH_STATE(SPATTERN);      /* ⟨ Lexing SPATTERN state ([*+?@!](...|...)) 363 ⟩ */
      break;
    }
    ungetsc(c2);
  }
```

This code is cited in sections 355, 357, and 363.

This code is used in section 340.

342. Two consecutive "\"s append QCHAR and a \ to the lexing output. An unescaped "'" or "" appends OQUOTE and pushes the state SSQUOTE or SDQUOTE respectively to scan the next character (which may be newline), except in a heredoc body or ${ ... }, "\'" appends CHAR and '.

```
⟨ Unescape \ or detect quotes and switch state 342 ⟩ ≡
  switch (c) {
  case '\\':
    c ← getsc( );
    if (c)      /* trailing "\" is lost */
      *wp++ ← QCHAR, *wp++ ← c;
    break;
  case '\'':
    if ((cf & HEREDOC) ∨ state ≡ SBRACEQ) {
      *wp++ ← CHAR, *wp++ ← c;
      break;
    }
    *wp++ ← OQUOTE;
    ignore_backslash_newline++;
    PUSH_STATE(SSQUOTE);      /* ⟨ Lexing SSQUOTE state ('...') 351 ⟩ */
    break;
  case '"':
    *wp++ ← OQUOTE;
    PUSH_STATE(SDQUOTE);      /* ⟨ Lexing SDQUOTE state ("...") 352 ⟩ */
    break;
  default: goto Subst;
      /* This is the same as a FALLTHROUGH to Subst (⟨ Detect escapes and substitutions 343 ⟩) */
  }
```

This code is cited in sections 356 and 360.

This code is used in section 340.

343. If c isn't one of [\$`] then CHAR and the character are appended.

```
⟨ Detect escapes and substitutions 343 ⟩ ≡
  switch (c) {
  case '\\': ⟨ Detect \-escapes 344 ⟩ break;
  case '$': ⟨ Detect $-expansion 345 ⟩ break;
  case '`': ⟨ Detect backtick-expansion 350 ⟩ break;
  default: *wp++ ← CHAR, *wp++ ← c;
  }     /* a break immediately follows */
```

This code is cited in sections 342, 350, 352, and 359.

This code is used in section 340.

344. This block may be jumped into part-way through scanning a character so \-escapes are detected again. TODO: Figure out which states jump here.

```
⟨ Detect \-escapes 344 ⟩ ≡
  c ← getsc( );
  switch (c) {
  case '\\': case '$': case '`':      /* this is a backtick ("`") */
    *wp++ ← QCHAR, *wp++ ← c;
    break;
  case '"':
    if ((cf & HEREDOC) ≡ 0) {
      *wp++ ← QCHAR, *wp++ ← c;
      break;
    }
    else      /* FALLTHROUGH */
  default:
    if (cf & UNESCAPE) {
      *wp++ ← QCHAR, *wp++ ← c;
      break;
    }
    Xcheck(ws, wp);
    if (c) {      /* trailing "\" is lost */
      *wp++ ← CHAR, *wp++ ← '\\';
      *wp++ ← CHAR, *wp++ ← c;
    }
    break;
  }
```

This code is used in section 343.

345. The type of $-expansion depends on the next character[1] or CHAR and $ are appended if no expansion should take place. Scanning for expansion sub-tokens is dealt with after the lexing states.

```
⟨ Detect $-expansion 345 ⟩ ≡
  c ← getsc( );
  if (c ≡ '(') {⟨ Detect $(-expansion 346 ⟩}
  else if (c ≡ '{') {⟨ Detect ${-expansion 347 ⟩}
  else if (ctype(c, C_ALPHA)) {⟨ Detect $variable-expansion 348 ⟩}
  else if (ctype(c, C_VAR1) ∨ digit(c)) {⟨ Detect punctuation and $[0-9]-expansion 349 ⟩}
  else {
    *wp++ ← CHAR, *wp++ ← '$';
    ungetsc(c);
  }
```

This code is used in section 343.

346. Look for a second (and enter SASPAREN if found or SCSPAREN if not.

```
⟨ Detect $(-expansion 346 ⟩ ≡
  c ← getsc( );
  if (c ≡ '(') {
    PUSH_STATE(SASPAREN);      /* ⟨ Lexing SASPAREN state ($(( ... )) ) 354 ⟩ */
    statep¬ls_sasparen.nparen ← 2;
    statep¬ls_sasparen.start ← Xsavepos(ws, wp);
    *wp++ ← EXPRSUB;
  }
  else {
    ungetsc(c);
    PUSH_STATE(SCSPAREN);      /* ⟨ Lexing SCSPAREN state ($( ... )) 353 ⟩ */
    statep¬ls_scsparen.nparen ← 1;
    statep¬ls_scsparen.csstate ← 0;
    *wp++ ← COMSUB;
  }
```

This code is used in section 345.

[1] C_VAR1 is any of [*@#!$-?]

347. `${`-expansion expects a variable name and modifiers. The name is scanned for by *get_brace_var* below and then modification by "`#`" or "`%`" is detected. "`:#`" and "`:%`" are also accepted for compatibility with ksh88.

```
⟨ Detect ${-expansion 347 ⟩ ≡
  *wp++ ← OSUBST;
  *wp++ ← '{';
  wp ← get_brace_var(&ws, wp);
  c ← getsc( );
  if (c ≡ ':') {
    *wp++ ← CHAR, *wp++ ← c;
    c ← getsc( );
  }
  if (c ≡ '#' ∨ c ≡ '%') {
    ungetsc(c);
    PUSH_STATE(STBRACE);      /* ⟨ Lexing STBRACE state (${var[#%]...}) 357 ⟩ */
  }
  else {
    ungetsc(c);
    if (state ≡ SDQUOTE ∨ state ≡ SBRACEQ) PUSH_STATE(SBRACEQ);
        /* ⟨ Lexing SBRACEQ state ("${⟨var⟩}") 356 ⟩ */
    else PUSH_STATE(SBRACE);      /* ⟨ Lexing SBRACE state (${⟨var⟩}) 355 ⟩ */
  }
```

This code is used in section 345.

348. ⟨ Detect $*variable*-expansion 348 ⟩ ≡

```
  *wp++ ← OSUBST;
  *wp++ ← 'X';
  do {
    Xcheck(ws, wp);
    *wp++ ← c;
    c ← getsc( );
  } while (ctype(c, C_ALPHA) ∨ digit(c));
  *wp++ ← '\0';
  *wp++ ← CSUBST;
  *wp++ ← 'X';
  ungetsc(c);
```

This code is cited in section 562.

This code is used in section 345.

349. ⟨ Detect punctuation and $*[0-9]*-expansion 349 ⟩ ≡

```
  Xcheck(ws, wp);
  *wp++ ← OSUBST;
  *wp++ ← 'X';
  *wp++ ← c;
  *wp++ ← '\0';
  *wp++ ← CSUBST;
  *wp++ ← 'X';
```

This code is used in section 345.

350. Also detect whether the current state is inside double quotes since sh/AT&T-ksh translate the `\"` to `"` in `"`...\"...`"`.

```
⟨ Detect backtick-expansion 350 ⟩ ≡
  PUSH_STATE(SBQUOTE);      /* ⟨ Lexing SBQUOTE state (` ... `) 358 ⟩ */
  *wp++ ← COMSUB;
  statep¬ls_sbquote.indquotes ← 0;
  Lex_state *s ← statep;
  Lex_state *base ← state_info.base;
  while (1) {
    for ( ; s ≠ base; s--) {
      if (s¬ls_state ≡ SDQUOTE) {
        statep¬ls_sbquote.indquotes ← 1;
        break;      /* ... from ⟨ Detect escapes and substitutions 343 ⟩ */
      }
    }
    if (s ≠ base) break;                    /* — „ — */
    if (¬(s ← s¬ls_info.base)) break;       /* — „ — */
    base ← s-- - STATE_BSIZE;
  }
```

This code is used in section 343.

351. If "`'`" is found when scanning a single-quoted string, append `CHAR` & `'` if scanning in `"${ ... }"`, otherwise `CQUOTE`, and pop to the previous state. Other characters are appended after `QCHAR`.

```
⟨ Lexing SSQUOTE state ('...') 351 ⟩ ≡
  if (c ≡ '\'') {
    POP_STATE();
    if (state ≡ SBRACEQ) {
      *wp++ ← CHAR, *wp++ ← c;
      break;
    }
    *wp++ ← CQUOTE;
    ignore_backslash_newline--;
  }
  else *wp++ ← QCHAR, *wp++ ← c;
  break;
```

This code is cited in sections 342 and 361.

This code is used in section 340.

352. Finding the end of a double-quoted string simply ends it and pops state, otherwise attempt substitution in ⟨ Detect escapes and substitutions 343 ⟩.

```
⟨ Lexing SDQUOTE state ("...") 352 ⟩ ≡
  if (c ≡ '"') {
    POP_STATE();
    *wp++ ← CQUOTE;
  }
  else goto Subst;      /* ⟨ Detect escapes and substitutions 343 ⟩ */
  break;
```

This code is cited in section 342.

This code is used in section 340.

353. To scan a command substitution (`$( ... )`) the state machine has an inner state machine (but this time with magic numbers). The **goto**s in the comments below indicate which inner-state within the command substitution to enter next time the analyser reaches this `SCSPAREN` state.

After the character is scanned if the inner state machine has returned to normal (ie. all parentheses and quotes are matched and escapes are complete) the lexing state is popped and zero is appended to the output indicating the end of a command substitution (`COMSUB`). Otherwise the character is part of the command substitution and is appended.

```
⟨ Lexing SCSPAREN state ($( ... )) 353 ⟩ ≡
  switch (statep→ls_scsparen.csstate) {
  case 0:      /* normal */
    switch (c) {
    case '(': statep→ls_scsparen.nparen++; break;
    case ')': statep→ls_scsparen.nparen--; break;
    case '\\': statep→ls_scsparen.csstate ← 1; break;      /* goto \ in normal */
    case '"': statep→ls_scsparen.csstate ← 2; break;      /* goto " */
    case '\'': statep→ls_scsparen.csstate ← 4;      /* goto ' */
      ignore_backslash_newline++;
      break;
    }
    break;
  case 1:      /* \ in normal */
  case 3:      /* \ in " */
    --statep→ls_scsparen.csstate;      /* goto normal or " */
    break;
  case 2:      /* " */
    if (c ≡ '"') statep→ls_scsparen.csstate ← 0;      /* goto normal */
    else if (c ≡ '\\') statep→ls_scsparen.csstate ← 3;      /* goto \ in " */
    break;
  case 4:      /* ' */
    if (c ≡ '\'') {
      statep→ls_scsparen.csstate ← 0;      /* goto normal */
      ignore_backslash_newline--;
    }
    break;
  }
  if (statep→ls_scsparen.nparen ≡ 0) {
    POP_STATE();
    *wp++ ← 0;      /* end of COMSUB */
  }
  else *wp++ ← c;
  break;
```

This code is cited in sections 346 and 354.

This code is used in section 340.

354. Rather than being concerned with strings arithmetic substitution (`$(( ... ))`) is only concerned with matching parentheses. If the token ends and the parentheses don't match then command substitution is assumed and the state and output are adjusted as though that had been scanned for all along.

```
⟨ Lexing SASPAREN state ($(( ... ))) 354 ⟩ ≡
  if (c ≡ '(') statep→ls_sasparen.nparen++;
  else if (c ≡ ')') {
    statep→ls_sasparen.nparen−−;
    if (statep→ls_sasparen.nparen ≡ 1) {
      if ((c2 ← getsc()) ≡ ')') {
        POP_STATE();
        *wp++ ← 0;      /* end of EXPRSUB */
        break;
      }
      else {
        char *s;
        ungetsc(c2);
        s ← Xrestpos(ws, wp, statep→ls_sasparen.start);
        memmove(s + 1, s, wp − s);
        *s++ ← COMSUB;
        *s ← '(';
        wp++;
        statep→ls_scsparen.nparen ← 1;
        statep→ls_scsparen.csstate ← 0;
        state ← statep→ls_state ← SCSPAREN;      /* ⟨ Lexing SCSPAREN state ($( ... )) 353 ⟩ */
      }
    }
  }
  *wp++ ← c;
  break;
```

This code is cited in section 346.

This code is used in section 340.

355. `SBRACE` is looking for the closing "`}`" of "`${ ... }`" or attempts substitution in ⟨ Detect matching a pattern and **switch to** `SPATTERN` 341 ⟩.

```
⟨ Lexing SBRACE state (${⟨var⟩}) 355 ⟩ ≡
  if (c ≡ '}') {
    POP_STATE();
    *wp++ ← CSUBST;
    *wp++ ← '}';
  }
  else goto Sbase1;      /* ⟨ Detect matching a pattern and switch to SPATTERN 341 ⟩ */
  break;
```

This code is cited in section 347.

This code is used in section 340.

356. SBRACEQ is doing the same but inside double quotes, so falls back to ⟨ Unescape \ or detect quotes and **switch state** 342 ⟩.

```
⟨ Lexing SBRACEQ state ("${⟨var⟩}") 356 ⟩ ≡
  if (c ≡ '}') {
    POP_STATE();
    *wp++ ← CSUBST;
    *wp++ ← '}';
  }
  else goto Sbase2;      /* ⟨ Unescape \ or detect quotes and switch state 342 ⟩ */
  break;
```

This code is cited in section 347.

This code is used in section 340.

357. STBRACE occurs when "#" or "%" is encountered inside a ${-expansion that is looking for a variable name. "}" indicates the token has finished, "|" indicates a pattern separator and "(" beginning a pattern-group ("␣" is appended as a simile for an @(...) pattern).

```
⟨ Lexing STBRACE state (${var[#%]...}) 357 ⟩ ≡
  if (c ≡ '}') {
    POP_STATE();
    *wp++ ← CSUBST;
    *wp++ ← '}';
  }
  else if (c ≡ '|') {
    *wp++ ← SPAT;
  }
  else if (c ≡ '(') {
    *wp++ ← OPAT;
    *wp++ ← '␣';
    PUSH_STATE(SPATTERN);
  }
  else goto Sbase1;      /* ⟨ Detect matching a pattern and switch to SPATTERN 341 ⟩ */
  break;
```

This code is cited in section 347.

This code is used in section 340.

358. Reading a backtick expansion means looking for another "`". If instead "\\", "\$", "\`" or "\"" is encountered the escaped character is appended to the lexing output.

If the escapee was "" it is only appended as-is if it was inside a "` ... `" construct. Otherwise—or if any other character was escaped—\ and the escapee are appended. Everything else is appended as-is.

```
⟨ Lexing SBQUOTE state (` ... `) 358 ⟩ ≡
  if (c ≡ '`') {
    *wp++ ← 0;
    POP_STATE();
  }
  else if (c ≡ '\\') {
    switch (c ← getsc()) {
    case '\\': case '$': case '`':
      *wp++ ← c;
      break;
    case '"':
      if (statep¬ls_sbquote.indquotes) {
        *wp++ ← c;
        break;
      }
      else      /* FALLTHROUGH */
    default:
      if (c) {      /* trailing  is lost */
        *wp++ ← '\\';
        *wp++ ← c;
      }
      break;
    }
  }
  else *wp++ ← c;
  break;
```

This code is cited in section 350.

This code is used in section 340.

359. This state is only entered during ⟨ Initialise the lexing state machine 339 ⟩ if the source is coming from a single word.

```
⟨ Lexing SWORD state (single word) 359 ⟩ ≡
  goto Subst;      /* ⟨ Detect escapes and substitutions 343 ⟩ */
```

This code is cited in section 339.

This code is used in section 340.

360. As with arithmetic substitution a *let*-expression (`(( ... ))`) is concerned only with counting parentheses. If a mismatch occurs ("`)`" in the final place not followed by another "`)`") the closing parenthesis is returned to the read buffer and lexing resumes.

```
⟨ Lexing SLETPAREN state (( ... )) 360 ⟩ ≡
  if (c ≡ ')') {
    if (statep→ls_sletparen.nparen > 0) --statep→ls_sletparen.nparen;
    else if ((c2 ← getsc()) ≡ ')') {
      c ← 0;
      *wp++ ← CQUOTE;
      goto Done;
    }
    else ungetsc(c2);
  }
  else if (c ≡ '(') ++statep→ls_sletparen.nparen;
      /* parenthesis inside quotes and backslashes are lost, but AT&T ksh doesn't count them either */
  goto Sbase2;      /* ⟨ Unescape \ or detect quotes and switch state 342 ⟩ */
```

This code is cited in section 339.

This code is used in section 340.

361. When the syntax parser below is given a `<<` or `<<-` token it calls back into *yylex* with the `HEREDELIM` flag and the initial state is set to `SHEREDELIM` to read the heredoc's delimiter.

In this state escape sequences are appended verbatim and single & double quotes are detected, in the latter case switching to the `SHEREDQUOTE` (⟨ Lexing `SHEREDQUOTE` state (`"` in `<<`/`<<-` delimiter) 362 ⟩)[1] not `SDQUOTE`.

```
⟨ Lexing SHEREDELIM state (<<|<<- delimiter) 361 ⟩ ≡
  if (c ≡ '\\') {
    c ← getsc();
    if (c) {      /* trailing "\" is lost */
      *wp++ ← QCHAR;
      *wp++ ← c;
    }
  }
  else if (c ≡ '\'') {
    PUSH_STATE(SSQUOTE);      /* ⟨ Lexing SSQUOTE state ('...') 351 ⟩ */
    *wp++ ← OQUOTE;
    ignore_backslash_newline++;
  }
  else if (c ≡ '"') {
    state ← statep→ls_state ← SHEREDQUOTE;      /* (see commentary)[1] */
    *wp++ ← OQUOTE;
  }
  else {
    *wp++ ← CHAR;
    *wp++ ← c;
  }
  break;
```

This code is cited in sections 339 and 362.

This code is used in section 340.

[1] The full comment is too wide in the source listing.

362. In this state the analyser unescapes things which ought to be and otherwise appends characters until encountering an unescaped "`"`" and switching back to `SHEREDELIM`.

```
⟨ Lexing SHEREDQUOTE state (" in <</<<- delimiter) 362 ⟩ ≡
  if (c ≡ '"') {
    *wp++ ← CQUOTE;
    state ← statep¬ls_state ← SHEREDELIM;      /* ⟨ Lexing SHEREDELIM state (<</<<- delimiter) 361 ⟩ */
  }
  else {
    if (c ≡ '\\') {
      switch (c ← getsc()) {
      case '\\': case '"': case '$': case '`': break;
      default:
        if (c) {      /* trailing lost */
          *wp++ ← CHAR;
          *wp++ ← '\\';
        }
        break;
      }
    }
    *wp++ ← CHAR;
    *wp++ ← c;
  }
  break;
```

This code is cited in section 361.

This code is used in section 340.

363. Almost no special characters can be encountered in a pattern so detecting one is exceptionally simple.

```
⟨ Lexing SPATTERN state ([*+?@!](...|...)) 363 ⟩ ≡
  if (c ≡ ')') {
    *wp++ ← CPAT;
    POP_STATE();
  }
  else if (c ≡ '|') {
    *wp++ ← SPAT;
  }
  else if (c ≡ '(') {
    *wp++ ← OPAT;
    *wp++ ← ' ';
    PUSH_STATE(SPATTERN);      /* (same state, recursively) */
  }
  else goto Sbase1;      /* ⟨ Detect matching a pattern and switch to SPATTERN 341 ⟩ */
  break;
```

This code is cited in section 341.

This code is used in section 340.

364. If the token discovered indicates some sort of I/O redirection then it is recorded in this object.

```
#define IOTYPE 0xF /* type: bits 0:3 */
#define IOREAD 0x1 /* < */
#define IOWRITE 0x2 /* > */
#define IORDWR 0x3 /* <> (TODO) */
#define IOHERE 0x4 /* << (heredoc) */
#define IOCAT 0x5 /* >> */
#define IODUP 0x6 /* <&/>& */
#define IOEVAL BIT(4) /* expand in << */
#define IOSKIP BIT(5) /* in <<-, skip leading tabs */
#define IOCLOB BIT(6) /* >|; override -o noclobber */
#define IORDUP BIT(7) /* x<&y (as opposed to x>&y) */
#define IONAMEXP BIT(8) /* name has been expanded */
```

⟨ Type definitions 17 ⟩ +≡

```
struct ioword {
  int unit; /* unit affected */
  int flag; /* action (see above) */
  char *name; /* file name (unused if heredoc) */
  char *delim; /* delimiter for <</<<- */
  char *heredoc; /* content of heredoc */
};
```

365. This is a rather mechanical transcription of the desired redirection into an **ioword** object. Recall that *c2* was set in the test that enters this section by $c2 \leftarrow Xlength(ws, wp)$.

⟨ Prepare redirection token and **return** 365 ⟩ ≡

```
struct ioword *iop ← alloc(sizeof (*iop), ATEMP);
if (c2 ≡ 2) iop→unit ← dp[1] − '0';
else iop→unit ← c ≡ '>'; /* 0 for <, 1 (true) for > */
c2 ← getsc( );
if (c ≡ c2 ∨ (c ≡ '<' ∧ c2 ≡ '>')) { /* "<<", ">>" & "<>" are OK, "><" is not */
  iop→flag ← c ≡ c2 ?
      (c ≡ '>' ? IOCAT : IOHERE) : IORDWR;
  if (iop→flag ≡ IOHERE) {
    if ((c2 ← getsc( )) ≡ '-') iop→flag |= IOSKIP;
    else ungetsc(c2);
  }
}
else if (c2 ≡ '&')
  iop→flag ← IODUP | (c ≡ '<' ? IORDUP : 0);
else {
  iop→flag ← c ≡ '>' ? IOWRITE : IOREAD;
  if (c ≡ '>' ∧ c2 ≡ '|') iop→flag |= IOCLOB;
  else ungetsc(c2);
}
iop→name ← Λ;
iop→delim ← Λ;
iop→heredoc ← Λ;
Xfree(ws, wp); /* free word */
yylval.iop ← iop;
return REDIR;
```

This code is used in section 338.

366. If there was no token then consider instead how the current character is terminating the previous token. If it's the pair "||", "&&" or ";;", or "|&", then the appropriate symbol is returned. If it's a newline then any pending heredocs are consumed and the analyser is restarted. If not in sh mode and "((" is encountered MDPAREN is returned.

```
⟨ Process no token and return or restart 366 ⟩ ≡
  Xfree(ws, wp);      /* free word buffer */
  switch (c) {
  default: return c;
  case '|': case '&': case ';':
    if ((c2 ← getsc()) ≡ c)
          c ← (c ≡ ';') ? BREAK :
              (c ≡ '|') ? LOGOR :
              (c ≡ '&') ? LOGAND :
              YYERRCODE;      /* unreachable */
    else if (c ≡ '|' ∧ c2 ≡ '&') c ← COPROC;
    else ungetsc(c2);
    return c;
  case '\n':
    gethere();
    if (cf & CONTIN) goto Again;
    return c;
  case '(':
    if (¬Flag(FSH)) {
      if ((c2 ← getsc()) ≡ '(') c ← MDPAREN;      /* XXX need to handle ((...); (...)) */
      else ungetsc(c2);
    }
    return c;
  case ')': return c;      /* superfluous */
  }
```

This code is used in section 338.

367. Up to IDENT characters from the output stream are copied to *ident* (without their "lexeme" prefixes), which is '\0'-padded[1].

```
⟨ Finish the token (may return or restart) 367 ⟩ ≡
  *wp++ ← EOS;      /* terminate word */
  yylval.cp ← Xclose(ws, wp);
  if (state ≡ SWORD ∨ state ≡ SLETPAREN) return LWORD;      /* ONEWORD? */
  ungetsc(c);      /* unget terminator */
  for (sp ← yylval.cp, dp ← ident;
       dp < ident + IDENT ∧ (c ← *sp++) ≡ CHAR;
       /* — */ )
    *dp++ ← *sp++;      /* copy word to unprefixed string ident */
  memset(dp, 0, (ident + IDENT) − dp + 1);      /* Make sure the ident array stays '\0' padded */
  if (c ≠ EOS) *ident ← '\0';      /* word is not unquoted */
  if (*ident ≠ '\0' ∧ (cf & (KEYWORD | ALIAS))) {⟨ Finish an ALIAS or KEYWORD 369 ⟩}
```

This code is cited in section 338.

This code is used in section 338.

[1] TODO: Is there an off-by-one here (**for** test should use IDENT − 1)?

368. Lexing-Supporting Functions. When the lexical analyser needs to temporarily query another **Source** object it pushes a new one onto the source stack using *pushs*.

```
⟨ lex.c 314 ⟩ +≡
  Source *pushs(int type, Area *areap)
  {
    Source *s;
    s ← alloc(sizeof(Source), areap);
    s→type ← type;
    s→str ← null;
    s→start ← Λ;
    s→line ← 0;
    s→cmd_offset ← 0;
    s→errline ← 0;
    s→file ← Λ;
    s→flags ← 0;
    s→next ← Λ;
    s→areap ← areap;
    if (type ≡ SFILE ∨ type ≡ SSTDIN) {
      char *dummy;
      Xinit(s→xs, dummy, 256, s→areap);
    }
    else memset(&s→xs, 0, sizeof (s→xs));
    return s;
  }
```

369. Aliases have some sort of trivial magic to do with trailing spaces and a kludge in the character reader before the lexer even gets involved that I can't be bothered to work out right now. This completes it.

```
⟨ Finish an ALIAS or KEYWORD 369 ⟩ ≡
  struct tbl *p;
  int h ← hash(ident);
  if ((cf & KEYWORD) ∧ (p ← ktsearch(&keywords, ident, h)) ∧
        (¬(cf & ESACONLY) ∨ p→val.i ≡ ESAC ∨ p→val.i ≡ '}')) {
    afree(yylval.cp, ATEMP);
    return p→val.i;
  }
  if ((cf & ALIAS) ∧ (p ← ktsearch(&aliases, ident, h)) ∧ (p→flag & ISSET)) {
    Source *s;
    for (s ← source; s→type ≡ SALIAS; s ← s→next)
      if (s→u.tblp ≡ p) return LWORD;
    ;       /* push alias expansion: */
    s ← pushs(SALIAS, source→areap);
    s→start ← s→str ← p→val.s;
    s→u.tblp ← p;
    s→next ← source;
    source ← s;
    afree(yylval.cp, ATEMP);
    goto Again;
  }
```

This code is used in section 367.

370. C-shell-style history. Does anyone use this? TODO: Figure out what this is doing.

```
⟨ Lexing SBASE handle csh-style history 370 ⟩ ≡
  char **replace ← Λ;
  int get, i;
  char match[200] ← {0}, *str ← match;
  size_t mlen;
  c2 ← getsc( );
  if (c2 ≡ '\0' ∨ c2 ≡ '␣' ∨ c2 ≡ '\t') ;
  else if (c2 ≡ '!') replace ← hist_get_newest(0);
  else if (isdigit(c2) ∨ c2 ≡ '-' ∨ isalpha(c2)) {
    get ← ¬isalpha(c2);
    *str++ ← c2;
    do {
      if ((c2 ← getsc( )) ≡ '\0') break;
      if (c2 ≡ '\t' ∨ c2 ≡ '␣' ∨ c2 ≡ '\n') {
        ungetsc(c2);
        break;
      }
      *str++ ← c2;
    } while (str < &match[sizeof (match) − 1]);
    *str ← '\0';
    if (get) {
      int h ← findhistrel(match);
      if (h ≥ 0) replace ← &history[h];
    }
    else {
      int h ← findhist(−1, 0, match, true);
      if (h ≥ 0) replace ← &history[h];
    }
  }
  ;    /* XXX ksh history buffer saves un-expanded commands. Until the history buffer code is changed
      to contain expanded commands, we ignore the bad commands (spinning sucks) */
  if (replace ∧ **replace ≡ '!') ungetsc(c2);
  else if (replace) {
    Source *s;    /* do not strdup replacement via alloc */
    s ← pushs(SREREAD, source→areap);
    s→start ← s→str ← *replace;
    s→next ← source;
    s→u.freeme ← Λ;
    source ← s;
    continue;
  }
  else if (*match ≠ '\0') {    /* restore what followed the '!' */
    mlen ← strlen(match);
    for (i ← mlen − 1; i ≥ 0; i−−) ungetsc(match[i]);
  }
  else ungetsc(c2);
```

This code is used in section 340.

371. If a token looks like assignment to an array (`foo[n]=...`) then this figures out what n is.

```
⟨ Detect assignment to an array 371 ⟩ ≡
  *wp ← EOS;      /* temporary */
  if (is_wdvarname(Xstring(ws, wp), false)) {
    char *p, *tmp;
    if (arraysub(&tmp)) {
      *wp++ ← CHAR;
      *wp++ ← c;
      for (p ← tmp; *p; ) {
        Xcheck(ws, wp);
        *wp++ ← CHAR;
        *wp++ ← *p++;
      }
      afree(tmp, ATEMP);
      break;
    }
    else {
      Source *s;
      s ← pushs(SREREAD, source→areap);
      s→start ← s→str ← s→u.freeme ← tmp;
      s→next ← source;
      source ← s;
    }
  }
  *wp++ ← CHAR;
  *wp++ ← c;
  break;
```

This code is used in section 340.

372. To read heredocs from the input stream (which should be byte-for-byte, generally, and not lexed) *gethere* walks the pending list in *heres*.

```
⟨ lex.c 314 ⟩ +≡
  static void gethere(void)
  {
    struct ioword **p;
    for (p ← heres; p < herep; p++) readhere(*p);
    herep ← heres;
  }
```

373. Reading a heredoc involves creating a temporary file which will later be associated with a file descriptor for the command in question. This parses a single heredoc into a closed **XString**.

```
⟨ lex.c 314 ⟩ +≡
  static void readhere(struct ioword *iop)
  {
    int c;
    char *volatile eof;
    char *eofp;
    int skiptabs;
    XString xs;
    char *xp;
    int xpos;
    eof ← evalstr(iop→delim, 0);
    if (¬(iop→flag & IOEVAL)) ignore_backslash_newline++;
    Xinit(xs, xp, 256, ATEMP);
    for ( ; ; ) {
      eofp ← eof;
      skiptabs ← iop→flag & IOSKIP;
      xpos ← Xsavepos(xs, xp);
      while ((c ← getsc()) ≠ 0) {
        if (skiptabs) {
          if (c ≡ '\t') continue;
          skiptabs ← 0;
        }
        if (c ≠ *eofp) break;
        Xcheck(xs, xp);
        Xput(xs, xp, c);
        eofp++;
      }
      if (*eofp ≡ '\0' ∧ (c ≡ 0 ∨ c ≡ '\n')) {      /* Allow EOF here so commands without trailing
            newlines will work (eg. ksh -c '...', $(...), etc.). */
        xp ← Xrestpos(xs, xp, xpos);
        break;
      }
      ungetsc(c);
      while ((c ← getsc()) ≠ '\n') {
        if (c ≡ 0) yyerror("here␣document␣`%s'␣unclosed\n", eof);
        Xcheck(xs, xp);
        Xput(xs, xp, c);
      }
      Xcheck(xs, xp);
      Xput(xs, xp, c);
    }
    Xput(xs, xp, '\0');
    iop→heredoc ← Xclose(xs, xp);
    if (¬(iop→flag & IOEVAL)) ignore_backslash_newline--;
  }
```

374. Append an array subscript to the lexing output stream. Returns true if a matching bracket was found, false if EOF or newline was found (the returned string is double-`'\0'` terminated)[1].

```
⟨ lex.c 314 ⟩ +≡
  static int arraysub(char **strp)
  {
    XString ws;
    char *wp;
    char c;
    int depth ← 1;      /* we are just past the initial [ */
    Xinit(ws, wp, 32, ATEMP);
    do {
      c ← getsc( );
      Xcheck(ws, wp);
      *wp++ ← c;
      if (c ≡ '[') depth++;
      else if (c ≡ ']') depth−−;
    } while (depth > 0 ∧ c ∧ c ≠ '\n');
    *wp++ ← '\0';
    *strp ← Xclose(ws, wp);
    return depth ≡ 0 ? 1 : 0;
  }
```

375. Scanning for a variable is more involved (TODO: put this above as per `lex.c`?). Here we have yet another state machine.

```
⟨ lex.c 314 ⟩ +≡
  static char *get_brace_var(XString *wsp, char *wp)
  {
    enum parse_state {
      PS_INITIAL, PS_SAW_HASH, PS_IDENT, PS_NUMBER, PS_VAR1, PS_END
    } state;
    char c;
    state ← PS_INITIAL;
    while (1) {
      c ← getsc( );
      ⟨ Figure out where the variable-name part ends 376 ⟩
      if (state ≡ PS_END) {
        *wp++ ← '\0';      /* end of variable part */
        ungetsc(c);
        break;
      }
      Xcheck(*wsp, wp);
      *wp++ ← c;
    }
    return wp;
  }
```

[1] According to `lex.c`; I'm not seeing it.

376. ⟨ Figure out where the variable-name part ends 376 ⟩ ≡

```
switch (state) {
case PS_INITIAL:
  if (c ≡ '#') {
    state ← PS_SAW_HASH;
    break;
  }
  else      /* FALLTHROUGH */
case PS_SAW_HASH:
  if (letter(c)) state ← PS_IDENT;
  else if (digit(c)) state ← PS_NUMBER;
  else if (ctype(c, C_VAR1)) state ← PS_VAR1;
  else state ← PS_END;
  break;
case PS_IDENT:
  if (¬letnum(c)) {
    state ← PS_END;
    if (c ≡ '[') {
      char *tmp, *p;
      if (¬arraysub(&tmp)) yyerror("missing ]\n");
      *wp++ ← c;
      for (p ← tmp; *p; ) {
        Xcheck(*wsp, wp);
        *wp++ ← *p++;
      }
      afree(tmp, ATEMP);
      c ← getsc();      /* the ] */
    }
  }
  break;
case PS_NUMBER:
  if (¬digit(c)) state ← PS_END;
  break;
case PS_VAR1: state ← PS_END;
  break;
case PS_END:      /* keep gcc happy */
  break;
}
```

This code is used in section 375.

377. Interactive Prompt. When asking a user for the next line of input ksh outputs a prompt using another custom printf-like formatter for the user to customise. For some reason this requires a routine which replaces any part of `$PS1` which isn't exactly "`\$`" with "`p`".

```
⟨ lex.c 314 ⟩ +≡
  static char *special_prompt_expand(char *str)
  {
    char *p ← str;
    while ((p ← strstr(p, "\\$")) ≠ Λ) {
      *(p + 1) ← 'p';
    }
    return str;
  }
```

378. Here is *set_prompt* which needs that.

```
#define PS1 0    /* command */
#define PS2 1    /* command continuation */
⟨ lex.c 314 ⟩ +≡
  void set_prompt(int to)
  {
    char *ps1;
    Area *saved_atemp;
    cur_prompt ← to;
    switch (to) {
    case PS1:    /* command */
      ps1 ← str_save(str_val(global("PS1")), ATEMP);
      saved_atemp ← ATEMP;    /* ps1 is freed by substitute */
      newenv(E_ERRH);
      if (sigsetjmp(genv→jbuf, 0)) {
        prompt ← safe_prompt;
          /* Don't print an error—assume it has already been printed. Reason is we may have forked to
             run a command and the child may be unwinding its stack through this code as it exits. */
      }
      else {    /* expand $... before other substitutions are done */
        char *tmp ← special_prompt_expand(ps1);
        prompt ← str_save(substitute(tmp, 0), saved_atemp);
      }
      quitenv(Λ);
      break;
    case PS2:    /* command continuation */
      prompt ← str_val(global("PS2"));
      break;
    }
  }
```

379. The main prompt-printing routine (which uses *set_prompt*) has two front-ends, one to print it and another to return its length.

```
⟨ lex.c 314 ⟩ +≡
  void pprompt(const char *cp, int ntruncate)
  { doprompt(cp, ntruncate, Λ, 1); }
  int promptlen(const char *cp, const char **spp)
  { return doprompt(cp, 0, spp, 0); }
```

380. Construct the current prompt into a string buffer. Capture the delimiter if there is one.

```
⟨ lex.c 314 ⟩ +≡
  static int doprompt(const char *sp, int ntruncate, const char **spp, int doprint)
  {
    char strbuf[1024], tmpbuf[1024], *p, *str, nbuf[32], delimiter ← '\0';
    int len, c, n, totlen ← 0, indelimit ← 0, counting ← 1, delimitthis;
    const char *cp ← sp;
    struct tm *tm;
    time_t t;
    if (*cp ∧ cp[1] ≡ '\r') {
      delimiter ← *cp;
      cp += 2;
    }
    while (*cp ≠ 0) {⟨ Copy the next prompt character 381 ⟩}
    if (doprint) shf_flush(shl_out);
    if (spp) *spp ← sp;
    return (totlen);
  }
```

381. Characters that aren't a delimiter (usually (always?) `'\0'`) are copied as-is. A backslash is treated specially.

```
⟨ Copy the next prompt character 381 ⟩ ≡
  delimitthis ← 0;
  if (indelimit ∧ ∗cp ≠ delimiter) ;      /∗ do nothing ∗/
  else if (∗cp ≡ '\n' ∨ ∗cp ≡ '\r') {      /∗ reset totlen to 0 on newline ∗/
    totlen ← 0;
    sp ← cp + 1;
  }
  else if (∗cp ≡ '\t') {
    if (counting) totlen ← (totlen | 7) + 1;      /∗ increase totlen to modulo 8 on tab ∗/
  }
  else if (∗cp ≡ delimiter) {
    indelimit ← ¬indelimit;
    delimitthis ← 1;
  }
  if (∗cp ≡ '\\') {⟨ Found \ in a prompt, determine what it is 382 ⟩}
  else if (∗cp ≠ '!') c ← ∗cp++;      /∗ copy anything which isn't "!" ∗/
  else if (∗++cp ≡ '!') c ← ∗cp++;      /∗ resolve "!!" to "!" ∗/
  else {⟨ Inject the current line number 383 ⟩}
  if (counting ∧ ntruncate) −−ntruncate;
  else if (doprint) {
    shf_putc(c, shl_out);
  }
  if (counting ∧ ¬indelimit ∧ ¬delimitthis) totlen++;
```

This code is used in section 380.

382. sh only understands the meta-characters \h and \$ which is somehow represented internally as \p.

```
⟨ Found \ in a prompt, determine what it is 382 ⟩ ≡
  cp++;
  if (¬*cp) break;
  if (Flag(FSH) ∧ ¬(*cp ≡ 'h' ∨ *cp ≡ 'p'))      /* sh only understands \h and \$[1] */
    snprintf(strbuf, sizeof strbuf, "\\%c", *cp);
  else
    switch (*cp) {⟨ Expand a prompt meta-character 384 ⟩}
  cp++;
  str ← strbuf;
  len ← strlen(str);
  if (ntruncate) {
    if (ntruncate ≥ len) {
      ntruncate −= len;
      continue;
    }
    str += ntruncate;
    len −= ntruncate;
    ntruncate ← 0;
  }
  if (doprint) shf_write(str, len, shl_out);
  if (counting ∧ ¬indelimit ∧ ¬delimitthis) totlen += len;
  continue;
```

This code is used in section 381.

383.

```
⟨ Inject the current line number 383 ⟩ ≡
  shf_snprintf(p ← nbuf, sizeof (nbuf), "%d", source→line + 1);
  len ← strlen(nbuf);
  if (ntruncate) {
    if (ntruncate ≥ len) {
      ntruncate −= len;
      continue;
    }
    p += ntruncate;
    len −= ntruncate;
    ntruncate ← 0;
  }
  if (doprint) shf_write(p, len, shl_out);
  if (counting ∧ ¬indelimit ∧ ¬delimitthis) totlen += len;
  continue;
```

This code is used in section 381.

[1] The comment said $ but the code says p...

384. The beast itself. There is no reasonable way to break this up into pieces other than **case**-by-**case**. Starting with `\a` to ring the bell.

Beware of the octal.

```
⟨ Expand a prompt meta-character 384 ⟩ ≡
case 'a': strbuf[0] ← '\007';
  strbuf[1] ← '\0';
  break;
```

See also sections 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, and 411.

This code is used in section 382.

385. `\d`: "<*day-of-week*>␣<*month*>␣<*numeric-day-of-month*>".

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'd': time(&t);
  tm ← localtime(&t);
  strftime(strbuf, sizeof strbuf, "%a␣%b␣%d", tm);
  break;
```

386. `\D{`*format*`}`: `{<`*strftime*(format)`>}`.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'D': p ← strchr(cp + 2, '}');
  if (cp[1] ≠ '{' ∨ p ≡ Λ) {
    snprintf(strbuf, sizeof strbuf, "\\%c", ∗cp);
    break;
  }
  strlcpy(tmpbuf, cp + 2, sizeof tmpbuf);
  p ← strchr(tmpbuf, '}');
  if (p) ∗p ← '\0';
  time(&t);
  tm ← localtime(&t);
  strftime(strbuf, sizeof strbuf, tmpbuf, tm);
  cp ← strchr(cp + 2, '}');
  break;
```

387. `\e`: Escape.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'e': strbuf[0] ← '\033';
  strbuf[1] ← '\0';
  break;
```

388. `\h`: Shortened hostname.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'h': gethostname(strbuf, sizeof strbuf);
  p ← strchr(strbuf, '.');
  if (p) ∗p ← '\0';
  break;
```

389. `\H`: Full hostname.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'H': gethostname(strbuf, sizeof strbuf);
  break;
```

390. `\j`: Number of jobs.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'j': snprintf(strbuf, sizeof strbuf, "%d", j_njobs());
  break;
```

391. `\l`: Basename of TTY.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'l': p ← ttyname(0);
  if (p) p ← basename(p);
  if (p) strlcpy(strbuf, p, sizeof strbuf);
  break;
```

392. `\n`: Newline; resets the line length to zero.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'n': strbuf[0] ← '\n';
  strbuf[1] ← '\0';
  totlen ← 0;
  sp ← cp + 1;
  break;
```

393. `\$`: "`$`" or "`#`" depending on user id.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'p': strbuf[0] ← ksheuid ? '$' : '#';
  strbuf[1] ← '\0';
  break;
```

394. `\r`: Return to character position 0 (and reset the length).

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'r': strbuf[0] ← '\r';
  strbuf[1] ← '\0';
  totlen ← 0;
  sp ← cp + 1;
  break;
```

395. `\s`: `` `basename␣$0` ``; word after last "`/`" of `$0`.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 's': strlcpy(strbuf, kshname, sizeof strbuf);
  break;
```

396. `\t`: 24-hour time formatted as `HH:MM:SS`.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 't': time(&t);
  tm ← localtime(&t);
  strftime(strbuf, sizeof strbuf, "%T", tm);
  break;
```

397. `\T`: 12-hour time formatted as `HH:MM:SS`.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'T': time(&t);
  tm ← localtime(&t);
  strftime(strbuf, sizeof strbuf, "%l:%M:%S", tm);
  break;
```

398. `\@`: 12-hour time formatted locale-dependent.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case '@': time(&t);
  tm ← localtime(&t);
  strftime(strbuf, sizeof strbuf, "%r", tm);
  break;
```

399. `\A`: 12-hour time formatted as `HH:MM`.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'A': time(&t);
  tm ← localtime(&t);
  strftime(strbuf, sizeof strbuf, "%R", tm);
  break;
```

400. `\u`: Username.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'u': strlcpy(strbuf, username, sizeof strbuf);
  break;
```

401. `\v`: Short version.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'v': p ← strchr(ksh_version, '␣');
  if (p) p ← strchr(p + 1, '␣');
  if (p) {
    p++;
    strlcpy(strbuf, p, sizeof strbuf);
    p ← strchr(strbuf, '␣');
    if (p) *p ← '\0';
  }
  break;
```

402. `\V`: Long version.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'V': strlcpy(strbuf, ksh_version, sizeof strbuf);
  break;
```

403. `\w`: Current directory. This also takes care of replacing the user's home directory with "`~`".

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'w': p ← str_val(global("PWD"));
  n ← strlen(str_val(global("HOME")));
  if (strcmp(p, "/") ≡ 0) {
    strlcpy(strbuf, p, sizeof strbuf);
  }
  else if (strcmp(p, str_val(global("HOME"))) ≡ 0) {
    strbuf[0] ← '~';
    strbuf[1] ← '\0';
  }
  else if (strncmp(p, str_val(global("HOME")), n) ≡ 0 ∧ p[n] ≡ '/') {
    snprintf(strbuf, sizeof strbuf, "~/%s", str_val(global("PWD")) + n + 1);
  }
  else strlcpy(strbuf, p, sizeof strbuf);
  break;
```

404. `\W`: `` `basename $CWD` ``; word after last "`/`" of `$CWD`; also takes care of `$HOME` → `~`.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case 'W': p ← str_val(global("PWD"));
  if (strcmp(p, str_val(global("HOME"))) ≡ 0) {
    strbuf[0] ← '~';
    strbuf[1] ← '\0';
  }
  else strlcpy(strbuf, basename(p), sizeof strbuf);
  break;
```

405. `\!`: Current history line number.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case '!': snprintf(strbuf, sizeof strbuf, "%d", source→line + 1);
  break;
```

406. `\#`: Current command line number.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case '#': snprintf(strbuf, sizeof strbuf, "%d", source→line − source→cmd_offset + 1);
  break;
```

407. `\`[0-7][0-7][0-7]: Octal-representation of a byte.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
  if ((cp[1] > '7' ∨ cp[1] < '0') ∨ (cp[2] > '7' ∨ cp[2] < '0')) {
    snprintf(strbuf, sizeof strbuf, "\\%c", ∗cp);
    break;
  }
  n ← (cp[0] − '0') ∗ 8 ∗ 8 + (cp[1] − '0') ∗ 8 + (cp[2] − '0');
  snprintf(strbuf, sizeof strbuf, "%c", n);
  cp += 2;
  break;
```

408. `\\`: A backslash.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case '\\': strbuf[0] ← '\\';
  strbuf[1] ← '\0';
  break;
```

409. `\[`: Stop counting the prompt length.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case '[': strbuf[0] ← '\0';
  counting ← 0;
  break;
```

410. `\]`: Resume counting the prompt length.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
case ']': strbuf[0] ← '\0';
  counting ← 1;
  break;
```

411. **default**: Print the escapee.

```
⟨ Expand a prompt meta-character 384 ⟩ +≡
default: snprintf(strbuf, sizeof strbuf, "\\%c", *cp);
  break;
```

412. Compilation. A valid stream of lexemes can be parsed into a syntax tree for compilation née interpretation. This is started by *compile* (which simply calls *yyparse*) in `syn.c`.

```
⟨ syn.c 412 ⟩ ≡
#include <string.h>
#include "sh.h"
#include "c_test.h"
  static void yyparse(void);
  static struct Op *pipeline(int);
  static struct Op *andor(void);
  static struct Op *c_list(int);
  static struct ioword *synio(int);
  static void musthave(int, int);
  static struct Op *nested(int, int, int);
  static struct Op *get_command(int);
  static struct Op *dogroup(void);
  static struct Op *thenpart(void);
  static struct Op *elsepart(void);
  static struct Op *caselist(void);
  static struct Op *casepart(int);
  static struct Op *function_body(char *, int);
  static char **wordlist(void);
  static struct Op *block(int, struct Op *, struct Op *, char **);
  static struct Op *newtp(int);
  static void syntaxerr(const char *)__attribute__((__noreturn__));
  static void nesting_push(struct nesting_state *, int);
  static void nesting_pop(struct nesting_state *);
  static int assign_command(char *);
  static int inalias(struct Source *);
  static int dbtestp_isa(Test_env *, Test_meta);
  static const char *dbtestp_getopnd(Test_env *, Test_op, int);
  static int dbtestp_eval(Test_env *, Test_op, const char *, const char *, int);
  static void dbtestp_error(Test_env *, int, const char *);
  static struct Op *outtree;      /* yyparse output */
  static struct nesting_state nesting;      /* \n changed to ; */
  static int reject;      /* token(cf) gets symbol again */
  static int symbol;      /* yylex value */
```

See also sections 415, 416, 417, 418, 419, 420, 421, 423, 424, 426, 431, 432, 434, 435, 436, 438, 440, 442, 443, 447, 450, 454, 455, 459, 460, 1327, 1329, 1330, 1331, and 1332.

413. ⟨ Shared function declarations 4 ⟩ +≡

```
  void initkeywords(void);
  struct Op *compile(Source *);
```

414. The syntax tree parsed from a streme of lexemes is this *op* object which includes two links to its subordinate *op* objects. Each *op* represents one of these `T*` constants and its descendents its arguments, if any.

```
#define TEOF 0
#define TCOM 1    /* command */
#define TPAREN 2    /* (command-list) */
#define TPIPE 3    /* a | b */
#define TLIST 4    /* a ; b */
#define TOR 5    /* || */
#define TAND 6    /* && */
#define TBANG 7    /* ! */
#define TDBRACKET 8    /* [[ ... ]] */
#define TFOR 9
#define TSELECT 10
#define TCASE 11
#define TIF 12
#define TWHILE 13
#define TUNTIL 14
#define TELIF 15
#define TPAT 16    /* pattern in case */
#define TBRACE 17    /* {command-list} */
#define TASYNC 18    /* c & */
#define TFUNCT 19    /* function name { command; } */
#define TTIME 20    /* time pipeline */
#define TEXEC 21    /* fork+exec eval'd TCOM */
#define TCOPROC 22    /* coprocess |& */
```

⟨ Type definitions 17 ⟩ +≡

```
struct Op {
  short type;    /* operation type, see above */
  union {    /* WARNING: newtp, tcopy use evalflags ← 0 to clear union */
    short evalflags;    /* TCOM: arg expansion eval flags */
    short ksh_func;    /* TFUNC: function x (vs. x()) */
  } u;
  char **args;    /* arguments to a command */
  char **vars;    /* variable assignments */
  struct ioword **ioact;    /* I/O actions (eg. < > >>) */
  struct Op *left, *right;    /* descendents */
  char *str;    /* word for case; identifier for for, select and functions; path to execute for TEXEC;
      time hook for TCOM. */
  int lineno;    /* TCOM/TFUNC: $LINENO for this */
};
```

415. These two lists which map the text label of a token to its symbolic representation must match each other.

Although out of order and thus hard to see, these lists don't actually match perfectly. The three constants LWORD, REDIR and YYERRCODE are not represented in *tokentab* and neither do "{" and "}" have constant. YYERRCODE is of course not expected in the input stream—it's an error marker. The others, LWORD and REDIR, as well as EOF (0), are handled specially. I'm not sure yet why braces are odd.

```
#define LWORD 256
#define LOGAND 257    /* && */
#define LOGOR 258    /* || */
#define BREAK 259    /* ;; */
#define IF 260
#define THEN 261
#define ELSE 262
#define ELIF 263
#define FI 264
#define CASE 265
#define ESAC 266
#define FOR 267
#define SELECT 268
#define WHILE 269
#define UNTIL 270
#define DO 271
#define DONE 272
#define IN 273
#define FUNCTION 274
#define TIME 275
#define REDIR 276
#define MDPAREN 277    /* (( ... )) */
#define BANG 278    /* ! */
#define DBRACKET 279    /* [[ ... ]] */
#define COPROC 280    /* |& */
#define YYERRCODE 300
```

⟨ syn.c 412 ⟩ +≡

```
const struct tokeninfo {
  const char *name;
  short val;
  short reserved;
} tokentab[] ← {    /* Reserved words */
{"if", IF, true}, {"then", THEN, true}, {"else", ELSE, true}, {"elif", ELIF, true},
{"fi", FI, true}, {"case", CASE, true}, {"esac", ESAC, true}, {"for", FOR, true},
{"select", SELECT, true}, {"while", WHILE, true}, {"until", UNTIL, true}, {"do", DO, true},
{"done", DONE, true}, {"in", IN, true}, {"function", FUNCTION, true}, {"time", TIME, true},
{"{", '{', true}, {"}", '}', true}, {"!", BANG, true}, {"[[", DBRACKET, true},
{"&&", LOGAND, false}, {"||", LOGOR, false}, {";;", BREAK, false}, {"((", MDPAREN, false},
{"|&", COPROC, false},
{"newline", '\n', false},    /* and one (more) special case */
{0}};
```

416. The keywords database is initialised from this list by *initkeywords*.

```
⟨ syn.c 412 ⟩ +≡
  void initkeywords(void)
  {
    struct tokeninfo const *tt;
    struct tbl *p;
    ktinit(&keywords, APERM, 32);      /* must be 2^n (currently 20 keywords, table extends at 22) */
    for (tt ← tokentab; tt→name; tt++) {
      if (tt→reserved) {
        p ← ktenter(&keywords, tt→name, hash(tt→name));
        p→flag |= DEFINED | ISSET;
        p→type ← CKEYWD;
        p→val.i ← tt→val;
      }
    }
  }
```

417. Pipelines. The basic operation of the compiler is to obtain a complete token and depending on what it is request the next with appropriate *yylex* flags for the type of token expected. The token is represented by a **Op** object created by *newtp*.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *newtp(int type)
  {
    struct Op *t;
    t ← alloc(sizeof (*t), ATEMP);
    t→type ← type;
    t→u.evalflags ← 0;
    t→args ← t→vars ← Λ;
    t→ioact ← Λ;
    t→left ← t→right ← Λ;
    t→str ← Λ;
    return (t);
  }
```

418. The process is begun by in *compile* which immediately bubbles control up through *yyparse* → *c_list* → *andor* → *pipeline*, which is where the work begins and is responsible for combining tokens into an individual command (or rather, a pipeline of commands which the shell will execute as one unit, hence the name).

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *pipeline(int cf)
  {
    struct Op *t, *p, *tl ← Λ;
    t ← get_command(cf);
    if (t ≠ Λ) {
      while (token(0) ≡ '|') {
        if ((p ← get_command(CONTIN)) ≡ Λ) syntaxerr(Λ);
        if (tl ≡ Λ) t ← tl ← block(TPIPE, t, p, Λ);
        else tl ← tl→right ← block(TPIPE, tl→right, p, Λ);
      }
      reject ← true;
    }
    return (t);
  }
```

419. *pipeline* is reached via *andor* which connects together individual *pipeline* trees separated by "`&&`" & "`||`" into a sequence.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *andor(void)
  {
    struct Op *t, *p;
    int c;
    t ← pipeline(0);
    if (t ≠ Λ) {
      while ((c ← token(0)) ≡ LOGAND ∨ c ≡ LOGOR) {
        if ((p ← pipeline(CONTIN)) ≡ Λ) syntaxerr(Λ);
        t ← block(c ≡ LOGAND ? TAND : TOR, t, p, Λ);
      }
      reject ← true;
    }
    return (t);
  }
```

420. Parse a command list. The token saved into c has always been read/rejected at that point, so there is no need to worry about what flags to pass *token* née *yylex*.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *c_list(int multi)
  {
    struct Op *t ← Λ, *p, *tl ← Λ;
    int c;
    int have_sep;
    while (1) {
      p ← andor( );
      c ← token(0);
      have_sep ← 1;
      if (c ≡ '\n' ∧ (multi ∨ inalias(source))) {⟨ Ignore blank lines 422 ⟩}
      else if (¬p) break;
      else if (c ≡ '&' ∨ c ≡ COPROC) p ← block(c ≡ '&' ? TASYNC : TCOPROC, p, Λ, Λ);
      else if (c ≠ ';') have_sep ← 0;
      if (¬t) t ← p;
      else if (¬tl) t ← tl ← block(TLIST, t, p, Λ);
      else tl ← tl→right ← block(TLIST, tl→right, p, Λ);
      if (¬have_sep) break;
    }
    reject ← true;
    return t;
  }
```

421. Check if we are in the middle of reading an alias.

```
⟨ syn.c 412 ⟩ +≡
  static int inalias(struct Source *s)
  {
    for ( ; s ∧ s→type ≡ SALIAS; s ← s→next)
      if (¬(s→flags & SF_ALIASEND)) return 1;
    return 0;
  }
```

422. ⟨ Ignore blank lines 422 ⟩ ≡

```
  if (¬p) continue;
```

This code is used in section 420.

423.

```
⟨ syn.c 412 ⟩ +≡
  static void yyparse(void)
  {
    int c;
    reject ← false;
    outtree ← c_list(source→type ≡ SSTRING);
    c ← tpeek(0);
    if (c ≡ 0 ∧ ¬outtree) outtree ← newtp(TEOF);
    else if (c ≠ '\n' ∧ c ≠ 0) syntaxerr(Λ);
  }
```

424.

```
⟨ syn.c 412 ⟩ +≡
  struct Op *compile(Source *s)
  {
    nesting.start_token ← 0;
    nesting.start_line ← 0;
    herep ← heres;
    source ← s;
    yyparse( );
    return outtree;
  }
```

425. Now that we know the way in here is the heart of the parser, *get_command*.

426. Starting off with an empty tree the first token requests a token which may (also) be a KEYWORD, ALIAS or VARASN (variable assignment).

The resulting tree is created in t.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *get_command(int cf)
  {
    struct Op *t;
    int c, iopn ← 0, syniocf;
    struct ioword *iop, **iops;
    XPtrV args, vars;
    struct nesting_state old_nesting;
    iops ← areallocarray(Λ, NUFILE + 1, sizeof(struct ioword *), ATEMP);
    XPinit(args, 16);
    XPinit(vars, 16);
    syniocf ← KEYWORD | ALIAS;
    switch (c ← token(cf | KEYWORD | ALIAS | VARASN)) {
    default: reject ← true;
      afree(iops, ATEMP);
      XPfree(args);
      XPfree(vars);
      return Λ;      /* empty line */
    ⟨ Parse a command based on its first token 427 ⟩
    }
    ⟨ Process I/O redirection for irregular commands 453 ⟩
    if (t→type ≡ TCOM ∨ t→type ≡ TDBRACKET) {
      XPput(args, Λ);
      t→args ← (char **) XPclose(args);
      XPput(vars, Λ);
      t→vars ← (char **) XPclose(vars);
    }
    else {
      XPfree(args);
      XPfree(vars);
    }
    return t;
  }
```

427. The return value from *yylex* indicates what sort of token has been found. This will then consume as many tokens as it requires until a complete command has been obtained, or there was an error.

The first cases parse "normal" shell syntax and are jumped into from most others.

```
⟨ Parse a command based on its first token 427 ⟩ ≡
case LWORD: case REDIR: ⟨ Parse a regular command word 444 ⟩
Leave:
  break;
Subshell:
    /* FALLTHROUGH */
case '(': t ← nested(TPAREN, '(', ')'); break;
case '{': t ← nested(TBRACE, '{', '}'); break;
```

See also sections 428, 429, 430, 437, 439, 441, 449, 451, and 452.

This code is used in section 426.

428. The simplest command word is "`!`" which inverts the return value of the command that follows (0→1, anything else→0). The implementation consumes a pipeline and constructs a `TBANG` node to save it in.

```
⟨ Parse a command based on its first token 427 ⟩ +≡
case BANG: syniocf &= ∼(KEYWORD | ALIAS);
  t ← pipeline(0);
  if (t ≡ Λ) syntaxerr(Λ);
  t ← block(TBANG, Λ, t, Λ);
  break;
```

429. Like `!` "`time`" is followed by a regular pipeline command. Unlike `!` it needs two bytes in the *str* member for "`TF_*` flags". What this means will hopefully become evident as I ascend from the bowels of the ksh codebase.

```
⟨ Parse a command based on its first token 427 ⟩ +≡
case TIME: syniocf &= ∼(KEYWORD | ALIAS);
  t ← pipeline(0);
  if (t) {
    if (t→str) {
      t→str ← str_save(t→str, ATEMP);
    }
    else {
      t→str ← alloc(2, ATEMP);
      t→str[0] ← '\0';
      t→str[1] ← '\0';
    }
  }
  t ← block(TTIME, t, Λ, Λ);
  break;
```

430. Similarly simple with a twist is "`if`". The implementation consumes first a fresh command from *c_list*[1] to perform the test and result clause is consumed by *thenpart* below. These are put in the left and right branches of a new `TIF` node. After the consequent and optional alternate clauses have been read in, the next word **must** be the keyword "`fi`".

```
⟨ Parse a command based on its first token 427 ⟩ +≡
case IF: nesting_push(&old_nesting, c);
  t ← newtp(TIF);
  t→left ← c_list(true);
  t→right ← thenpart( );
  musthave(FI, KEYWORD | ALIAS);
  nesting_pop(&old_nesting);
  break;
```

[1] TODO: Why *c_list* not *andor*? Condition can include `&`, `;` & `|&`...

431. The first command word of a conditional result **must** be "`then`" and then a command list is consumed by *c_list* and put in a new node's left branch[1].

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *thenpart(void)
  {
    struct Op *t;
    musthave(THEN, KEYWORD | ALIAS);
    t ← newtp(0);
    t→left ← c_list(true);
    if (t→left ≡ Λ) syntaxerr(Λ);
    t→right ← elsepart( );
    return (t);
  }
```

432. An alternate clause of a conditional result can be indicated by either "`else`" or "`elif`". The simpler `else` returns the command list which follows it and is done.

In case of `elif` the alternative clause is actually another conditional so a `TELIF` node is created with the test command and the parser recurses back into *thenpart*. `TELIF` acts identically to `TIF`—the two are distinct so that they can be decompiled back into their original form.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *elsepart(void)
  {
    struct Op *t;
    switch (token(KEYWORD | ALIAS | VARASN)) {
    case ELSE:
      if ((t ← c_list(true)) ≡ Λ) syntaxerr(Λ);
      return (t);
    case ELIF: t ← newtp(TELIF);
      t→left ← c_list(true);
      t→right ← thenpart( );
      return (t);
    default: reject ← true;
    }
    return Λ;
  }
```

433. The twist wasn't the recursion, twisty as that may be, but nesting which is the mechanism by which multi-line commands can be read in full prior to evaluation—by interpreting a newline as `;` normally would be: a command separator rather than the input terminator.

The structure doubles as a reminder of the line number a nested command began on and which command it was.

```
⟨ Type definitions 17 ⟩ +≡
  struct nesting_state {
    int start_token;      /* token than began nesting (eg, FOR) */
    int start_line;       /* line which nesting began on */
  };
```

[1] TODO: How does *c_list* not consume the `else`?

434. Nesting functions maintain their own stack of previous nesting states.

```
⟨ syn.c 412 ⟩ +≡
  static void nesting_push(struct nesting_state *save, int tok)
  {
    *save ← nesting;
    nesting.start_token ← tok;
    nesting.start_line ← source→line;
  }
```

435.
```
⟨ syn.c 412 ⟩ +≡
  static void nesting_pop(struct nesting_state *saved)
  {
    nesting ← *saved;
  }
```

436. The identically-parsed "`( ... )`" and "`{ ... }`" are implemented by this—consume a command list and then the appropriate terminating marker, while nested.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *nested(int type, int smark, int emark)
  {
    struct Op *t;
    struct nesting_state old_nesting;
    nesting_push(&old_nesting, smark);
    t ← c_list(true);
    musthave(emark, KEYWORD | ALIAS);
    nesting_pop(&old_nesting);
    return (block(type, t, Λ, Λ));
  }
```

437. "`while`" and "`until`" are quite similar to `if`—a test command is first consumed then *dogroup* consumes the loop body. All nested to allow multi-line commands, the result is a `TWHILE` or `TUNTIL` node.

```
⟨ Parse a command based on its first token 427 ⟩ +≡
case WHILE: case UNTIL: nesting_push(&old_nesting, c);
  t ← newtp((c ≡ WHILE) ? TWHILE : TUNTIL);
  t→left ← c_list(true);
  if (t→left ≡ Λ) syntaxerr(Λ);
  t→right ← dogroup( );
  nesting_pop(&old_nesting);
  break;
```

438. The construct "`{ ... }`" can be used instead of "`do ...; done`" for `for`/`select` loops but not for `while`/`until` loops—we don't need to check if it is a `while` loop because it would have been parsed as part of the conditional command list. This consumes a command list and then demands whichever terminator is required.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *dogroup(void)
  {
    int c;
    struct Op *list;
    c ← token(CONTIN | KEYWORD | ALIAS);
    if (c ≡ DO) c ← DONE;
    else if (c ≡ '{') c ← '}';
    else syntaxerr(Λ);
    list ← c_list(true);
    musthave(c, KEYWORD | ALIAS);
    return list;
  }
```

439. The "`for`" and "`select`" loops as hinted above also use *dogroup*, after consuming an identifier followed by arguments.

```
⟨ Parse a command based on its first token 427 ⟩ +≡
case FOR: case SELECT: t ← newtp((c ≡ FOR) ? TFOR : TSELECT);
  musthave(LWORD, ARRAYVAR);
  if (¬is_wdvarname(yylval.cp, true)) yyerror("%s: bad identifier\n", c ≡ FOR ? "for" : "select");
  t→str ← str_save(ident, ATEMP);
  nesting_push(&old_nesting, c);
  t→vars ← wordlist( );
  t→left ← dogroup( );
  nesting_pop(&old_nesting);
  break;
```

440. *wordlist* consumes "`in`" then a list of items to iterate through, if present. AT&T ksh accepts "`;`" after the identifier if there is no list.

```
⟨ syn.c 412 ⟩ +≡
  static char **wordlist(void)
  {
    int c;
    XPtrV args;
    XPinit(args, 16);
    if ((c ← token(CONTIN | KEYWORD | ALIAS)) ≠ IN) {      /* POSIX does not do alias expansion here */
      if (c ≠ ';') reject ← true;      /* non-POSIX */
      return Λ;
    }
    while ((c ← token(0)) ≡ LWORD) XPput(args, yylval.cp);
    if (c ≠ '\n' ∧ c ≠ ';') syntaxerr(Λ);
    XPput(args, Λ);
    return (char **) XPclose(args);
  }
```

441. The final piece of syntax parsed in this manner is "case". The next expression in the input stream is consumed and copied into *str* in a TCASE node (this expression will be evaluated later), then the list of cases.

```
⟨ Parse a command based on its first token 427 ⟩ +≡
case CASE: t ← newtp(TCASE);
  musthave(LWORD, 0);
  t→str ← yylval.cp;
  nesting_push(&old_nesting, c);
  t→left ← caselist( );
  nesting_pop(&old_nesting);
  break;
```

442. Each case part returns a new TPAT node. First the patterns are consumed into *ptns* and saved in *t→vars* then after the closing ")" a command list.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *casepart(int endtok)
  {
    struct Op *t;
    int c;
    XPtrV ptns;
    XPinit(ptns, 16);
    t ← newtp(TPAT);
    c ← token(CONTIN | KEYWORD);      /* no ALIAS here */
    if (c ≠ '(') reject ← true;
    do {
      musthave(LWORD, 0);
      XPput(ptns, yylval.cp);
    } while ((c ← token(0)) ≡ '|');
    reject ← true;
    XPput(ptns, Λ);
    t→vars ← (char **) XPclose(ptns);
    musthave(')', 0);
    t→left ← c_list(true);
    if ((tpeek(CONTIN | KEYWORD | ALIAS)) ≠ endtok) musthave(BREAK, CONTIN | KEYWORD | ALIAS);
        /* POSIX requires the ;; */
    return (t);
  }
```

443. The list of TPAT nodes returned from *casepart* are attached in a linked list growing to the right, ending with the terminator "}" or "esac"—"{ ... }" is permitted in place of "in ... esac" as with for & select.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *caselist(void)
  {
    struct Op *t, *tl;
    int c;
    c ← token(CONTIN | KEYWORD | ALIAS);
    if (c ≡ IN) c ← ESAC;
    else if (c ≡ '{') c ← '}';
    else syntaxerr(Λ);
    t ← tl ← Λ;
    while ((tpeek(CONTIN | KEYWORD | ESACONLY)) ≠ c) {      /* no ALIAS here */
      struct Op *tc ← casepart(c);
      if (tl ≡ Λ) t ← tl ← tc, tl¬right ← Λ;
      else tl¬right ← tc, tl ← tc;
    }
    musthave(c, KEYWORD | ALIAS);
    return (t);
  }
```

444. Now that these specific cases have been laid out it should be easier to understand how this general case works. If the lexical analyser has a valid token but it doesn't know what it is then it returns LWORD and if it's a redirection then it returns REDIR. Each token that forms a command will be processed individually by this section.

```
⟨ Parse a regular command word 444 ⟩ ≡
  reject ← true;
  syniocf &= ∼(KEYWORD | ALIAS);
  t ← newtp(TCOM);
  t¬lineno ← source¬line;
  while (1) {
    cf ← (t¬u.evalflags ? ARRAYVAR : 0) | (XPsize(args) ≡ 0 ? ALIAS | VARASN : CMDWORD);
    switch (tpeek(cf)) {
    case REDIR: ⟨ Save a redirection in iops 445 ⟩
      break;
    case LWORD: reject ← false;
      ⟨ Save a token in args or vars 446 ⟩
      break;
    case '(':
        /* Check for "> foo (echo hi)", which AT&T ksh * allows (not POSIX, but not disallowed) */
      afree(t, ATEMP);
      ⟨ Check for a valid subshell or function 448 ⟩
      goto Leave;
    default: goto Leave;
    }
  }
```

This code is used in section 427.

445. A redirection is simply appended to *iops*. The details of how *syncio* does so are not important yet.

⟨ Save a redirection in *iops* 445 ⟩ ≡
```
if (iopn ≥ NUFILE) yyerror("too␣many␣redirections\n");
iops[iopn++] ← synio(cf);
```
This code is used in section 444.

446. This section's original claimed that "the $iopn \equiv 0$ and $XPsize(vars) \equiv 0$ are dubious but AT&T ksh acts this way". It does seem odd, TODO: come back to this after importing the evaluator and discovering what DOVACHECK is for. In any case, this checks whether a token is a variable assignment or not and assigns the string representing it to *args* or *vars*.

⟨ Save a token in *args* or *vars* 446 ⟩ ≡
```
if (iopn ≡ 0 ∧ XPsize(vars) ≡ 0 ∧ XPsize(args) ≡ 0 ∧ assign_command(ident))
  t→u.evalflags ← DOVACHECK;
if ((XPsize(args) ≡ 0 ∨ Flag(FKEYWORD)) ∧ is_wdvarassign(yylval.cp)) XPput(vars, yylval.cp);
else XPput(args, yylval.cp);
```
This code is used in section 444.

447. This kludge exists to take care of a Bourne sh/AT&T ksh oddity in which the arguments of an `alias`, `export`, `readonly` or `typeset` have no field splitting, file globbing, or (normal) tilde expansion done. AT&T ksh seems to do something similar to this:

```
> touch a=a; typeset a=[ab]; echo "$a"
a=[ab]
> x=typeset; $x a=[ab]; echo "$a"
a=a
>
```

⟨ syn.c 412 ⟩ +≡
```
static int assign_command(char *s)
{
  if (Flag(FPOSIX) ∨ ¬*s) return 0;
  return (strcmp(s, "alias") ≡ 0) ∨ (strcmp(s, "export") ≡ 0) ∨
      (strcmp(s, "readonly") ≡ 0) ∨ (strcmp(s, "typeset") ≡ 0);
}
```

448. If a "`(`" token is found and there are not yet any arguments or variable assignments then it indicates a new subshell, possibly after redirections such as "`> foo (subshell...)`" which AT&T ksh allows but POSIX has nothing to say about.

If exactly one argument and no assignments have been encountered so far then the `(` indicates a bourne-style function declaration "`foo() { command-list; }`".

⟨ Check for a valid subshell or function 448 ⟩ ≡
```
if (XPsize(args) ≡ 0 ∧ XPsize(vars) ≡ 0) {
  reject ← false;      /* TODO: unnecessary */
  goto Subshell;
}     /* else it must be a bourne-style function... */
if (iopn ≠ 0 ∨ XPsize(args) ≠ 1 ∨ XPsize(vars) ≠ 0) syntaxerr(Λ);
reject ← false;      /* TODO: unnecessary */
musthave(')', 0);
t ← function_body(XPptrv(args)[0], false);
```
This code is used in section 444.

449. Before *function_body* here is its other user, the Korn-style function constructor "`function`". It needs a name and a body.

⟨ Parse a command based on its first token 427 ⟩ +≡
```
case FUNCTION: musthave(LWORD, 0);
  t ← function_body(yylval.cp, true);
  break;
```

450. First check that the name is valid. POSIX and ksh93 say only to allow [a-zA-Z_0-9] but this allows more as old pdksh's have traditionally allowed more. The following were never allowed:

Λ|␣|*newline*|*tab*|$|'|"|\|`|(|)|&|||;|=|<|>

C_QUOTE covers all of these but = and adds [#|[|?|*].

Technically POSIX allows only compound statements for bourne-style functions (foo()) but sh and AT&T ksh allow any command, which "shouldn't break anything" (this has in fact been the norm for as long as I can remember and I had to check what a compound statement function is[1]). On the other hand for korn-style function foo AT&T ksh only accepts an open-brace. It's put back in the input stream so that *get_command* can parse it.

Throughout ksh the EF_FUNC_PARSE flag is toggled but apparently never checked in order to make it do anything.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *function_body(char *name, int ksh_func)
  {
    char *sname, *p;
    struct Op *t;
    int old_func_parse;
    sname ← wdstrip(name);
    for (p ← sname; *p; p++)
      if (ctype(*p, C_QUOTE) ∨ *p ≡ '=') yyerror("%s:␣invalid␣function␣name\n", sname);
    t ← newtp(TFUNCT);
    t→str ← sname;
    t→u.ksh_func ← ksh_func;
    t→lineno ← source→line;
    if (ksh_func) {
      musthave('{', CONTIN | KEYWORD | ALIAS);
      reject ← true;
    }
    old_func_parse ← genv→flags & EF_FUNC_PARSE;
    genv→flags |= EF_FUNC_PARSE;
    if ((t→left ← get_command(CONTIN)) ≡ Λ) {      /* No command probably means something like
          foo() followed by EOF or ; which is accepted by sh and ksh88 */
      t→left ← newtp(TCOM);      /* define a command manually */
      t→left→args ← areallocarray(Λ, 2, sizeof(char *), ATEMP);
      t→left→args[0] ← alloc(3, ATEMP);
      t→left→args[0][0] ← CHAR;
      t→left→args[0][1] ← ':';      /* pretend the function body is ":" so the function is printable */
      t→left→args[0][2] ← EOS;
      t→left→args[1] ← Λ;
      t→left→vars ← alloc(sizeof(char *), ATEMP);
      t→left→vars[0] ← Λ;
      t→left→lineno ← 1;
    }
    if (¬old_func_parse) genv→flags &= ∼EF_FUNC_PARSE;
    return t;
  }
```

[1] `foo() echo bar;`

451. Leave KEYWORD in *syniocf* (allow "`if (( 1 )) then ...`").

⟨ Parse a command based on its first token 427 ⟩ +≡

```
case MDPAREN:
  {
    static const char let_cmd[] ← {CHAR,'l',CHAR,'e',CHAR,'t',EOS};
    t ← newtp(TCOM);
    t¬lineno ← source¬line;
    reject ← false;
    XPput(args, wdcopy(let_cmd, ATEMP));
    musthave(LWORD, LETEXPR);
    XPput(args, yylval.cp);
    break;
  }
```

452. Leave KEYWORD in *syniocf* (allow "`if [[ -n 1 ]] then ...`").

⟨ Parse a command based on its first token 427 ⟩ +≡

```
case DBRACKET: t ← newtp(TDBRACKET);
  reject ← false;
  {
    Test_env te;
    te.flags ← TEF_DBRACKET;
    te.pos.av ← &args;
    te.isa ← dbtestp_isa;
    te.getopnd ← dbtestp_getopnd;
    te.eval ← dbtestp_eval;
    te.error ← dbtestp_error;
    test_parse(&te);
  }
  break;
```

453. I/O redirection.

Tests, subshells etc. may (usually do) have redirection tokens after the command—any that are present are saved in *iops*.

If at the end of parsing a command there were no redirections then *iops* can be feed otherwise they are saved in the syntax tree.

```
⟨ Process I/O redirection for irregular commands 453 ⟩ ≡
  while ((iop ← synio(syniocf)) ≠ Λ) {
    if (iopn ≥ NUFILE) yyerror("too␣many␣redirections\n");
    iops[iopn++] ← iop;
  }
  if (iopn ≡ 0) {
    afree(iops, ATEMP);
    t→ioact ← Λ;
  }
  else {
    iops[iopn++] ← Λ;
    iops ← areallocarray(iops, iopn, sizeof(struct ioword *), ATEMP);
    t→ioact ← iops;
  }
```

This code is used in section 426.

454. If the I/O redirection indicates a heredoc then parse its delimeter, or parse the file descriptor. An **ioword** object is filled in and returned.

```
⟨ syn.c 412 ⟩ +≡
  static struct ioword *synio(int cf)
  {
    struct ioword *iop;
    int ishere;
    if (tpeek(cf) ≠ REDIR) return Λ;
    reject ← false;
    iop ← yylval.iop;
    ishere ← (iop→flag & IOTYPE) ≡ IOHERE;
    musthave(LWORD, ishere ? HEREDELIM : 0);
    if (ishere) {
      iop→delim ← yylval.cp;
      if (*ident ≠ 0) iop→flag |= IOEVAL;      /* unquoted */
      if (herep ≥ &heres[HERES]) yyerror("too␣many␣<<'s\n");
      *herep++ ← iop;
    }
    else iop→name ← yylval.cp;
    return iop;
  }
```

455. Supporting Functions. Error messages are a type of support.

```
⟨ syn.c 412 ⟩ +≡
  static void syntaxerr(const char *what)
  {
    char redir[6];      /* "n<<-" is the longest redirection */
    const char *s;
    struct tokeninfo const *tt;
    int c;
    if (¬what) what ← "unexpected";
    reject ← true;
    c ← token(0);
  Again:
    switch (c) {⟨ Report (details of) a syntax error 456 ⟩}
    yyerror("syntax␣error:␣‘%s’␣%s\n", s, what);
  }
```

456. If the input ends prematurely and a closing delimiter (such as `)` or `fi`) is required that is reported instead.

```
⟨ Report (details of) a syntax error 456 ⟩ ≡
case 0:
  if (nesting.start_token) {
    c ← nesting.start_token;
    source→errline ← nesting.start_line;
    what ← "unmatched";
    goto Again;
  }
  yyerror("syntax␣error:␣unexpected␣EOF\n");      /* don't quote "EOF" like the main message */
```

See also sections 457 and 458.

This code is used in section 455.

457. In the case of an unexpected word or redirection, it is printed.

```
⟨ Report (details of) a syntax error 456 ⟩ +≡
case LWORD: s ← snptreef(Λ, 32, "%S", yylval.cp); break;
case REDIR: s ← snptreef(redir, sizeof (redir), "%R", yylval.iop); break;
```

458. Otherwise this thing; fill in *redir* with whatever detail is at fault.

```
⟨ Report (details of) a syntax error 456 ⟩ +≡
default:
  for (tt ← tokentab; tt→name; tt++)
    if (tt→val ≡ c) break;
  if (tt→name) s ← tt→name;
  else {
    if (c > 0 ∧ c < 256) {
      redir[0] ← c;
      redir[1] ← '\0';
    }
    else shf_snprintf(redir, sizeof (redir), "?%d", c);
    s ← redir;
  }
```

459. When a specific type of token is required this consumes one and raises an error if it's the wrong type.

```
⟨ syn.c 412 ⟩ +≡
  static void musthave(int c, int cf)
  {
    if ((token(cf)) ≠ c) syntaxerr(Λ);
  }
```

460. Construct a new *op* object with the given *left* and *right* links.

```
⟨ syn.c 412 ⟩ +≡
  static struct Op *block(int type, struct Op *t1, struct Op *t2, char **wp)
  {
    struct Op *t;
    t ← newtp(type);
    t→left ← t1;
    t→right ← t2;
    t→vars ← wp;
    return (t);
  }
```

461. Execution.

⟨ exec.c 461 ⟩ ≡

```
#include <sys/stat.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <paths.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
#include "c_test.h"
static int comexec(struct Op *, struct tbl *volatile , char **, int volatile, volatile int *);
static void scriptexec(struct Op *, char **);
static int call_builtin(struct tbl *, char **);
static int iosetup(struct ioword *, struct tbl *);
static int herein(const char *, int);
static char *do_selectargs(char **, bool);
static int dbteste_isa(Test_env *, Test_meta);
static const char *dbteste_getopnd(Test_env *, Test_op, int);
static int dbteste_eval(Test_env *, Test_op, const char *, const char *, int);
static void dbteste_error(Test_env *, int, const char *);
static char *select_fmt_entry(void *arg, int i, char *buf , int buflen);
static char *plain_fmt_entry(void *arg, int i, char *buf , int buflen);
```

See also sections 463, 523, 524, 536, 539, 540, 541, 542, 545, 553, 623, 631, 763, 765, 1333, 1334, 1335, 1336, 1337, 1338, 1341, 1342, and 1343.

462. ⟨ Shared function declarations 4 ⟩ +≡

```
int execute(struct Op *volatile , volatile int, volatile int *);
int shcomexec(char **);
struct tbl *findfunc(const char *, unsigned int, int);
int Define(const char *, struct Op *);      /* renamed for cosmetic reasons */
void builtin(const char *, int(*)(char **));
struct tbl *findcom(const char *, int);
void flushcom(int);
char *search(const char *, const char *, int, int *);
int search_access(const char *, int, int *);
int pr_menu(char *const *);
int pr_list(char *const *);
```

463. This routine to execute a syntax tree (which may be part of a larger syntax tree) is called from many places, including within itself. The main user entry point though is from *shell* in ⟨ Compile and interpret an expression 233 ⟩.

If *xerrok* points to an integer then it will be updated to inform recursive callers in `-e` mode that short-circuit `&&` or `||` shouldn't be treated as an error. If the `XERROK` flag is set then the error will be handled by the caller and not raised (which would unwind the stack) here.

The `PS4_SUBSTITUTE` macro started off its life togglable between *substitute*((*s*), 0) and (*s*). The toggle was lost in 2004 when V7-style `sh` support was abandoned in favour of POSIX `sh` but the comment remains.

```
#define PS4_SUBSTITUTE(s) substitute((s), 0)    /* Does $PS4 get parameter substitutions done? */
#define XEXEC BIT(0)    /* execute without forking */
#define XFORK BIT(1)    /* fork before executing */
#define XBGND BIT(2)    /* command & */
#define XPIPEI BIT(3)    /* input is pipe */
#define XPIPEO BIT(4)    /* output is pipe */
#define XPIPE (XPIPEI | XPIPEO)    /* member of pipe */
#define XXCOM BIT(5)    /* ` ... ` command[1] */
#define XPCLOSE BIT(6)    /* exchild: close(close_fd) in parent */
#define XCCLOSE BIT(7)    /* exchild: close(close_fd) in child */
#define XERROK BIT(8)    /* non-zero exit ok (for set -e) */
#define XCOPROC BIT(9)    /* starting a co-process */
#define XTIME BIT(10)    /* timing TCOM command */
```

```
⟨ exec.c 461 ⟩ +≡
  int execute(struct Op *volatile t, volatile int flags, volatile int *xerrok)
  {
    int i, dummy ← 0, save_xerrok ← 0;
    volatile int rv ← 0;
    int pv[2];
    char **volatile ap;
    char *s, *cp;
    struct ioword **iowp;
    struct tbl *tp ← Λ;
    if (t ≡ Λ) return 0;
    if (xerrok ≡ Λ) xerrok ← &dummy;    /* avoid repeatedly checking if xerrok ≠ Λ */
    if ((flags & XFORK) ∧ ¬(flags & XEXEC) ∧ t→type ≠ TPIPE)
      return exchild(t, flags & ∼XTIME, xerrok, −1);    /* run in sub-process */
    newenv(E_EXEC);
    if (trap) runtraps(0);
    ⟨ Evaluate command arguments 464 ⟩
    ⟨ Prepare I/O redirection for execution 466 ⟩
    switch (t→type) {⟨ Execute a syntax tree node and break or Break 506 ⟩}
  Break: exstat ← rv;
    quitenv(Λ);    /* restores I/O */
    if ((flags & XEXEC)) unwind (LEXIT);    /* exit child */
    if (rv ≠ 0 ∧ ¬(flags & XERROK) ∧ ¬*xerrok) {
      trapsig(SIGERR_);
      if (Flag(FERREXIT)) unwind (LERROR);
    }
    return rv;
  }
```

[1] TODO: Does `XXCOM` also include `$( ... )` (see also `JF_XXCOM`)?

464. Preparation began with this commented out block introduced (as a comment[1]) in the very first CVS revision ("update to pdksh-5.2.8") in August 1996 and it would be heartless to remove code which has sat there faithfully for 25 years waiting to be activated. "Is this the end of a pipeline? If so, we want to evaluate the command arguments".

```
⟨ Evaluate command arguments 464 ⟩ ≡
#if 0
  bool eval_done ← false;
  if ((flags & XFORK) ∧ ¬(flags & XEXEC) ∧ (flags & XPCLOSE)) {
    eval_done ← true;
    tp ← eval_execute_args(t, &ap);
  }
#endif
```

See also section 465.

This code is used in section 463.

465. In case no actual command is executed *subst_exstat*—the exit status of a command substitution—is reset to zero before argument expansion. Evaluation will result in an array of C-strings assigned to *ap* which will become the new command's *argv*. If at least one string was returned from *eval* then it is the command/alias/etc. name which *findcom* locates either on the filesystem or in memory.

TODO: Find out what is bizarre. That POSIX time has options? CVS log says "pdksh-5.2.14".

```
⟨ Evaluate command arguments 464 ⟩ +≡
  if (t→type ≡ TCOM) {
    subst_exstat ← 0;      /* Clear subst_exstat */
    current_lineno ← t→lineno;      /* for $LINENO */
      /* POSIX says expand command words first, then redirections, and assignments last.. */
    ap ← eval(t→args, t→u.evalflags | DOBLANK | DOGLOB | DOTILDE);
    if (flags & XTIME) timex_hook(t, &ap);      /* Allow option parsing (bizarre, but POSIX) */
    if (Flag(FXTRACE) ∧ ap[0]) {
      shf_fprintf(shl_out, "%s", PS4_SUBSTITUTE(str_val(global("PS4"))));
      for (i ← 0; ap[i]; i++) shf_fprintf(shl_out, "%s%s", ap[i], ap[i + 1] ? " " : "\n");
      shf_flush(shl_out);
    }
    if (ap[0]) tp ← findcom(ap[0], FC_BI | FC_FUNC);
  }
  flags &= ∼XTIME;
```

[1] Changed to preprocessor exclusion to avoid going insane trying to write C code in CWEB née TEX.

466. Loop over the I/O redirection instructions for this command that have been saved in *t→ioact* and perform them. The current file descriptor state is saved to be restored in *quitenv*.

```
⟨ Prepare I/O redirection for execution 466 ⟩ ≡
  if (t→ioact ≠ Λ ∨ t→type ≡ TPIPE ∨ t→type ≡ TCOPROC) {      /* initialize to not redirected */
    genv→savefd ← areallocarray(Λ, NUFILE, sizeof(short), ATEMP);
    memset(genv→savefd, 0, NUFILE * sizeof(short));
  }
  if (t→ioact ≠ Λ)
    for (iowp ← t→ioact; *iowp ≠ Λ; iowp++) {
      if (iosetup(*iowp, tp) < 0) {
        exstat ← rv ← 1;
        if (tp ∧ tp→type ≡ CSHELL ∧ (tp→flag & SPEC_BI))
            /* Redirection failures for special commands cause (non-interactive) shell to exit. */
          errorf(Λ);
        goto Break;      /* Deal with FERREXIT, quitenv, etc. */
      }
    }
```

This code is used in section 463.

467. Timing Commands.

```
⟨ Bourne commands 311 ⟩ +≡
  void timex_hook(struct Op *t, char **volatile *app)
  {
    char **wp ← *app;
    int optc;
    int i, j;
    Getopt opt;
    ksh_getopt_reset(&opt, 0);
    opt.optind ← 0;      /* start at the start */
    while ((optc ← ksh_getopt(wp, &opt, ":p")) ≠ −1)
      switch (optc) {
      case 'p': t→str[0] |= TF_POSIX;
        break;
      case '?': errorf("time: -%s unknown option", opt.optarg);
      case ':': errorf("time: -%s requires an argument", opt.optarg);
      }
    if (opt.optind ≠ 0) {      /* Copy command words down over options. */
      for (i ← 0; i < opt.optind; i++) afree(wp[i], ATEMP);
      for (i ← 0, j ← opt.optind; (wp[i] ← wp[j]); i++, j++) ;
    }
    if (¬wp[0]) t→str[0] |= TF_NOARGS;
    *app ← wp;
  }
```

468. There Two ways of getting CPU usage of a command: just use *t0* and *t1* (which will get CPU usage from other jobs that finish while we are executing *t→left*) or get the CPU usage of *t→left*. AT&T ksh does the former while pdksh tries to do the later (the *j_usrtime* hack doesn't really work as it only counts the last job).

```
#define TF_NOARGS BIT(0)
#define TF_NOREAL BIT(1)    /* don't report real time */
#define TF_POSIX BIT(2)     /* report in POSIX format */
```

⟨ Bourne commands 311 ⟩ +≡

```
int timex(struct Op *t, int f, volatile int *xerrok)
{
  int rv ← 0;
  struct rusage ru0, ru1, cru0, cru1;
  struct timeval usrtime, systime;
  struct timespec ts0, ts1, ts2;
  int tf ← 0;
  extern struct timeval j_usrtime, j_systime;    /* computed by j_wait */
  clock_gettime(CLOCK_MONOTONIC, &ts0);
  getrusage(RUSAGE_SELF, &ru0);
  getrusage(RUSAGE_CHILDREN, &cru0);
  if (t→left) {
    timerclear(&j_usrtime);
    timerclear(&j_systime);
    rv ← execute(t→left, f | XTIME, xerrok);
    if (t→left→type ≡ TCOM) tf |= t→left→str[0];
    clock_gettime(CLOCK_MONOTONIC, &ts1);
    getrusage(RUSAGE_SELF, &ru1);
    getrusage(RUSAGE_CHILDREN, &cru1);
  }
  else tf ← TF_NOARGS;
  if (tf & TF_NOARGS) {    /* ksh93 - report shell times (shell & kids) */
    tf |= TF_NOREAL;
    timeradd(&ru0.ru_utime, &cru0.ru_utime, &usrtime);
    timeradd(&ru0.ru_stime, &cru0.ru_stime, &systime);
  }
  else {
    timersub(&ru1.ru_utime, &ru0.ru_utime, &usrtime);
    timeradd(&usrtime, &j_usrtime, &usrtime);
    timersub(&ru1.ru_stime, &ru0.ru_stime, &systime);
    timeradd(&systime, &j_systime, &systime);
  }
  if (¬(tf & TF_NOREAL)) {
    timespecsub(&ts1, &ts0, &ts2);
    if (tf & TF_POSIX) p_ts(shl_out, 1, &ts2, 5, "real␣", "\n");
    else p_ts(shl_out, 0, &ts2, 5, Λ, "␣real␣");
  }
  if (tf & TF_POSIX) p_tv(shl_out, 1, &usrtime, 5, "user␣", "\n");
  else p_tv(shl_out, 0, &usrtime, 5, Λ, "␣user␣");
  if (tf & TF_POSIX) p_tv(shl_out, 1, &systime, 5, "sys␣␣", "\n");
  else p_tv(shl_out, 0, &systime, 5, Λ, "␣system\n");
  shf_flush(shl_out);
  return rv;
```

```
}
```

469. Syntax Tree. Should be moved into "Source Input" methinks and parts merged with "Formatted Output".

```
#define INDENT 4
```

⟨ tree.c 469 ⟩ ≡

```
#include <string.h>
#include "sh.h"
  static void ptree(struct Op *, int, struct Shf *);
  static void pioact(struct Shf *, int, struct ioword *);
  static void tputC(int, struct Shf *);
  static void tputS(char *, struct Shf *);
  static void vfptreef(struct Shf *, int, const char *, va_list);
  static struct ioword **iocopy(struct ioword **, Area *);
  static void iofree(struct ioword **, Area *);
```

See also sections 471, 478, 479, 480, 487, 488, 489, 499, 500, 501, 502, 503, 504, and 505.

470. These preprocessor constants are not used anywhere and should be removed if it becomes clear that they are a leftover and not part of future plans. They came with this comment:

The arguments of `[[ ... ]]` expressions are kept in *t→args* and flags indicating how the arguments have been munged are kept in *t→vars*. The contents of *t→vars* are stuffed strings (so they can be treated like all other *t→vars*) in which the second character is the one that is examined. The `DB_*` defines are the values for these second characters.

```
#define DB_NORM 1    /* normal argument */
#define DB_OR 2    /* || → -o conversion */
#define DB_AND 3    /* && → -a conversion */
#define DB_BE 4    /* an inserted -BE */
#define DB_PAT 5    /* a pattern argument */
```

⟨ tree.h 470 ⟩ ≡

```
  void fptreef(struct Shf *, int, const char *, ...);
  char *snptreef(char *, int, const char *, ...);
  struct Op *tcopy(struct Op *, Area *);
  char *wdcopy(const char *, Area *);
  char *wdscan(const char *, int);
  char *wdstrip(const char *);
  void tfree(struct Op *, Area *);
```

471. Print a syntax tree.

```
⟨ tree.c 469 ⟩ +≡
  static void ptree(struct Op *t, int indent, struct Shf *shf)
  {
    char **w;
    struct ioword **ioact;
    struct Op *t1;
  Chain:
    if (t ≡ Λ) return;
    switch (t→type) {⟨ Print a syntax tree node 472 ⟩}
    if ((ioact ← t→ioact) ≠ Λ) {
      int need_nl ← 0;
      while (*ioact ≠ Λ) pioact(shf, indent, *ioact++);
      for (ioact ← t→ioact; *ioact ≠ Λ; ) {      /* Print here documents after everything else */
        struct ioword *iop ← *ioact++;
        if ((iop→flag & IOTYPE) ≡ IOHERE ∧ iop→heredoc) {      /* heredoc is 0 when tracing (set -x) */
          tputc('\n', shf);
          shf_puts(iop→heredoc, shf);
          fptreef(shf, indent, "%s", evalstr(iop→delim, 0));
          need_nl ← 1;
        }
      }
      if (need_nl) tputc('\n', shf);      /* Necessary but often leads to an extra blank line */
    }
  }
```

472. ⟨ Print a syntax tree node 472 ⟩ ≡

```
case TCOM:
  if (t→vars) for (w ← t→vars; *w ≠ Λ; ) fptreef(shf, indent, "%S␣", *w++);
  else fptreef(shf, indent, "#no-vars#␣");
  if (t→args) for (w ← t→args; *w ≠ Λ; ) fptreef(shf, indent, "%S␣", *w++);
  else fptreef(shf, indent, "#no-args#␣");
  break;
case TEXEC: t ← t→left; goto Chain;
case TPAREN: fptreef(shf, indent + 2, "(␣%T)␣", t→left); break;
case TPIPE: fptreef(shf, indent, "%T|␣", t→left); t ← t→right; goto Chain;
case TLIST: fptreef(shf, indent, "%T%;", t→left); t ← t→right; goto Chain;
case TOR: case TAND: fptreef(shf, indent, "%T%s␣%T", t→left, (t→type ≡ TOR) ? "||" : "&&", t→right); break;
case TBANG: fptreef(shf, indent, "!␣"); t ← t→right; goto Chain;
case TDBRACKET:
  {
    int i;
    fptreef(shf, indent, "[[");
    for (i ← 0; t→args[i]; i++) fptreef(shf, indent, "␣%S", t→args[i]);
    fptreef(shf, indent, "␣]]␣");
    break;
  }
```

See also sections 473, 474, 475, 476, and 477.

This code is used in section 471.

473. `select` and `for` have the same syntax.

```
⟨ Print a syntax tree node 472 ⟩ +≡
case TSELECT: fptreef(shf, indent, "select␣%s␣", t→str);      /* FALLTHROUGH */
case TFOR:
  if (t→type ≡ TFOR) fptreef(shf, indent, "for␣%s␣", t→str);
  if (t→vars ≠ Λ) {
    fptreef(shf, indent, "in␣");
    for (w ← t→vars; *w; ) fptreef(shf, indent, "%S␣", *w++);
    fptreef(shf, indent, "%;");
  }
  fptreef(shf, indent + INDENT, "do%N%T", t→left);
  fptreef(shf, indent, "%;done␣");
  break;
```

474. `case`

```
⟨ Print a syntax tree node 472 ⟩ +≡
case TCASE: fptreef(shf, indent, "case␣%S␣in", t→str);
  for (t1 ← t→left; t1 ≠ Λ; t1 ← t1→right) {
    fptreef(shf, indent, "%N(");
    for (w ← t1→vars; *w ≠ Λ; w++) fptreef(shf, indent, "%S%c", *w, (w[1] ≠ Λ) ? '|' : ')');
    fptreef(shf, indent + INDENT, "%;%T%N;;", t1→left);
  }
  fptreef(shf, indent, "%Nesac␣");
  break;
```

475. `if` and `elif` are almost identical.

```
⟨ Print a syntax tree node 472 ⟩ +≡
case TIF: case TELIF:      /* 3 == strlen("if ") */
  fptreef(shf, indent + 3, "if␣%T", t→left);
  for ( ; ; ) {
    t ← t→right;
    if (t→left ≠ Λ) {
      fptreef(shf, indent, "%;");
      fptreef(shf, indent + INDENT, "then%N%T", t→left);
    }
    if (t→right ≡ Λ ∨ t→right→type ≠ TELIF) break;
    t ← t→right;
    fptreef(shf, indent, "%;");      /* 5 == strlen("elif ") */
    fptreef(shf, indent + 5, "elif␣%T", t→left);
  }
  if (t→right ≠ Λ) {
    fptreef(shf, indent, "%;");
    fptreef(shf, indent + INDENT, "else%;%T", t→right);
  }
  fptreef(shf, indent, "%;fi␣");
  break;
```

476. `while` and `until` are the same.

```
⟨ Print a syntax tree node 472 ⟩ +≡
case TWHILE: case TUNTIL:      /* 6 == strlen("while"/"until") */
  fptreef(shf, indent + 6, "%s␣%T", (t→type ≡ TWHILE) ? "while" : "until", t→left);
  fptreef(shf, indent, "%;do");
  fptreef(shf, indent + INDENT, "%;%T", t→right);
  fptreef(shf, indent, "%;done␣");
  break;
```

477. These are all variants on printing a function.

```
⟨ Print a syntax tree node 472 ⟩ +≡
case TBRACE: fptreef(shf, indent + INDENT, "{%;%T", t→left);
  fptreef(shf, indent, "%;}␣");
  break;
case TCOPROC: fptreef(shf, indent, "%T|&␣", t→left); break;
case TASYNC: fptreef(shf, indent, "%T&␣", t→left); break;
case TFUNCT: fptreef(shf, indent, t→u.ksh_func ? "function␣%s␣%T" : "%s()␣%T", t→str, t→left); break;
case TTIME: fptreef(shf, indent, "time␣%T", t→left); break;
default: fptreef(shf, indent, "<botch>"); break;
```

478. Before a command the I/O redirections and other actions are printed.

```
⟨ tree.c 469 ⟩ +≡
  static void pioact(struct Shf *shf, int indent, struct ioword *iop)
  {
    int flag ← iop→flag;
    int type ← flag & IOTYPE;
    int expected;
    expected ← (type ≡ IOREAD ∨ type ≡ IORDWR ∨ type ≡ IOHERE) ? 0 :
        (type ≡ IOCAT ∨ type ≡ IOWRITE) ? 1 :
        (type ≡ IODUP ∧ (iop→unit ≡ ¬(flag & IORDUP))) ? iop→unit :
        iop→unit + 1;
    if (iop→unit ≠ expected) tputc('0' + iop→unit, shf);
    switch (type) {
    case IOREAD: fptreef(shf, indent, "<␣"); break;
    case IOHERE:
      if (flag & IOSKIP) fptreef(shf, indent, "<<-␣");
      else fptreef(shf, indent, "<<␣");
      break;
    case IOCAT: fptreef(shf, indent, ">>␣"); break;
    case IOWRITE:
      if (flag & IOCLOB) fptreef(shf, indent, ">|␣");
      else fptreef(shf, indent, ">␣");
      break;
    case IORDWR: fptreef(shf, indent, "<>␣"); break;
    case IODUP:
      if (flag & IORDUP) fptreef(shf, indent, "<&");
      else fptreef(shf, indent, ">&");
      break;
    }     /* name & delim are 0 when printing syntax errors */
    if (type ≡ IOHERE) {
      if (iop→delim) fptreef(shf, indent, "%S␣", iop→delim);
    }
    else if (iop→name) fptreef(shf, indent, (iop→flag & IONAMEXP) ? "%s␣" : "%S␣", iop→name);
  }
```

479. Reinventing more of stdio.

```
#define tputc(c, shf) shf_putchar(c, shf);
⟨ tree.c 469 ⟩ +≡
  static void tputC(int c, struct Shf *shf)
  {
    if ((c & 0x60) ≡ 0) {      /* C0|C1 */
      tputc((c & 0x80) ? '$' : '^', shf);
      tputc(((c & 0x7F) | 0x40), shf);
    }
    else if ((c & 0x7F) ≡ 0x7F) {      /* DEL */
      tputc((c & 0x80) ? '$' : '^', shf);
      tputc('?', shf);
    }
    else tputc(c, shf);
  }
```

480. Problems:

* `` ` ... ` `` → `$( ... )`
* `'foo'` → `"foo"`

Could change the encoding to (TODO):

* `OQUOTE ["']` ... `CQUOTE ["']`
* `COMSUB [(`]` ... `'\0'` (handle `$`, `` ` ``, `\` and maybe `"` in `` ` ... ` `` case)

```
⟨ tree.c 469 ⟩ +≡
  static void tputS(char *wp, struct Shf *shf)
  {
    int c, quoted ← 0;
    while (1)
      switch ((c ← *wp++)) {
      case EOS: return;
        ⟨ Put a lexicographic string 481 ⟩
      }
  }
```

481. ⟨ Put a lexicographic string 481 ⟩ ≡

```
case CHAR: tputC(*wp++, shf); break;
```

See also sections 482, 483, 484, 485, and 486.

This code is used in section 480.

482. Quoted character.

```
⟨ Put a lexicographic string 481 ⟩ +≡
case QCHAR: c ← *wp++;
  if (¬quoted ∨ (c ≡ '"' ∨ c ≡ '`' ∨ c ≡ '$')) tputc('\\', shf);
  tputC(c, shf);
  break;
```

483. Command and expression substitution.

```
⟨ Put a lexicographic string 481 ⟩ +≡
case COMSUB: tputc('$', shf);
  tputc('(', shf);
  while (*wp ≠ 0) tputC(*wp++, shf);
  tputc(')', shf);
  wp++;
  break;
case EXPRSUB: tputc('$', shf);
  tputc('(', shf);
  tputc('(', shf);
  while (*wp ≠ 0) tputC(*wp++, shf);
  tputc(')', shf);
  tputc(')', shf);
  wp++;
  break;
```

484. Quoted string.

```
⟨ Put a lexicographic string 481 ⟩ +≡
case OQUOTE: quoted ← 1; tputc('"', shf); break;
case CQUOTE: quoted ← 0; tputc('"', shf); break;
```

485. Variable & command substitution.

```
⟨ Put a lexicographic string 481 ⟩ +≡
case OSUBST:
  tputc('$', shf);
  if (*wp++ ≡ '{') tputc('{', shf);
  while ((c ← *wp++) ≠ 0) tputC(c, shf);
  break;
case CSUBST:
  if (*wp++ ≡ '}') tputc('}', shf);
  break;
```

486. Pattern matching.

```
⟨ Put a lexicographic string 481 ⟩ +≡
case OPAT: tputc(*wp++, shf); tputc('(', shf); break;
case SPAT: tputc('|', shf); break;
case CPAT: tputc(')', shf); break;
```

487. Interfaces to the above including variadic wrappers.

```
⟨ tree.c 469 ⟩ +≡
  void fptreef(struct Shf *shf, int indent, const char *fmt, ...)
  {
    va_list va;
    va_start(va, fmt);
    vfptreef(shf, indent, fmt, va);
    va_end(va);
  }
```

488. ⟨ tree.c 469 ⟩ +≡

```
  char *snptreef(char *s, int n, const char *fmt, ...)
  {
    va_list va;
    struct Shf shf;
    shf_sopen(s, n, SHF_WR | (s ? 0 : SHF_DYNAMIC), &shf);
    va_start(va, fmt);
    vfptreef(&shf, 0, fmt, va);
    va_end(va);
    return shf_sclose(&shf);      /* '\0'-terminates */
  }
```

489. ⟨ tree.c 469 ⟩ +≡

```
static void vfptreef(struct Shf *shf, int indent, const char *fmt, va_list va)
{
  int c;
  while ((c ← *fmt++)) {
    if (c ≡ '%') {
      int64_t n;
      char *p;
      int neg;
      switch ((c ← *fmt++)) {⟨ Format a tree part 490 ⟩}
    }
    else tputc(c, shf);
  }
}
```

490. %c: Character argument.

⟨ Format a tree part 490 ⟩ ≡

```
case 'c': tputc(va_arg(va, int), shf);
  break;
```

See also sections 491, 492, 493, 494, 495, 496, 497, and 498.

This code is used in section 489.

491. %d: Signed decimal argument. n (and the first argument to *u64ton*) must be wider than the maximum width of the argument. In this case the argument is an **int** which may also be 64 bits itself but should be clamped elsewhere to between INT_MIN–INT_MAX.

⟨ Format a tree part 490 ⟩ +≡

```
case 'd': n ← va_arg(va, int);
  neg ← n < 0;
  p ← u64ton(neg ? −n : n, 10);
  if (neg) *−−p ← '-';
  while (*p) tputc(*p++, shf);
  break;
```

492. %s: Pointer to a '\0'-terminated string.

⟨ Format a tree part 490 ⟩ +≡

```
case 's': p ← va_arg(va, char *);
  while (*p) tputc(*p++, shf);
  break;
```

493. %S: Pointer to a lexicographic word/string argument.

⟨ Format a tree part 490 ⟩ +≡

```
case 'S': p ← va_arg(va, char *);
  tputS(p, shf);
  break;
```

494. %u: Unsigned decimal argument.

⟨ Format a tree part 490 ⟩ +≡

```
case 'u': p ← u64ton(va_arg(va, unsigned int), 10);
  while (*p) tputc(*p++, shf);
  break;
```

495. `%T`: An opcode tree argument (**Op** object).

```
⟨ Format a tree part 490 ⟩ +≡
case 'T': ptree(va_arg(va, struct Op *), indent, shf);
  break;
```

496. A newline or "`;`" (`%;`) or a newline or "␣" (`%N`).

```
⟨ Format a tree part 490 ⟩ +≡
case ';': case 'N':
  if (shf→flags & SHF_STRING) {
    if (c ≡ ';') tputc(';', shf);
    tputc('␣', shf);
  }
  else {
    int i;
    tputc('\n', shf);
    for (i ← indent; i ≥ 8; i −= 8) tputc('\t', shf);
    for ( ; i > 0; −−i) tputc('␣', shf);
  }
  break;
```

497. `%R`: I/O redirection argument.

```
⟨ Format a tree part 490 ⟩ +≡
case 'R': pioact(shf, indent, va_arg(va, struct ioword *));
  break;
```

498. Everything else goes as-is.

```
⟨ Format a tree part 490 ⟩ +≡
default: tputc(c, shf);
  break;
```

499. Copy a syntax tree. Simple rec-descent algorithm.

```
⟨ tree.c 469 ⟩ +≡
  struct Op *tcopy(struct Op *t, Area *ap)
  {
    struct Op *r;
    char **tw, **rw;
    if (t ≡ Λ) return Λ;
    r ← alloc(sizeof(struct Op), ap);
    r→type ← t→type;
    r→u.evalflags ← t→u.evalflags;
    r→str ← t→type ≡ TCASE ? wdcopy(t→str, ap) : str_save(t→str, ap);
    if (t→vars ≡ Λ) r→vars ← Λ;
    else {
      for (tw ← t→vars; *tw++ ≠ Λ; ) ;
      rw ← r→vars ← areallocarray(Λ, tw − t→vars + 1, sizeof (*tw), ap);
      for (tw ← t→vars; *tw ≠ Λ; ) *rw++ ← wdcopy(*tw++, ap);
      *rw ← Λ;
    }
    if (t→args ≡ Λ) r→args ← Λ;
    else {
      for (tw ← t→args; *tw++ ≠ Λ; ) ;
      rw ← r→args ← areallocarray(Λ, tw − t→args + 1, sizeof (*tw), ap);
      for (tw ← t→args; *tw ≠ Λ; ) *rw++ ← wdcopy(*tw++, ap);
      *rw ← Λ;
    }
    r→ioact ← (t→ioact ≡ Λ) ? Λ : iocopy(t→ioact, ap);
    r→left ← tcopy(t→left, ap);
    r→right ← tcopy(t→right, ap);
    r→lineno ← t→lineno;
    return r;
  }
```

500. And its free-er.

```
⟨ tree.c 469 ⟩ +≡
  void tfree(struct Op *t, Area *ap)
  {
    char **w;
    if (t ≡ Λ) return;
    afree(t→str, ap);
    if (t→vars ≠ Λ) {
      for (w ← t→vars; *w ≠ Λ; w++) afree(*w, ap);
      afree(t→vars, ap);
    }
    if (t→args ≠ Λ) {
      for (w ← t→args; *w ≠ Λ; w++) afree(*w, ap);
      afree(t→args, ap);
    }
    if (t→ioact ≠ Λ) iofree(t→ioact, ap);
    tfree(t→left, ap);
    tfree(t→right, ap);
    afree(t, ap);
  }
```

501. Copying I/O redirection.

```
⟨ tree.c 469 ⟩ +≡
  static struct ioword **iocopy(struct ioword **iow, Area *ap)
  {
    struct ioword **ior;
    int i;
    for (ior ← iow; *ior++ ≠ Λ; ) ;
    ior ← areallocarray(Λ, ior − iow + 1, sizeof (*ior), ap);
    for (i ← 0; iow[i] ≠ Λ; i++) {
      struct ioword *p, *q;
      p ← iow[i];
      q ← alloc(sizeof (*p), ap);
      ior[i] ← q;
      *q ← *p;
      if (p→name ≠ Λ) q→name ← wdcopy(p→name, ap);
      if (p→delim ≠ Λ) q→delim ← wdcopy(p→delim, ap);
      if (p→heredoc ≠ Λ) q→heredoc ← str_save(p→heredoc, ap);
    }
    ior[i] ← Λ;
    return ior;
  }
```

502. And *its* free-er.

```
⟨ tree.c 469 ⟩ +≡
  static void iofree(struct ioword **iow, Area *ap)
  {
    struct ioword **iop;
    struct ioword *p;
    for (iop ← iow; (p ← *iop++) ≠ Λ; ) {
      afree(p→name, ap);
      afree(p→delim, ap);
      afree(p→heredoc, ap);
      afree(p, ap);
    }
    afree(iow, ap);
  }
```

503. Copying character strings, has users other than *tcopy*.

```
⟨ tree.c 469 ⟩ +≡
  char *wdcopy(const char *wp, Area *ap)
  {
    size_t len ← wdscan(wp, EOS) − wp;
    return memcpy(alloc(len, ap), wp, len);
  }
```

504. Scan a string looking for a character type. Not fully generic.

```
⟨ tree.c 469 ⟩ +≡
  char *wdscan(const char *wp, int c)
  {
    int nest ← 0;
    while (1)
      switch (*wp++) {
      case EOS: return (char *) wp;
      case CHAR: case QCHAR: wp++; break;
      case COMSUB: case EXPRSUB: while (*wp++ ≠ 0) ; break;
      case OQUOTE: case CQUOTE: break;
      case OSUBST: nest++; while (*wp++ ≠ '\0') ; break;
      case CSUBST:
        wp++;
        if (c ≡ CSUBST ∧ nest ≡ 0) return (char *) wp;
        nest−−;
        break;
      case OPAT: nest++; wp++; break;
      case SPAT: case CPAT:
        if (c ≡ wp[−1] ∧ nest ≡ 0) return (char *) wp;
        if (wp[−1] ≡ CPAT) nest−−;
        break;
      default: internal_warningf("%s: unknown char 0x%x (carrying on)", __func__, wp[−1]);
      }
  }
```

505. Not even used by *tcopy*, copy a string with markup and quot characters stripped. Problems:

* `` ` ... ` `` → `$( ... )`
* `x${foo:-"bar"}` → `x${foo:-bar}`
* `x${foo:-'bar'}` → `x${foo:-bar}`

⟨ tree.c 469 ⟩ +≡

```
char *wdstrip(const char *wp)
{
  struct Shf shf;
  int c;
  shf_sopen(Λ, 32, SHF_WR | SHF_DYNAMIC, &shf);
  while (1)
    switch ((c ← *wp++)) {
    case EOS: return shf_sclose(&shf);      /* '\0'-terminates */
    case CHAR: case QCHAR: shf_putchar(*wp++, &shf);
      break;
    case COMSUB: shf_putchar('$', &shf);
      shf_putchar('(', &shf);
      while (*wp ≠ 0) shf_putchar(*wp++, &shf);
      shf_putchar(')', &shf);
      break;
    case EXPRSUB: shf_putchar('$', &shf);
      shf_putchar('(', &shf);
      shf_putchar('(', &shf);
      while (*wp ≠ 0) shf_putchar(*wp++, &shf);
      shf_putchar(')', &shf);
      shf_putchar(')', &shf);
      break;
    case OQUOTE: break;
    case CQUOTE: break;
    case OSUBST: shf_putchar('$', &shf);
      if (*wp++ ≡ '{') shf_putchar('{', &shf);
      while ((c ← *wp++) ≠ 0) shf_putchar(c, &shf);
      break;
    case CSUBST:
      if (*wp++ ≡ '}') shf_putchar('}', &shf);
      break;
    case OPAT: shf_putchar(*wp++, &shf);
      shf_putchar('(', &shf);
      break;
    case SPAT: shf_putchar('|', &shf);
      break;
    case CPAT: shf_putchar(')', &shf);
      break;
    }
}
```

506. TCOM is the most complex case and the work is offloaded to *comexec* (TODO: link to it). TPAREN is the simplest which recurses straight back into *execute* but telling it to fork a subshell first.

```
⟨ Execute a syntax tree node and break or Break 506 ⟩ ≡
case TCOM: rv ← comexec(t, tp, ap, flags, xerrok); break;
case TPAREN: rv ← execute(t→left, flags | XFORK, xerrok); break;
case TPIPE: ⟨ Execute a TPIPE node and break 507 ⟩
case TLIST: ⟨ Execute a TLIST node and break 508 ⟩
case TCOPROC: ⟨ Execute a TCOPROC node and break 509 ⟩
case TASYNC: ⟨ Execute a TASYNC node and break 510 ⟩
case TOR: case TAND: ⟨ Execute a TOR/TAND node and break 511 ⟩
case TBANG: ⟨ Execute a TBANG node and break 512 ⟩
case TDBRACKET: ⟨ Execute a TDBRACKET node and break 513 ⟩
case TFOR: case TSELECT: ⟨ Execute a TFOR/TSELECT node and break 514 ⟩
case TWHILE: case TUNTIL: ⟨ Execute a TWHILE/TUNTIL node and break 518 ⟩
case TIF: case TELIF: ⟨ Execute a TIF/TELIF node and break 519 ⟩
case TCASE: ⟨ Execute a TCASE node and break 520 ⟩
case TBRACE: rv ← execute(t→left, flags & XERROK, xerrok); break;
case TFUNCT: rv ← Define(t→str, t); break;
case TTIME: rv ← timex(t, flags & ~XEXEC, xerrok); break;      /* Ensure execute doesn't exit */
                                                               /* (allows "ls -l | time grep foo") */
case TEXEC: ⟨ Execute a TEXEC node and break 521 ⟩
```

This code is used in section 463.

507. "⟨t→left⟩ | ⟨(t ← t→right)→left⟩ | ⟨(t ← t→right)→left⟩ | ...".

/* Let *exchild*( ) close *pv*[0] in child (if this isn't done, commands like `(: ; cat /etc/termcap) |sleep 1` will hang forever). */

File descriptors and forks. I will figure this extreme unixing out in the morning over coffee rather than in the evening over wine.

```
⟨ Execute a TPIPE node and break 507 ⟩ ≡
  flags |= XFORK;
  flags &= ~XEXEC;
  genv→savefd[0] ← savefd(0);
  genv→savefd[1] ← savefd(1);
  while (t→type ≡ TPIPE) {
    openpipe(pv);
    (void) ksh_dup2(pv[1], 1, false);      /* stdout of curr */
    exchild(t→left, flags | XPIPEO | XCCLOSE, Λ, pv[0]);
    (void) ksh_dup2(pv[0], 0, false);      /* stdin of next */
    closepipe(pv);
    flags |= XPIPEI;
    t ← t→right;
  }
  restfd(1, genv→savefd[1]);      /* stdout of last */
  genv→savefd[1] ← 0;      /* no need to re-restore this */
  i ← exchild(t, flags | XPCLOSE, xerrok, 0);      /* Let exchild close 0 in parent, after fork, before wait */
  if (¬(flags & XBGND) ∧ ¬(flags & XXCOM)) rv ← i;
  break;
```

This code is used in section 506.

508. TLIST calls into *execute* over and over again until there are no TLIST nodes left.

```
⟨ Execute a TLIST node and break 508 ⟩ ≡
  while (t→type ≡ TLIST) {
    execute(t→left, flags & XERROK, Λ);
    t ← t→right;
  }
  rv ← execute(t, flags & XERROK, xerrok);
  break;
```

This code is used in section 506.

509. Block SIGCHLD as we are using things changed in the signal handler.
All of this work is to set up pipe FDs then the process part is done by *exchild*.

```
⟨ Execute a TCOPROC node and break 509 ⟩ ≡
  {
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);      /* block SIGCHLD for the duration */
    genv→type ← E_ERRH;
    i ← sigsetjmp(genv→jbuf, 0);
    if (i) {
      sigprocmask(SIG_SETMASK, &omask, Λ);
      quitenv(Λ);
      unwind (i);
      ;      /* NOTREACHED */
    }
    if (coproc.job ∧ coproc.write ≥ 0)      /* Already have a (live) co-process? */
      errorf("coprocess␣already␣exists");
    coproc_cleanup(true);      /* Can we re-use the existing co-process pipe? */
    genv→savefd[0] ← savefd(0);
    genv→savefd[1] ← savefd(1);      /* do this before opening pipes, in case these fail */
    openpipe(pv);
    if (pv[0] ≠ 0) {
      ksh_dup2(pv[0], 0, false);
      close(pv[0]);
    }
    coproc.write ← pv[1];
    coproc.job ← Λ;
    if (coproc.readw ≥ 0) ksh_dup2(coproc.readw, 1, false);
    else {
      openpipe(pv);
      coproc.read ← pv[0];
      ksh_dup2(pv[1], 1, false);
      coproc.readw ← pv[1];      /* closed before first read */
      coproc.njobs ← 0;
      ++coproc.id;      /* create new coprocess id */
    }
    sigprocmask(SIG_SETMASK, &omask, Λ);
    genv→type ← E_EXEC;      /* no more need for error handler */
    flags &= ∼XEXEC;
    exchild(t→left, flags | XBGND | XFORK | XCOPROC | XCCLOSE, Λ, coproc.readw);      /* exchild closes
        coproc.* in child after fork, will also increment coproc.njobs when the job is actually created. */
    break;
  }
```

This code is used in section 506.

510. This case came with a note that is was possibly non-optimal—"`(foo &)`", forks for `()`, forks again for async" and that it could be optimised to "`foo &`". This is an excellent example of Knuth's 97%.

```
⟨ Execute a TASYNC node and break 510 ⟩ ≡
  rv ← execute(t→left, (flags & ∼XEXEC) | XBGND | XFORK, xerrok);
  break;
```

This code is used in section 506.

511. Calls *execute* on *t→left* then on *t→right* depending on whether or not *t→left* failed and whether or not is should have.

```
⟨ Execute a TOR/TAND node and break 511 ⟩ ≡
  rv ← execute(t→left, XERROK, xerrok);
  if ((rv ≡ 0) ≡ (t→type ≡ TAND)) rv ← execute(t→right, flags & XERROK, xerrok);
  else {
    flags |= XERROK;
    *xerrok ← 1;
  }
  break;
```

This code is used in section 506.

512. Recurse back into *execute* and negate the result. Makes sure failure is not considered an error.

```
⟨ Execute a TBANG node and break 512 ⟩ ≡
  rv ← ¬execute(t→right, XERROK, xerrok);
  flags |= XERROK;
  *xerrok ← 1;
  break;
```

This code is used in section 506.

513. Prepare a double-bracket test environment and run the test described in *t→args*.

```
⟨ Execute a TDBRACKET node and break 513 ⟩ ≡
  {
    Test_env te;
    te.flags ← TEF_DBRACKET;
    te.pos.wp ← t→args;
    te.isa ← dbteste_isa;
    te.getopnd ← dbteste_getopnd;
    te.eval ← dbteste_eval;
    te.error ← dbteste_error;
    rv ← test_parse(&te);
    break;
  }
```

This code is used in section 506.

514. *ap* is set to the evaluation of *t→vars* if there is one, or the second element of the environment's *argv* (the first is the function or script name).

```
⟨ Execute a TFOR/TSELECT node and break 514 ⟩ ≡
  {
    volatile bool is_first ← true;
    ap ← (t→vars ≠ Λ) ? eval(t→vars, DOBLANK | DOGLOB | DOTILDE) : genv→loc→argv + 1;
    ⟨ Establish a return point for break/continue 515 ⟩
    if (t→type ≡ TFOR) {⟨ Execute a TFOR node 516 ⟩}
    else {⟨ Execute a TSELECT node 517 ⟩}
  }
  break;
```

This code is used in section 506.

515. Ending a loop early with the shell commands `break` or `continue` will long jump with `LBREAK` or `LCONTIN` respectively. `EF_BRKCONT_PASS` is set in *c_brkcont* to indicate that unwinding the stack should carry on.

```
⟨ Establish a return point for break/continue 515 ⟩ ≡
  genv→type ← E_LOOP;
  while (1) {
    i ← sigsetjmp(genv→jbuf, 0);
    if (¬i) break;
    if ((genv→flags & EF_BRKCONT_PASS) ∨ (i ≠ LBREAK ∧ i ≠ LCONTIN)) {
      quitenv(Λ);
      unwind (i);
    }
    else if (i ≡ LBREAK) {
      rv ← 0;
      goto Break;
    }
  }
  rv ← 0;      /* in case of a continue—the loop is about to start */
```

This code is used in sections 514 and 518.

516. Execute *t→left* for each value in *t→args* after saving it to the global variable named in *t→str*.

```
⟨ Execute a TFOR node 516 ⟩ ≡
  save_xerrok ← *xerrok;
  while (*ap ≠ Λ) {
    setstr(global(t→str), *ap++, KSH_UNWIND_ERROR);
    *xerrok ← save_xerrok;      /* undo xerrok in all iterations except the last */
    rv ← execute(t→left, flags & XERROK, xerrok);
  }     /* ripple xerrok set at final iteration */
```

This code is used in section 514.

517. *do_selectargs* returns with the user's selection of items in *ap* which similarly to `for` is saved in the global variable named in *t→str* then *t→left* is executed.

```
⟨ Execute a TSELECT node 517 ⟩ ≡
  for ( ; ; ) {
    if (¬(cp ← do_selectargs(ap, is_first))) {
      rv ← 1;
      break;
    }
    is_first ← false;
    setstr(global(t→str), cp, KSH_UNWIND_ERROR);
    rv ← execute(t→left, flags & XERROK, xerrok);
  }
```

This code is used in section 514.

518. TWHILE and TUNTIL prepare a long jump return point in the same way and then repeatedly execute *t→right* if executing *t→left* is acceptable.

```
⟨ Execute a TWHILE/TUNTIL node and break 518 ⟩ ≡
  ⟨ Establish a return point for break/continue 515 ⟩
  while ((execute(t→left, XERROK, Λ) ≡ 0) ≡ (t→type ≡ TWHILE))
    rv ← execute(t→right, flags & XERROK, xerrok);
  break;
```

This code is used in section 506.

519. TIF and TELIF are distinguished only so that a syntax tree can be reconstructed into the correct source code. Both operators execute the condition in *t→left* then either the consequent or alternate in *t→right→left* or *t→right→right* respectively.

```
⟨ Execute a TIF/TELIF node and break 519 ⟩ ≡
  if (t→right ≡ Λ) break;      /* should be error */
  rv ← execute(t→left, XERROK, Λ) ≡ 0 ? execute(t→right→left, flags & XERROK, xerrok) : execute(t→right→right,
      flags & XERROK, xerrok);
  break;
```

This code is used in section 506.

520. TCASE implements case by evaluating the value to test into *cp* then walking the right-hand path of *t→left* until a match is found, then executing *its t→left*.

```
⟨ Execute a TCASE node and break 520 ⟩ ≡
  cp ← evalstr(t→str, DOTILDE);
  for (t ← t→left; t ≠ Λ ∧ t→type ≡ TPAT; t ← t→right) {
    for (ap ← t→vars; *ap; ap++) {
      if ((s ← evalstr(*ap, DOTILDE | DOPAT)) ∧ gmatch(cp, s, false)) goto Found;
    }
  }
  break;
Found: rv ← execute(t→left, flags & XERROK, xerrok);
  break;
```

This code is used in section 506.

521. Note that while shell's exec will end up here to perform the *execve* part, this is *not* "the implementation of exec". Any evaluated TCOM (ie. regular command) will execute a TEXEC node—possibly in a sub-process—if the command is a foreign executable.

```
⟨ Execute a TEXEC node and break 521 ⟩ ≡
  s ← t→args[0];
  ap ← makenv( );
  restoresigs( );
  cleanup_proc_env( );
  execve(t→str, t→args, ap);
  if (errno ≡ ENOEXEC) scriptexec(t, ap);      /* execute-enabled file is not executable (eg. a script) */
  else errorf("%s: %s", s, strerror(errno));
```

This code is used in section 506.

522. This space intentionally left blank.

523. If a file is available for execution—accessible, readable and its execute bits are set—but lacks the magic bytes which instruct the kernel how to execute it, ksh tries to interpret it as a script with *scriptexec*. The interpreter to use is taken from `$EXECSHELL` or `_PATH_BSHELL` and prepended to *tp→args* before retrying *execve*. Note that this overwrites memory not technically allocated for *args* but this is still "safe" as it in fact holds the **Area** link that *args* is allocated in and *execve* is about to be called[1].

This is largely a historic artefact from before unix understood a leading hashbang ("`#!`") to indicate which interpreter to use on the file. The kernel in *execve* will load and run the interpreter except on an odd or ancient unix.

In short this is **not** used for a normal executable script beginning with "`#!/bin/sh`".

```
⟨ exec.c 461 ⟩ +≡
  static void scriptexec(struct Op *tp, char **ap)
  {
    char *shell;
    shell ← str_val(global("EXECSHELL"));
    if (shell ∧ *shell) shell ← search(shell, search_path, X_OK, Λ);
    if (¬shell ∨ ¬*shell) shell ← _PATH_BSHELL;
    *tp→args -- ← tp→str;
    *tp→args ← shell;
    execve(tp→args[0], tp→args, ap);
    errorf("%s: %s: %s", tp→str, shell, strerror(errno));
      /* report both the program that was run and the bogus shell */
  }
```

[1] TODO: Should **tp→args* be restored if this second *execve* fails?

524. External Commands. Deal with the shell built-ins `builtin`, `exec` and `command` since they can be followed by other commands. This must be done before we know if we should create a local block, which must be done before we can do a path search (in case the assignments change `$PATH`).

Odd cases:

* `FOO=bar exec > /dev/null $FOO` is kept but not exported
* `FOO=bar exec foobar $FOO` is exported
* `FOO=bar command exec > /dev/null $FOO` is neither kept nor exported
* `FOO=bar command $FOO` is neither kept nor exported
* `PATH=... foobar` use new `$PATH` in foobar search

```
#define FC_SPECBI BIT(0) /* special builtin */
#define FC_FUNC BIT(1) /* function builtin */
#define FC_REGBI BIT(2) /* regular builtin */
#define FC_UNREGBI BIT(3) /* un-regular builtin (¬special, ¬regular) */
#define FC_BI (FC_SPECBI | FC_REGBI | FC_UNREGBI)
#define FC_PATH BIT(4) /* do path search */
#define FC_DEFPATH BIT(5) /* use default path in path search */
```

⟨ exec.c 461 ⟩ +≡

```
static int comexec(struct Op *t, struct tbl *volatile tp, char **ap, volatile int flags,
        volatile int *xerrok)
{
  int i;
  volatile int rv ← 0;
  char *cp;
  char **lastp;
  struct Op *texec;
  int type_flags;
  int keepasn_ok;
  int fcflags ← FC_BI | FC_FUNC | FC_PATH;
  int bourne_function_call ← 0;
  ⟨ Set $_ to the last command 525 ⟩
  keepasn_ok ← 1;
  while (tp ∧ tp→type ≡ CSHELL) {
    fcflags ← FC_BI | FC_FUNC | FC_PATH; /* undo effects of command */
    if (tp→val.f ≡ c_builtin) {⟨ Special treatment for built-in builtin 526 ⟩}
    else if (tp→val.f ≡ c_exec) {⟨ Special treatment for built-in exec 527 ⟩}
    else if (tp→val.f ≡ c_command) {⟨ Special treatment for built-in command 528 ⟩}
    else break;
    tp ← findcom(ap[0], fcflags & (FC_BI | FC_FUNC));
  }
  ⟨ Set the type of new variables 529 ⟩
  ⟨ Locate the command's details 530 ⟩
  switch (tp→type) {⟨ Carry out a command 531 ⟩}
Leave:
  if (flags & XEXEC) {
    exstat ← rv;
    unwind (LLEAVE);
  }
  return rv;
}
```

525.

/* snag the last argument for $_ XXX not the same as AT&T ksh, * which only seems to set $_ after a newline (but not in * functions/dot scripts, but in interactive and script) - * perhaps save last arg here and set it in shell()?.

⟨ Set $_ to the last command 525 ⟩ ≡

```
if (¬Flag(FSH) ∧ Flag(FTALKING) ∧ ∗(lastp ← ap)) {
  while (∗++lastp)
    /∗ skip to the end of the list ∗/ ;
  setstr(typeset("_", LOCAL, 0, INTEGER, 0), ∗−−lastp, KSH_RETURN_ERROR);
    /∗ setstr() can't fail here ∗/
}
```

This code is used in section 524.

526. The error message is an inconsistent use of builtin (C) vs. built-in (English).

⟨ Special treatment for built-in builtin 526 ⟩ ≡

```
if ((cp ← ∗++ap) ≡ Λ) {
  tp ← Λ;
  break;
}
tp ← findcom(cp, FC_BI);
if (tp ≡ Λ) errorf("builtin: %s: not a builtin", cp);
continue;
```

This code is used in section 524.

527. This is (part of) where the shell built-in exec is implemented, when it has arguments, by setting the XEXEC flag and using findcom to locate the new executable.

⟨ Special treatment for built-in exec 527 ⟩ ≡

```
if (ap[1] ≡ Λ) break;
ap++;
flags |= XEXEC;
```

This code is used in section 524.

528. It's ugly dealing with options in two places (here and in *c_command*), but such is life.

```
⟨ Special treatment for built-in command 528 ⟩ ≡
  int optc, saw_p ← 0;
  ksh_getopt_reset(&builtin_opt, 0);
  while ((optc ← ksh_getopt(ap, &builtin_opt, ":p")) ≡ 'p') saw_p ← 1;
  if (optc ≠ EOF) break;      /* command -vV or something */
  fcflags ← FC_BI | FC_PATH;      /* don't look for functions */
  if (saw_p) {
    if (Flag(FRESTRICTED)) {
      warningf(true, "command␣-p:␣restricted");
      rv ← 1;
      goto Leave;
    }
    fcflags |= FC_DEFPATH;
  }
  ap += builtin_opt.optind;
  keepasn_ok ← 0;      /* POSIX says special builtins lose their status if accessed using command. */
  if (¬ap[0]) {      /* ensure command with no args exits with 0 */
    subst_exstat ← 0;
    break;
  }
```

This code is used in section 524.

529.

```
⟨ Set the type of new variables 529 ⟩ ≡
  if (keepasn_ok ∧ (¬ap[0] ∨ (tp ∧ (tp→flag & KEEPASN)))) type_flags ← 0;
  else {
    newblock( );      /* create new variable/function block */
    if (keepasn_ok ∧ tp ∧ tp→type ≡ CFUNC ∧ ¬(tp→flag & FKSH)) {
      /* ksh functions don't keep assignments, POSIX functions do. */
      bourne_function_call ← 1;
      type_flags ← 0;
    }
    else type_flags ← LOCAL | LOCAL_COPY | EXPORT;
  }
  if (Flag(FEXPORT)) type_flags |= EXPORT;
  for (i ← 0; t→vars[i]; i++) {
    cp ← evalstr(t→vars[i], DOASNTILDE);
    if (Flag(FXTRACE)) {
      if (i ≡ 0) shf_fprintf(shl_out, "%s", PS4_SUBSTITUTE(str_val(global("PS4"))));
      shf_fprintf(shl_out, "%s%s", cp, t→vars[i + 1] ? "␣" : "\n");
      if (¬t→vars[i + 1]) shf_flush(shl_out);
    }
    typeset(cp, type_flags, 0, 0, 0);
    if (bourne_function_call ∧ ¬(type_flags & EXPORT)) typeset(cp, LOCAL | LOCAL_COPY | EXPORT, 0, 0, 0);
  }
```

This code is used in section 524.

530. ⟨ Locate the command's details 530 ⟩ ≡

```
if ((cp ← *ap) ≡ Λ) {
  rv ← subst_exstat;
  goto Leave;
}
else if (¬tp) {
  if (Flag(FRESTRICTED) ∧ strchr(cp, '/')) {
    warningf(true, "%s: restricted", cp);
    rv ← 1;
    goto Leave;
  }
  tp ← findcom(cp, fcflags);
}
```

This code is used in section 524.

531. ⟨ Carry out a command 531 ⟩ ≡

```
case CSHELL: rv ← call_builtin(tp, ap); break;    /* shell built-in */
```

See also sections 532 and 535.

This code is used in section 524.

532.

```
⟨ Carry out a command 531 ⟩ +≡
case CFUNC:
  {
    volatile int old_xflag, old_inuse;
    const char *volatile old_kshname;
    if (¬(tp→flag & ISSET)) {⟨ Load a function that's in $FPATH or break 533 ⟩}
    ⟨ Process $0 and function arguments 534 ⟩
    old_xflag ← Flag(FXTRACE);
    Flag(FXTRACE) ← tp→flag & TRACE ? true : false;
    old_inuse ← tp→flag & FINUSE;
    tp→flag |= FINUSE;
    genv→type ← E_FUNC;
    i ← sigsetjmp(genv→jbuf, 0);
    if (i ≡ 0) {      /* seems odd to pass XERROK here, but AT&T ksh does */
      exstat ← execute(tp→val.t, flags & XERROK, xerrok);
      i ← LRETURN;
    }
    kshname ← old_kshname;
    Flag(FXTRACE) ← old_xflag;
    tp→flag ← (tp→flag & ∼FINUSE) | old_inuse;
    if ((tp→flag & (FDELETE | FINUSE)) ≡ FDELETE) {
        /* Were we deleted while executing? If so, free the execution tree. TODO: Unfortunately, the
           table entry is never re-used until the lookup table is expanded. */
      if (tp→flag & ALLOC) {
        tp→flag &= ∼ALLOC;
        tfree(tp→val.t, tp→areap);
      }
      tp→flag ← 0;
    }
    switch (i) {
    case LRETURN: case LERROR: rv ← exstat; break;
    case LINTR: case LEXIT: case LLEAVE: case LSHELL: quitenv(Λ); unwind (i);
    default: quitenv(Λ); internal_errorf("CFUNC␣%d", i);
    }
    break;
  }
```

533.

⟨ Load a function that's in $FPATH or **break** 533 ⟩ ≡

```
struct tbl *ftp;
if (¬tp→u.fpath) {
  if (tp→u2.errno_) {
    warningf(true, "%s: can't find function ""definition file - %s", cp, strerror(tp→u2.errno_));
    rv ← 126;
  }
  else {
    warningf(true, "%s: can't find function ""definition file", cp);
    rv ← 127;
  }
  break;
}
if (Include(tp→u.fpath, 0, Λ, 0) < 0) {
  warningf(true, "%s: can't open function definition file %s - %s", cp, tp→u.fpath,
      strerror(errno));
  rv ← 127;
  break;
}
if (¬(ftp ← findfunc(cp, hash(cp), false)) ∨ ¬(ftp→flag & ISSET)) {
  warningf(true, "%s: function not defined by %s", cp, tp→u.fpath);
  rv ← 127;
  break;
}
tp ← ftp;
```

This code is used in section 532.

534.

⟨ Process $0 and function arguments 534 ⟩ ≡

```
old_kshname ← kshname;
  /* ksh functions set $0 to function name, POSIX functions leave $0 unchanged. */
if (tp→flag & FKSH) kshname ← ap[0];
else ap[0] ← (char *) kshname;
genv→loc→argv ← ap;
for (i ← 0; *ap++ ≠ Λ; i++) ;
genv→loc→argc ← i − 1;
if (tp→flag & FKSH) {
    /* ksh-style functions handle getopts sanely, bourne/posix functions are insane... */
  genv→loc→flags |= BF_DOGETOPTS;
  genv→loc→getopts_state ← user_opt;
  getopts_reset(1);
}
```

This code is used in section 532.

535. *errno_* will be set if the named command was found but could not be executed (permissions, no execute bit, directory, etc.). Print out a (hopefully) useful error message and set the exit status to 126.

```
⟨ Carry out a command 531 ⟩ +≡
case CEXEC:      /* executable command */
case CTALIAS:    /* tracked alias */
  if (¬(tp→flag & ISSET)) {
    if (tp→u2.errno_) {
      warningf(true, "%s: cannot execute - %s", cp, strerror(tp→u2.errno_));
      rv ← 126;      /* POSIX */
    }
    else {
      warningf(true, "%s: not found", cp);
      rv ← 127;
    }
    break;
  }
  if (¬Flag(FSH)) {      /* set $_ to program's full path */
    setstr(typeset("_", LOCAL | EXPORT, 0, INTEGER, 0), tp→val.s, KSH_RETURN_ERROR);
      /* setstr( ) can't fail here */
  }
  if (flags & XEXEC) {
    j_exit( );
    if (¬(flags & XBGND) ∨ Flag(FMONITOR)) {
      setexecsig(&sigtraps[SIGINT], SS_RESTORE_ORIG);
      setexecsig(&sigtraps[SIGQUIT], SS_RESTORE_ORIG);
    }
  }
  memset(&texec, 0, sizeof (texec));      /* to fork we set up a TEXEC node and call exchild */
  texec.type ← TEXEC;
  texec.left ← t;      /* for tprint */
  texec.str ← tp→val.s;
  texec.args ← ap;
  rv ← exchild(&texec, flags, xerrok, −1);
  break;
```

536. Functions & Built-in Commands. Much more simple than an external command, *shcomexec* executes a shell built-in command.

```
⟨ exec.c 461 ⟩ +≡
  int shcomexec(char **wp)
  {
    struct tbl *tp;
    tp ← ktsearch(&builtins, *wp, hash(*wp));
    if (tp ≡ Λ) internal_errorf("%s: %s", __func__, *wp);
    return call_builtin(tp, wp);
  }
```

537. Built-in commands use *builtin_argv0* to mimic `$0`. This is probably an historical artefact as the value is passed to the built-in's handler.

```
⟨ Global variables 5 ⟩ +≡
  char *builtin_argv0;      /* name of the built-in (its $0) */
  int builtin_flag;         /* flags of called built-in (SPEC_BI &c.) */
```

538.
```
⟨ Externally-linked variables 6 ⟩ +≡
  extern char *builtin_argv0;
  extern int builtin_flag;
```

539.
```
⟨ exec.c 461 ⟩ +≡
  static int call_builtin(struct tbl *tp, char **wp)
  {
    int rv;
    builtin_argv0 ← wp[0];
    builtin_flag ← tp→flag;
    shf_reopen(1, SHF_WR, shl_stdout);
    shl_stdout_ok ← 1;
    ksh_getopt_reset(&builtin_opt, GF_ERROR);
    rv ← (*tp→val.f)(wp);
    shf_flush(shl_stdout);
    shl_stdout_ok ← 0;
    builtin_flag ← 0;
    builtin_argv0 ← Λ;
    return rv;
  }
```

540. Built-in commands are saved in *builtins* by *builtin*. First it checks which flags should be set for the built-in command then inserts it into the global hash table.

```
⟨ exec.c 461 ⟩ +≡
  void builtin(const char *name, int(*func)(char **))
  {
    struct tbl *tp;
    int flag;
    for (flag ← 0; ; name++) {
      if (*name ≡ '=') flag |= KEEPASN;      /* command does variable assignment */
      else if (*name ≡ '*') flag |= SPEC_BI;      /* POSIX special builtin */
      else if (*name ≡ '+') flag |= REG_BI;      /* POSIX regular builtin */
      else break;
    }
    tp ← ktenter(&builtins, name, hash(name));
    tp→flag ← DEFINED | flag;
    tp→type ← CSHELL;
    tp→val.f ← func;
  }
```

541. User-defined functions are saved in **Env**→*funs*. *fundfunc* searches for one and creates it if it isn't found. It appears as though functions are always created in the top level environment so searching while $l\text{→}next \neq \Lambda$ is pointless. To change this *ktenter* should use &*genv*→*loc*→*funs*. TODO: More research needed.

```
⟨ exec.c 461 ⟩ +≡
  struct tbl *findfunc(const char *name, unsigned int h, int create)
  {
    struct Block *l;
    struct tbl *tp ← Λ;
    for (l ← genv→loc; l; l ← l→next) {
      tp ← ktsearch(&l→funs, name, h);
      if (tp) break;
      if (¬l→next ∧ create) {
        tp ← ktenter(&l→funs, name, h);
        tp→flag ← DEFINED;
        tp→type ← CFUNC;
        tp→val.t ← Λ;
        break;
      }
    }
    return tp;
  }
```

542. A user-defined function is created or removed with *Define* née *define*. If the function is found but is in use it is marked to be deleted rather than immediately overwritten.

```
⟨ exec.c 461 ⟩ +≡
  int Define(const char *name, struct Op *t)
  {
    struct tbl *tp;
    int was_set ← 0;
    while (1) {
      tp ← findfunc(name, hash(name), true);
      if (tp→flag & ISSET) was_set ← 1;
      if (tp→flag & FINUSE) {⟨ Mark an in-use function to be deleted later 543 ⟩}
      else break;
    }
    ⟨ Free any old allocation and undefine a user function 544 ⟩
    tp→val.t ← tcopy(t→left, tp→areap);
    tp→flag |= (ISSET | ALLOC);
    if (t→u.ksh_func) tp→flag |= FKSH;
    return 0;
  }
```

543. If a function is being executed but is due to be replaced we zap its table entry so further calls to *findfunc* won't see it.

```
⟨ Mark an in-use function to be deleted later 543 ⟩ ≡
  tp→name[0] ← '\0';
  tp→flag &= ~DEFINED;
  tp→flag |= FDELETE;
```

This code is used in section 542.

544. ⟨ Free any old allocation and undefine a user function 544 ⟩ ≡

```
  if (tp→flag & ALLOC) {
    tp→flag &= ~(ISSET | ALLOC);
    tfree(tp→val.t, tp→areap);
  }
  if (t ≡ Λ) {
    ktdelete(tp);
    return was_set ? 0 : 1;
  }
```

This code is used in section 542.

545. A command can refer to either a built-in, a user-defined function or a filesystem command (or alias). *fundcom* searches for a function, cached command or built-in in that order. POSIX, however, says special built-ins first, then functions, then POSIX regular built-ins, then search `$PATH`.

TODO: POSIX also says that non-special/non-regular built-ins must be triggered by some user-controllable means like a special directory in `$PATH` which requires modifications to *search*. Tracked aliases should be modified to allow tracking of built-in commands. This should be under control of the `FPOSIX` flag. If this is changed, also change *c_whence*.

```
⟨ exec.c 461 ⟩ +≡
  struct tbl *findcom(const char *name, int flags)
  {
    static struct tbl temp;
    unsigned int h ← hash(name);
    struct tbl *tp ← Λ, *tbi;
    int insert ← Flag(FTRACKALL);      /* insert if not found */
    char *fpath;      /* for function autoloading */
    char *npath;
    if (strchr(name, '/') ≠ Λ) {      /* the command is an absolute pathname */
      insert ← 0;
      flags &= ~FC_FUNC;      /* prevent $FPATH search below */
      goto Search;
    }
    ⟨ (if (flags & FC_SPECBI)) Search for a special built-in 546 ⟩
    if (¬tp ∧ (flags & FC_FUNC)) {⟨ Search for a user-defined function 547 ⟩}
    if (¬tp ∧ (flags & FC_REGBI) ∧ tbi ∧ (tbi→flag & REG_BI)) tp ← tbi;
    if (¬tp ∧ (flags & FC_UNREGBI) ∧ tbi) tp ← tbi;
    if (¬tp ∧ (flags & FC_PATH) ∧ ¬(flags & FC_DEFPATH)) {⟨ Search for a tracked alias 548 ⟩}
  Search:
    if ((¬tp ∨ (tp→type ≡ CTALIAS ∧ ¬(tp→flag & ISSET))) ∧
          (flags & FC_PATH)) {
      ⟨ Search $PATH or $FPATH 549 ⟩
    }
    return tp;
  }
```

546.
```
⟨ (if (flags & FC_SPECBI)) Search for a special built-in 546 ⟩ ≡
  tbi ← (flags & FC_BI) ? ktsearch(&builtins, name, h) : Λ;
  if ((flags & FC_SPECBI) ∧ tbi ∧ (tbi→flag & SPEC_BI)) tp ← tbi;
```
This code is used in section 545.

547.
```
⟨ Search for a user-defined function 547 ⟩ ≡
  tp ← findfunc(name, h, false);
  if (tp ∧ ¬(tp→flag & ISSET)) {
    if ((fpath ← str_val(global("FPATH"))) ≡ null) {
      tp→u.fpath ← Λ;
      tp→u2.errno_ ← 0;
    }
    else tp→u.fpath ← search(name, fpath, R_OK, &tp→u2.errno_);
  }
```
This code is used in section 545.

548. ⟨ Search for a tracked alias 548 ⟩ ≡
```
tp ← ktsearch(&taliases, name, h);
if (tp ∧ (tp→flag & ISSET) ∧ access(tp→val.s, X_OK) ≠ 0) {
  if (tp→flag & ALLOC) {
    tp→flag &= ∼ALLOC;
    afree(tp→val.s, APERM);
  }
  tp→flag &= ∼ISSET;
}
```
This code is used in section 545.

549. ⟨ Search $PATH or $FPATH 549 ⟩ ≡
```
if (¬tp) {⟨ Prepare a new command object 550 ⟩}
npath ← search(name, flags & FC_DEFPATH ? def_path : search_path, X_OK, &tp→u2.errno_);
if (npath) {⟨ Cache the pathname found 551 ⟩}
else if ((flags & FC_FUNC) ∧
    (fpath ← str_val(global("FPATH"))) ≠ null ∧
    (npath ← search(name, fpath, R_OK, &tp→u2.errno_)) ≠ Λ)
{⟨ Cache the function pathname found 552 ⟩}
```
This code is used in section 545.

550. ⟨ Prepare a new command object 550 ⟩ ≡
```
if (insert ∧ ¬(flags & FC_DEFPATH)) {
  tp ← ktenter(&taliases, name, h);
  tp→type ← CTALIAS;
}
else {
  tp ← &temp;
  tp→type ← CEXEC;
}
tp→flag ← DEFINED;      /* make ∼ISSET */
```
This code is used in section 549.

551. ⟨ Cache the pathname found 551 ⟩ ≡
```
if (tp ≡ &temp) {
  tp→val.s ← npath;
}
else {
  tp→val.s ← str_save(npath, APERM);
  if (npath ≠ name) afree(npath, ATEMP);
}
tp→flag |= ISSET | ALLOC;
```
This code is used in section 549.

552. An undocumented feature of AT&T ksh is that it searches $FPATH if a command is not found, even if the command hasn't been set up as an autoloaded function (ie. no "`typeset -uf`").

⟨ Cache the function pathname found 552 ⟩ ≡
```
tp ← &temp;
tp→type ← CFUNC;
tp→flag ← DEFINED;      /* make ∼ISSET */
tp→u.fpath ← npath;
```
This code is used in section 549.

553. When the function/command cache becomes stale they can be flushed, either all of them or just those with relative pathnames.

```
⟨ exec.c 461 ⟩ +≡
  void flushcom(int all)
  {
    struct tbl *tp;
    struct tstate ts;
    for (ktwalk(&ts, &taliases); (tp ← ktnext(&ts)) ≠ Λ; )
      if ((tp→flag & ISSET) ∧ (all ∨ tp→val.s[0] ≠ '/')) {
        if (tp→flag & ALLOC) {
          tp→flag &= ∼(ALLOC | ISSET);
          afree(tp→val.s, APERM);
        }
        tp→flag &= ∼ISSET;
      }
  }
```

554. Expansion/Substitution. The province of `eval.c`, expanding a string is carried out in two stages. The first pass handles quoting, `$IFS` separation and substitution of `${ ... }`, `$( ... )` and `$(( ... ))`. The second pass handles alternation (`{...,...}`) and filename expansion (ie. globbing, or `*/?/[...]`).

⟨ `eval.c` 554 ⟩ ≡

```
#include <sys/stat.h>
#include <ctype.h>
#include <dirent.h>
#include <fcntl.h>
#include <pwd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
static int varsub(Expand *, char *, char *, int *, int *);
static int comsub(Expand *, char *);
static char *trimsub(char *, char *, int);
static void glob(char *, XPtrV *, int);
static void globit(XString *, char **, char *, XPtrV *, int);
static char *maybe_expand_tilde(char *, XString *, char **, int);
static char *tilde(char *);
static char *homedir(char *);
static void alt_expand(XPtrV *, char *, char *, char *, int);
static struct tbl *varcpy(struct tbl *);
```

See also sections 556, 557, 558, 559, 565, 575, 594, 600, 601, 606, 611, 612, 613, 768, 769, 784, and 785.

555. ⟨ Shared function declarations 4 ⟩ +≡

```
char *substitute(const char *, int);
char **eval(char **, int);
char *evalstr(char *cp, int);
char *evalonestr(char *cp, int);
char *debunk(char *, const char *, size_t);
void expand(char *, XPtrV *, int);
int glob_str(char *, XPtrV *, int);
```

556. There are several global interfaces to the main *expand* routine. The simplest, *evalstr*, performs substitution over a single lexical word.

⟨ `eval.c` 554 ⟩ +≡

```
char *evalstr(char *cp, int f)
{
  XPtrV w;
  XPinit(w, 1);
  expand(cp, &w, f);
  cp ← (XPsize(w) ≡ 0) ? null : (char *) *XPptrv(w);
  XPfree(w);
  return cp;
}
```

557. *eval* does the same over each of the expressions in the list *ap* and returns the result as a single concatenated C-string?

```
⟨ eval.c 554 ⟩ +≡
  char **eval(char **ap, int f)
  {
    XPtrV w;
    if (*ap ≡ Λ) return ap;
    XPinit(w, 32);
    XPput(w, Λ);      /* space for shell name $0 */
    while (*ap ≠ Λ) expand(*ap++, &w, f);
    XPput(w, Λ);
    return (char **) XPclose(w) + 1;
  }
```

558. *substitute* compiles an expression into a lexical word and then performs substitution on it. It is used to evaluate plain C-strings.

```
⟨ eval.c 554 ⟩ +≡
  char *substitute(const char *cp, int f)
  {
    struct Source *s, *sold;
    if (disable_subst) return str_save(cp, ATEMP);
    sold ← source;
    s ← pushs(SWSTR, ATEMP);
    s→start ← s→str ← cp;
    source ← s;
    if (yylex(ONEWORD) ≠ LWORD) internal_errorf("substitute");
    source ← sold;
    afree(s, ATEMP);
    return evalstr(yylval.cp, f);
  }
```

559. The final interface is used by *iosetup* and returns "only one component ... to expand redirection files".

```
⟨ eval.c 554 ⟩ +≡
  char *evalonestr(char *cp, int f)
  {
    XPtrV w;
    XPinit(w, 1);
    expand(cp, &w, f);
    switch (XPsize(w)) {
    case 0: cp ← null; break;
    case 1: cp ← (char *) *XPptrv(w); break;
    default: cp ← evalstr(cp, f & ∼DOGLOB); break;
    }
    XPfree(w);
    return cp;
  }
```

560. The enormous *expand* routine. State machine in *type*. Loops over each lexeme entering a new state depending on character type, stack is kept in *st*. Looks for a token (character or expansion or list) to append to the *argv* being built.

Moving parts:
ds/dp are the destination string, sp the source string initially set to cp.
wp will become a list of pointers to strings?
type is the type of expansion the current character is part of.
word dictates how the current character will affect word splitting.
quote indicates whether the current character is being quoted.

561. Given "`abc'def'"ghi"`". Enter loop *type* ≡ `XBASE`; *word* ≡ `IFS_WS` usually or `IFS_WORD` if 'blanks' (ie. `$IFS`) should be ignored, the next $c \equiv$ `CHAR` so set $c \leftarrow *sp{+}{+}$ &check for word-separation: there is none so append to *dp* and set *word* ← `IFS_WORD`.

Repeat for `b` & `c`.

The fourth time around the next $c \equiv$ `OQUOTE` so *word* ← `IFS_QUOTE`, disable *tilde_ok* and enable *quote* but does not check for word separation.

The next $c \equiv$ `QCHAR` and *quote* is magically enabled (or `CHAR`? Impossible?); either way $c \leftarrow *sp{+}{+}$ and no word separator means `d` is appended to *dp*. Repeat for `e` & `f`.

The second `'` has been identified as `CQUOTE` so *quote* is disabled and the next character examined. It is `"` marked `OQUOTE` so *quote* is enabled again.

`g`, `h` & `i` are appended as `def` was (the type of quote is not distinguished here) and the second `"` is a `CQUOTE` which disables *quote*.

Finally the next $c \equiv$ `EOS` so $c \leftarrow 0$ and word separation does not append it. *word* ≡ `IFS_WORD` and *fdo* ≡ 0, untouched. The destination string is closed and globbing magic may be debunked according to f. The (debunked) string's pointer is appended to *wp* and *ds* & *dp* are re-initialised.

If a globbing construct were encountered (eg. with "`abc*def`") this would be noticed during the attempt at ending the word (**if** (¬*quote*) in ⟨ Check for end of word or `$IFS` separation 586 ⟩). This will enable the `DOMAGIC_` and/or `DOGLOB` flags in *fdo* according to the particular character and f. `MAGIC` is appended before the character itself.

562. Given "`$foo.$bar`", enter loop `XBASE` & `IFS_WS`. The first $c \equiv$ `OSUBST` so the variable name is saved in **char** ∗*varname* and *sp* incremented to the next byte after it, which will be the `CSUBST` from ⟨ Detect `$`*variable*-expansion 348 ⟩.

varsub will look the variable's value up, saving the variable and value in *x.var* and *x.str*. The state is changed to `XSUB` and *sp* is moved to the character after the `CSUBST`.

In `XSUB` & `XSUBMID` states *c* is set to the next character from the substituted value until the end is reached and the state returns to `XBASE`. After each character the state is "changed" to `XSUBMID` before attempting word-splitting, including attempting globbing etc. if unquoted.

When the end of `$foo`'s value is reached the `XBASE` will encounter and append `.` as before. The same repeats for `$bar`.

563. If `$foo` contains "`foo`" and `$bar` "`bar␣␣and␣baz`" then expanding `$foo` appends 3 bytes as above. Expanding `$bar` appends "`bar`" then the next character is ␣ so word-splitting changes—it's an `$IFS` character and *word* ≡ `IFS_WORD` so the word so far is emitted. *word* is set to `IFW_WS`.

The next character is ␣ again so proceeds directly to word-splitting but this time word is `IFS_WS` so another is not emitted.

The next character is `a` so word-splitting appends it and sets *word* to `IFS_WORD`. "`nd`" are then appended and the process repeats for the ␣ and "`bar`".

564. If instead bar `$bar` contains "`*`" then in the `XSUB` state the expander proceeds to the word-splitting algorithm. This causes `DOMAGIC_` and `DOGLOB` to be enabled in *fdo* and `MAGIC` appended to the output followed by the `*`.

The machine loops again and encounters the `EOS` so word-splitting is performed by calling *glob*.

565.

```
#define DOBLANK BIT(0) /* perform blank interpretation */
#define DOGLOB BIT(1) /* expand [?*] */
#define DOPAT BIT(2) /* quote [?*] */
#define DOTILDE BIT(3) /* normal ␣ expansion (first character) */
#define DONTRUNCOMMAND BIT(4) /* do not run $(...) commands */
#define DOASNTILDE BIT(5) /* assignment ␣ expansion (after [=:]) */
#define DOBRACE_ BIT(6) /* used by expand: do brace expansion */
#define DOMAGIC_ BIT(7) /* — „ —: string contains MAGIC */
#define DOTEMP_ BIT(8) /* — „ —: in word part of ${...[%#=?]...} */
#define DOVACHECK BIT(9) /* variable assign check (for typeset, set, &c.) */
#define DOMARKDIRS BIT(10) /* force markdirs behaviour */
```

```
#define XBASE 0 /* scanning original */
#define XSUB 1 /* expanding "${...}" string */
#define XARGSEP 2 /* ifs0 between "$*" */
#define XARG 3 /* expanding $*, $@ */
#define XCOM 4 /* expanding $() */
#define XNULLSUB 5 /* "$@" when $# is 0 (don't generate a word) */
#define XSUBMID 6 /* middle of expanding "${...}" */
```

⟨ eval.c 554 ⟩ +≡

```
void expand(char *cp, /* input word */
XPtrV *wp, /* output words */
int f) /* DO* flags */
{
int c ← 0;
int type; /* expansion type */
int quote ← 0; /* quoted */
XString ds; /* destination string */
char *dp, *sp; /* destination / source */
int fdo, word; /* second pass flags; have word */
int doblank; /* field splitting of parameter/command subst */
Expand x ← {Λ, {Λ}, Λ, 0}; /* expansion variables; TODO: x ← {0} would also work */
SubType st_head, *st;
int newlines ← 0; /* For trailing newlines in COMSUB */
int saw_eq, tilde_ok;
int make_magic;
size_t len;
⟨ Initialise expand state 567 ⟩
while (1) {
Xcheck(ds, dp);
switch (type) { /* these also have potentially continue-ing branch */
⟨ Expand switch (type) ≡ XBASE 568 ⟩; break;
⟨ Expand switch (type) ≡ XNULLSUB 582 ⟩; continue;
⟨ Expand switch (type) ≡ (XSUB ∨ XSUBMID) 583 ⟩; break;
⟨ Expand switch (type) ≡ (XARGSEP ∨ XARG) 584 ⟩; break;
⟨ Expand switch (type) ≡ XCOM 585 ⟩; break;
}
⟨ Check for end of word or $IFS separation 586 ⟩
}
done:
for (st ← &st_head; st ≠ Λ; st ← st→next) {
```

```
    if (st→var ≡ Λ ∨ (st→var→flag & RDONLY) ≡ 0) continue;
    afree(st→var, ATEMP);
  }
}
```

566. Expansion takes places in buffers held by the **Expand** object x. Each recursion back into *expand* results in a new x somewhere in C's stack. The buffers held by it are (I expect) allocated in a temporary **Area**.

```
⟨ Type definitions 17 ⟩ +≡
typedef struct Expand {      /* TODO: any reason expect could not s/type/x.type/? */
    /* int type; */
  const char *str;      /* string */
  union {      /* source: */
    const char **strv;      /* string[] */
    struct Shf *shf;      /* file */
  } u;
  struct tbl *var;      /* variable in ${⟨var⟩} */
  short split;      /* split "$@" / call waitlast $(...) */
} Expand;
```

567. The same for everything (and much of it could/should be in the declarations), the expansion state machine is initialised to XBASE in *type* with an empty stack of nested substitutions in *st*. *tilde_ok* is a flag which seems to indicate, if instructed by the caller's flags f, whether expansion of ~ is to be performed. *word* is the character type which will end reading the current word.

```
⟨ Initialise expand state 567 ⟩ ≡
if (cp ≡ Λ) internal_errorf("expand(NULL)");
if ((f & DOVACHECK) ∧ is_wdvarassign(cp)) {      /* for alias, readonly, set & typeset commands */
  f &= ∼(DOVACHECK | DOBLANK | DOGLOB | DOTILDE);
  f |= DOASNTILDE;
}
if (Flag(FNOGLOB)) f &= ∼DOGLOB;
if (Flag(FMARKDIRS)) f |= DOMARKDIRS;
if (Flag(FBRACEEXPAND) ∧ (f & DOGLOB)) f |= DOBRACE_;
Xinit(ds, dp, 128, ATEMP);      /* initial destination string */
fdo ← saw_eq ← doblank ← make_magic ← 0; type ← XBASE; sp ← cp;      /* TODO: move up */
tilde_ok ← (f & (DOTILDE | DOASNTILDE)) ? 1 : 0;      /* must be 1 or 0 */
word ← (f & DOBLANK) ? IFS_WS : IFS_WORD;
memset(&st_head, 0, sizeof (st_head));
st ← &st_head;
```

This code is used in section 565.

568. Starts and is usually in XBASE state which watches out for lexemes which will enter a new state (open quote, substitute, etc.) or continues with the current character.

TODO: Move the **case** part of the statement to each section.

```
⟨ Expand switch (type) ≡ XBASE 568 ⟩ ≡
case XBASE:     /* original prefixed string */
  c ← *sp++;
  switch (c) {
  case EOS: c ← 0; break;
  case CHAR: c ← *sp++; break;
  case QCHAR: quote |= 2; c ← *sp++; break;     /* temporary quote */
  case OQUOTE: ⟨ Expand an OQUOTE character 569 ⟩ continue;
  case CQUOTE: quote ← 0; continue;
  case COMSUB: ⟨ Expand a COMSUB character 570 ⟩ continue;
  case EXPRSUB: ⟨ Expand an EXPRSUB character 571 ⟩ continue;
  case OSUBST:
    {
      ⟨ Expand an OSUBST character 572 ⟩ continue;     /* ${{#}⟨var⟩{:}[=+-?#%]⟨word⟩} */
    }
  case CSUBST:
    {
      ⟨ Expand a CSUBST character 579 ⟩ continue;     /* only get here if expanding word */
    }
  case OPAT:     /* open pattern: *(foo|bar) */     /* Next char is the type of pattern */
    make_magic ← 1;
    c ← *sp++ + 0x80;
    break;
  case SPAT:     /* pattern separator (|) */
    make_magic ← 1;
    c ← '|';
    break;
  case CPAT:     /* close pattern */
    make_magic ← 1;
    c ← ')';
    break;
  }
```

This code is used in section 565.

569. When expansion in XBASE encounters the beginning of quoted text for the first time it changes the terminating *word* to IFS_QUOTE which will be immediately changted to IFS_WORD in ⟨ Check for end of word or $IFS separation 586 ⟩ after appending the opening character. The lexer will (?) have taken care of making sure each character is a QCHAR rather than a CHAR. It's not clear why. Ah. It's in case of a quoted character that's not really quoted in quotes.

```
⟨ Expand an OQUOTE character 569 ⟩ ≡
  switch (word) {
  case IFS_QUOTE:      /* """something */
    word ← IFS_WORD;
    break;
  case IFS_WORD: break;
  default: word ← IFS_QUOTE;
    break;
  }
  tilde_ok ← 0;
  quote ← 1;
```

This code is used in section 568.

570.

```
⟨ Expand a COMSUB character 570 ⟩ ≡
  tilde_ok ← 0;
  if (f & DONTRUNCOMMAND) {
    word ← IFS_WORD;
    *dp++ ← '$';
    *dp++ ← '(';
    while (*sp ≠ '\0') {
      Xcheck(ds, dp);
      *dp++ ← *sp++;
    }
    *dp++ ← ')';
  }
  else {
    type ← comsub(&x, sp);      /* x recurses on C's stack */
    if (type ≡ XCOM ∧ (f & DOBLANK)) doblank++;
    sp ← strchr(sp, 0) + 1;
    newlines ← 0;
  }
```

This code is used in section 568.

571. ⟨ Expand an EXPRSUB character 571 ⟩ ≡

```
word ← IFS_WORD;
tilde_ok ← 0;
if (f & DONTRUNCOMMAND) {
  *dp++ ← '$';
  *dp++ ← '(';
  *dp++ ← '(';
  while (*sp ≠ '\0') {
    Xcheck(ds, dp);
    *dp++ ← *sp++;
  }
  *dp++ ← ')';
  *dp++ ← ')';
}
else {
  struct tbl v;
  char *p;
  v.flag ← DEFINED | ISSET | INTEGER;
  v.type ← 10;      /* not default */
  v.name[0] ← '\0';
  v_evaluate(&v, substitute(sp, 0), KSH_UNWIND_ERROR, true);
  sp ← strchr(sp, 0) + 1;
  for (p ← str_val(&v); *p; ) {
    Xcheck(ds, dp);
    *dp++ ← *p++;
  }
}
```

This code is used in section 568.

572. Format is: `OSUBST '[{X]'` ⟨*plain-variable-part*⟩ `'\0'` ⟨*compiled-word-part*⟩ `CSUBST '[}X]'`. This is where all syntax checking gets done.

```
⟨ Expand an OSUBST character 572 ⟩ ≡
  char *varname ← ++sp;      /* skip the { or X */
  int stype;
  int slen ← 0;
  sp ← strchr(sp, '\0') + 1;      /* skip variable */
  type ← varsub(&x, varname, sp, &stype, &slen);
  if (type < 0) {⟨ Report a bad substitution 578 ⟩}
  if (f & DOBLANK) doblank++;
  tilde_ok ← 0;
  if (word ≡ IFS_QUOTE ∧ type ≠ XNULLSUB) word ← IFS_WORD;
  if (type ≡ XBASE) {      /* expand? */
    ⟨ Fill in the (maybe allocated) next/nested substitution 574 ⟩
    if (stype) sp += slen;      /* skip qualifier(s) */
    switch (stype & 0x7f) {
    case '#': case '%': ⟨ Substitute trims 576 ⟩ break;
    case '=': ⟨ Substitute substitutions 577 ⟩ break;
    case '?': f &= ~DOBLANK; f |= DOTEMP_;
    default:      /* '-', '+' and, from FALLTHROUGH, '?' */
      if (quote) word ← IFS_WORD;
      else if (dp ≡ Xstring(ds, dp)) word ← IFS_IWS;
      tilde_ok ← 1;      /* Enable tilde expansion */
      f |= DOTILDE;
    }
  }
  else sp ← (char *) wdscan(sp, CSUBST);      /* scan to (past) next CSUBST */
```

This code is cited in section 594.

This code is used in section 568.

573. If there is a nested substitution, eg. "`${foo:=$bar}`", the stack detailing the substitution type is held in a list of **SubType** objects.

```
⟨ Type definitions 17 ⟩ +≡
  typedef struct SubType {
    short stype;      /* [=+-?%#] action after expanded word */
    short base;      /* begin position of expanded word */
    short f;      /* saved value of f (DOPAT, etc.) */
    struct tbl *var;      /* variable for ${var...} */
    short quote;      /* saved value of quote (for ${...[%#=?]...}) */
    struct SubType *prev;      /* old type */
    struct SubType *next;      /* poped type (to avoid re-allocating) (popped? -ed) */
  } SubType;
```

574. ⟨ Fill in the (maybe allocated) next/nested substitution 574 ⟩ ≡

```
if (¬st→next) {
  SubType *newst;
  newst ← alloc(sizeof(SubType), ATEMP);
  newst→next ← Λ;
  newst→prev ← st;
  st→next ← newst;
}
st ← st→next;
st→stype ← stype;
st→base ← Xsavepos(ds, dp);
st→f ← f;
st→var ← varcpy(x.var);
st→quote ← quote;
```

This code is used in section 572.

575. This is the only place that uses *varcpy*. Copies a variable if it's marked as read-only, which have static storage and so only one can be referenced at a time. This is necessary in order to allow variable expansion expressions to refer to multiple read-only variables.

⟨ eval.c 554 ⟩ +≡

```
static struct tbl *varcpy(struct tbl *vp)
{
  struct tbl *cpy;
  if (vp ≡ Λ ∨ (vp→flag & RDONLY) ≡ 0) return vp;
  cpy ← alloc(sizeof(struct tbl), ATEMP);
  memcpy(cpy, vp, sizeof(struct tbl));
  return cpy;
}
```

576. ⟨ Substitute trims 576 ⟩ ≡

```
f ← DOPAT | (f & DONTRUNCOMMAND) | DOTEMP_;      /* ie. ¬DOBLANK|DOBRACE_|DOTILDE */
quote ← 0;      /* Prepend open pattern (so | in a trim will work as expected) */
*dp++ ← MAGIC;
*dp++ ← '@' + 0x80U;
break;
```

This code is used in section 572.

577. ⟨ Substitute substitutions 577 ⟩ ≡

```
/* Enabling tilde expansion after :'s here is * non-standard ksh, but is consistent with rules for *
   other assignments. Not sure what POSIX thinks of * this. Not doing tilde expansion for integer *
   variables is a non-POSIX thing—makes sense though, * since ~ is an arithmetic operator. */
if (¬(x.var→flag & INTEGER)) f |= DOASNTILDE | DOTILDE;
f |= DOTEMP_;      /* These will be done after the * value has been assigned. */
f &= ∼(DOBLANK | DOGLOB | DOBRACE_);
tilde_ok ← 1;
```

This code is used in section 572.

578. ⟨ Report a bad substitution 578 ⟩ ≡

```
char endc;
char *str, *end;
sp ← varname − 2;  /* restore sp */
end ← (char *) wdscan(sp, CSUBST);  /* the } or X is already skipped */
endc ← *end;
*end ← EOS;
str ← snptreef(Λ, 64, "%S", sp);
*end ← endc;
errorf("%s: bad substitution", str);
```

This code is used in section 572.

579. ⟨ Expand a CSUBST character 579 ⟩ ≡

```
sp++;  /* skip the } or X */
tilde_ok ← 0;  /* in case of ${⟨unset⟩:-} */
*dp ← '\0';
quote ← st→quote;
f ← st→f;
if (f & DOBLANK) doblank−−;
switch (st→stype & 0x7f) {
case '#': case '%': ⟨ Append end-pattern 580 ⟩ continue;
case '=': ⟨ Apply an assignment substitution 581 ⟩ continue;
case '?':
  {
    char *s ← Xrestpos(ds, dp, st→base);
    errorf("%s: %s", st→var→name, dp ≡ s ? "parameter null or not set" : (debunk(s, s,
        strlen(s) + 1), s));
  }
}
st ← st→prev;
type ← XBASE;
```

This code is used in section 568.

580. ⟨ Append end-pattern 580 ⟩ ≡

```
*dp++ ← MAGIC;
*dp++ ← ')';
*dp ← '\0';
dp ← Xrestpos(ds, dp, st→base);
x.str ← trimsub(str_val(st→var), dp, st→stype);      /* Must use st→var, again */
if (x.str[0] ≠ '\0') {
  word ← IFS_IWS;
  type ← XSUB;
}
else if (quote) {
  word ← IFS_WORD;
  type ← XSUB;
}
else {
  if (dp ≡ Xstring(ds, dp)) word ← IFS_IWS;
  type ← XNULLSUB;
}
if (f & DOBLANK) doblank++;
st ← st→prev;
```

This code is used in section 579.

581. ⟨ Apply an assignment substitution 581 ⟩ ≡ /* Restore our position and substitute the value of *st→var* (may not be the assigned value in the presence of integer/right-adj/etc attributes). */

```
dp ← Xrestpos(ds, dp, st→base);      /* Must use st→var since calling global would cause with things like x[i+=1] to be evaluated twice. */      /* Note: not exported by FEXPORT in AT&T ksh. */
  /* XXX POSIX says readonly is only fatal for special builtins (setstr does readonly check). */
len ← strlen(dp) + 1;
setstr(st→var, debunk(alloc(len, ATEMP), dp, len), KSH_UNWIND_ERROR);
x.str ← str_val(st→var);
type ← XSUB;
if (f & DOBLANK) doblank++;
st ← st→prev;
if (quote ∨ ¬*x.str) word ← IFS_WORD;
else word ← IFS_IWS;
```

This code is used in section 579.

582. ⟨ Expand **switch** (*type*) ≡ XNULLSUB 582 ⟩ ≡

```
case XNULLSUB:      /* Special case for "$@" (and "${foo[@]}")—no word is generated if $# is 0 (unless there is other stuff inside the quotes). */
  type ← XBASE;
  if (f & DOBLANK) {
    doblank--;
    if (dp ≡ Xstring(ds, dp) ∧ word ≠ IFS_WORD) word ← IFS_IWS;
  }
```

This code is used in section 565.

583. ⟨ Expand **switch** (*type*) ≡ (XSUB ∨ XSUBMID) 583 ⟩ ≡

```
case XSUB: case XSUBMID:
  if ((c ← *x.str++) ≡ 0) {
    type ← XBASE;
    if (f & DOBLANK) doblank--;
    continue;
  }
```

This code is used in section 565.

584. ⟨ Expand **switch** (*type*) ≡ (XARGSEP ∨ XARG) 584 ⟩ ≡

```
case XARGSEP: type ← XARG;
  quote ← 1;
case XARG:
  if ((c ← *x.str++) ≡ '\0') {
      /* force null words to be created so * set -- '' 2 ''; foo "$@" will do * the right thing */
    if (quote ∧ x.split) word ← IFS_WORD;
    if ((x.str ← *x.u.strv++) ≡ Λ) {
      type ← XBASE;
      if (f & DOBLANK) doblank--;
      continue;
    }
    c ← ifs0;
    if (c ≡ 0) {
      if (quote ∧ ¬x.split) continue;
      if (¬quote ∧ word ≡ IFS_WS) continue;
      c ← ' ';      /* this is so we don't terminate */
      goto emit_word;      /* now force-emit a word; see ⟨ Emit a complete word 587 ⟩ */
    }
    if (quote ∧ x.split) {      /* terminate word for $@ */
      type ← XARGSEP;
      quote ← 0;
    }
  }
```

This code is used in section 565.

585. ⟨ Expand **switch** (*type*) ≡ XCOM 585 ⟩ ≡

```
case XCOM:
  if (x.u.shf ≡ Λ)      /* $(⟨ ... ) failed, fake EOF */
    c ← EOF;
  else if (newlines) {      /* Spit out saved nl's */
    c ← '\n';
    −−newlines;
  }
  else {
    while ((c ← shf_getc(x.u.shf)) ≡ 0 ∨ c ≡ '\n')
      if (c ≡ '\n') newlines++;      /* Save newlines */
    if (newlines ∧ c ≠ EOF) {
      shf_ungetc(c, x.u.shf);
      c ← '\n';
      −−newlines;
    }
  }
  if (c ≡ EOF) {
    newlines ← 0;
    if (x.u.shf ≠ Λ) shf_close(x.u.shf);
    if (x.split) subst_exstat ← waitlast( );
    else subst_exstat ← (x.u.shf ≡ Λ);
    type ← XBASE;
    if (f & DOBLANK) doblank−−;
    continue;
  }
```

This code is used in section 565.

586. WS == WORD, SEPARATING?

/* How words are broken up: * |value of c * word |ws nws 0 * ——————————— * IFS_WORD w/WS w/NWS w * IFS_WS -/WS w/NWS - * IFS_NWS -/NWS w/NWS - * IFS_IWS -/WS w/NWS - * (w means generate a word) */

"If the word has ended ($c \equiv 0$) or we're not in a quote, are processing blanks, are not processing magic and the current character is one of $IFS" and "If *word* is IFS_WORD or IFS_QUOTE, or if there is a character, word is IFS_IWS/IFS_NWS and the character is is not one of ␣, ~~␤~~ or ~~TAB~~."

Only the first $IFS character terminates a word. Subsequent $IFS characters which are exactly ␣, ~~␤~~ & ~~TAB~~ characters are ignored while those which aren't terminate a new (empty) word.

```
#define IFS_WORD 0      /* word has chars (or quotes) */
#define IFS_WS 1        /* have seen IFS white-space */
#define IFS_NWS 2       /* have seen IFS non-white-space */
#define IFS_IWS 3       /* beginning of word, ignore IFS white-space */
#define IFS_QUOTE 4     /* beginning of word with quote, becomes IFS_WORD unless "\"$@\"" */
```

⟨ Check for end of word or $IFS separation 586 ⟩ ≡

```
if (c ≡ 0 ∨ (¬quote
        ∧ (f & DOBLANK) ∧ doblank ∧ ¬make_magic ∧ ctype(c, C_IFS))) {
  if ((word ≡ IFS_WORD) ∨ (word ≡ IFS_QUOTE) ∨ (c ∧
        (word ≡ IFS_IWS ∨ word ≡ IFS_NWS) ∧
        ¬ctype(c, C_IFSWS))) {
    ⟨ Emit a complete word 587 ⟩
  }
  if (c ≡ 0) goto done;
  if (word ≠ IFS_NWS) word ← ctype(c, C_IFSWS) ? IFS_WS : IFS_NWS;
}
else {
  if (type ≡ XSUB) {      /* expanding substitution -ed */
    if (word ≡ IFS_NWS ∧ Xlength(ds, dp) ≡ 0) {      /* append an empty string */
      char *p;
      if ((p ← strdup("")) ≡ Λ) internal_errorf("unable␣""to␣allocate␣memory");
      XPput(*wp, p);
    }
    type ← XSUBMID;
  }
  tilde_ok ≪= 1;      /* age tilde_ok info—~ code tests second bit */
  if (¬quote)      /* mark any special second pass chars */
    switch (c) {
    case '[': case '!': case '-': case ']': ⟨ Mark character class characters 588 ⟩
    case '*': case '?': ⟨ Mark glob characters 589 ⟩
    case OBRACE: case ',': case CBRACE: ⟨ Mark brace expansion characters 590 ⟩
    case '=': ⟨ Mark first unquoted = for ~ 591 ⟩
    case ':': ⟨ Mark first unquoted : for ~ 592 ⟩
    case '~': ⟨ Reset tilde_ok 593 ⟩
    }
  else quote &= ~2;      /* undo temporary */
  if (make_magic) {
    make_magic ← 0;
    fdo |= DOMAGIC_ | (f & DOGLOB);
    *dp++ ← MAGIC;
  }
  else if (ISMAGIC(c)) {
```

```
        fdo |= DOMAGIC_;
        *dp++ ← MAGIC;
      }
      *dp++ ← c;      /* save output char */
      word ← IFS_WORD;
    }
```

This code is cited in sections 561 and 569.

This code is used in section 565.

587. ⟨ Emit a complete word 587 ⟩ ≡

```
  char *p;
emit_word: *dp++ ← '\0';
  p ← Xclose(ds, dp);
  if (fdo & DOBRACE_)      /* also does globbing */
    alt_expand(wp, p, p, p + Xlength(ds, (dp − 1)), fdo | (f & DOMARKDIRS));
  else if (fdo & DOGLOB) glob(p, wp, f & DOMARKDIRS);
  else if ((f & DOPAT) ∨ ¬(fdo & DOMAGIC_)) XPput(*wp, p);
  else XPput(*wp, debunk(p, p, strlen(p) + 1));
  fdo ← 0;
  saw_eq ← 0;
  tilde_ok ← (f & (DOTILDE | DOASNTILDE)) ? 1 : 0;
  if (c ≠ 0) Xinit(ds, dp, 128, ATEMP);
```

This code is cited in section 584.

This code is used in section 586.

588. ⟨ Mark character class characters 588 ⟩ ≡

```
    /* For character classes—doesn't hurt * to have magic [!-]]'s outside of * [ ... ] expressions. */
  if (f & (DOPAT | DOGLOB)) {
    fdo |= DOMAGIC_;
    if (c ≡ '[') fdo |= f & DOGLOB;
    *dp++ ← MAGIC;
  }
  break;
```

This code is used in section 586.

589. ⟨ Mark glob characters 589 ⟩ ≡

```
  if (f & (DOPAT | DOGLOB)) {
    fdo |= DOMAGIC_ | (f & DOGLOB);
    *dp++ ← MAGIC;
  }
  break;
```

This code is used in section 586.

590. ⟨ Mark brace expansion characters 590 ⟩ ≡

```
  if ((f & DOBRACE_) ∧ (c ≡ OBRACE ∨ (fdo & DOBRACE_))) {
    fdo |= DOBRACE_ | DOMAGIC_;
    *dp++ ← MAGIC;
  }
  break;
```

This code is used in section 586.

591. ⟨ Mark first unquoted = for ~ 591 ⟩ ≡ /* Note first unquoted = for ~ */

```
if (¬(f & DOTEMP_) ∧ ¬saw_eq) {
  saw_eq ← 1;
  tilde_ok ← 1;
}
break;
```

This code is used in section 586.

592. ⟨ Mark first unquoted : for ~ 592 ⟩ ≡ /* Note unquoted : for ~ */

```
if (¬(f & DOTEMP_) ∧ (f & DOASNTILDE)) tilde_ok ← 1;
break;
```

This code is used in section 586.

593. ⟨ Reset *tilde_ok* 593 ⟩ ≡ /* *tilde_ok* is reset whenever * any of `'" $( $(( ${ }` are seen. * Note that *tilde_ok* must be preserved * through the sequence `${A=a=}~` */

```
if (type ≡ XBASE ∧ (f & (DOTILDE | DOASNTILDE)) ∧ (tilde_ok & 2)) {
  char *p, *dp_x;
  dp_x ← dp;
  p ← maybe_expand_tilde(sp, &ds, &dp_x, f & DOASNTILDE);
  if (p) {
    if (dp ≠ dp_x) word ← IFS_WORD;
    dp ← dp_x;
    sp ← p;
    continue;
  }
}
break;
```

This code is used in section 586.

594. What was *sp* is now *word*. *sp* is *varname* from ⟨ Expand an OSUBST character 572 ⟩. As originally noted this was not a good choice.

```
⟨ eval.c 554 ⟩ +≡
  static int varsub(Expand *xp, char *sp, char *word, int *stypep, int *slenp)
  {
    int c;
    int state;      /* next state: XBASE, XARG, XSUB, XNULLSUB */
    int stype;      /* substitution type */
    int slen;       /* substitution length (ie. of =, :=, #, etc.) */
    char *p;
    struct tbl *vp;
    int zero_ok ← 0;
    if (sp[0] ≡ '\0') return −1;      /* Bad variable name */
    xp→var ← Λ;
    if (sp[0] ≡ '#' ∧ (c ← sp[1]) ≠ '\0') {
      if (*word ≠ CSUBST) return −1;      /* Can't have any qualifiers for ${#⟨var⟩} */
      ⟨ Expand to the length of a string or array 595 ⟩
      return XSUB;
    }
    ⟨ Check for qualifiers in word part 596 ⟩
    c ← sp[0];
    if (c ≡ '*' ∨ c ≡ '@') {⟨ Expand $@/$* 597 ⟩}
    else {
      if ((p ← strchr(sp, '[')) ∧
          (p[1] ≡ '*' ∨ p[1] ≡ '@') ∧
          p[2] ≡ ']') {⟨ Expand an array 598 ⟩}
      else {⟨ Expand a variable 599 ⟩}
    }
    c ← stype & 0x7f;      /* test the compiler's code generator */
    if (ctype(c, C_SUBOP2) ∨
      /* this does the "if set replace or if not set replace" logic of ${x:+y} and ${x:-y} */
    (((stype & 0x80) ? *xp→str ≡ '\0' : xp→str ≡ null) ?      /* undef? */
    c ≡ '=' ∨ c ≡ '-' ∨ c ≡ '?' : c ≡ '+')) state ← XBASE;
        /* expand word instead of variable value */
    if (Flag(FNOUNSET) ∧ xp→str ≡ null ∧ ¬zero_ok ∧ (ctype(c, C_SUBOP2) ∨ (state ≠ XBASE ∧ c ≠ '+')))
      errorf("%s: parameter not set", sp);
    return state;
  }
```

595. ⟨ Expand to the length of a string or array 595 ⟩ ≡

```
sp++;
if ((p ← strchr(sp, '[')) ∧
      (p[1] ≡ '*' ∨ p[1] ≡ '@') ∧
      p[2] ≡ ']') {      /* Checking for the size of an array */
  int n ← 0;
  vp ← global(arrayname(sp));
  if (vp→flag & (ISSET | ARRAY)) zero_ok ← 1;
  for ( ; vp; vp ← vp→u.array)
    if (vp→flag & ISSET) n++;
  c ← n;      /* ksh88/ksh93 go for number, not max index */
}
else if (c ≡ '*' ∨ c ≡ '@') c ← genv→loc→argc;
else {
  p ← str_val(global(sp));
  zero_ok ← p ≠ null;
  c ← strlen(p);
}
if (Flag(FNOUNSET) ∧ c ≡ 0 ∧ ¬zero_ok) errorf("%s: parameter not set", sp);
*stypep ← 0;      /* unqualified variable/string substitution */
xp→str ← str_save(u64ton((uint64_t) c, 10), ATEMP);
```

This code is used in section 594.

596. If there is none *word*[0] will be CSUBST. Saves the result in *stypep* & *slenp* for the caller.

⟨ Check for qualifiers in *word* part 596 ⟩ ≡

```
stype ← 0;
c ← word[slen ← 0] ≡ CHAR ? word[1] : 0;
if (c ≡ ':') {
  slen += 2;
  stype ← 0x80;
  c ← word[slen + 0] ≡ CHAR ? word[slen + 1] : 0;
}
if (ctype(c, C_SUBOP1)) {
  slen += 2;
  stype |= c;      /* keeps : bit */
}
else if (ctype(c, C_SUBOP2)) {      /* Note: ksh88 allows :%, :%%, etc */
  slen += 2;
  stype ← c;      /* loses : bit—${x:#y} ≠ ${x##y} */
  if (word[slen + 0] ≡ CHAR ∧ c ≡ word[slen + 1]) {
    stype |= 0x80;
    slen += 2;
  }
}
else if (stype) return −1;      /* plain : is not ok */
if (¬stype ∧ *word ≠ CSUBST) return −1;
*stypep ← stype;
*slenp ← slen;
```

This code is used in section 594.

597. ⟨ Expand `$@/$*` 597 ⟩ ≡

```
switch (stype & 0x7f) {
case '=':     /* can't assign to a vector */
case '%':     /* can't trim a vector (yet) */
case '#': return -1;
}
if (genv->loc->argc == 0) {
  xp->str = null;
  xp->var = global(sp);
  state = c == '@' ? XNULLSUB : XSUB;
}
else {
  xp->u.strv = (const char **) genv->loc->argv + 1;
  xp->str = *xp->u.strv++;
  xp->split = c == '@';     /* $@ */
  state = XARG;
}
zero_ok = 1;     /* exempt "$@" and "$*" from "set -u" */
```

This code is used in section 594.

598. Somehow notes that the contents of x/xp are an array, this similarly appends a list of strings built in wv.

⟨ Expand an array 598 ⟩ ≡

```
XPtrV wv;
switch (stype & 0x7f) {
case '=':     /* can't assign to a vector */
case '%':     /* can't trim a vector (yet) */
case '#':
case '?': return -1;
}
XPinit(wv, 32);
vp = global(arrayname(sp));
for ( ; vp; vp = vp->u.array) {
  if (¬(vp->flag & ISSET)) continue;
  XPput(wv, str_val(vp));
}
if (XPsize(wv) == 0) {
  xp->str = null;
  state = p[1] == '@' ? XNULLSUB : XSUB;
  XPfree(wv);
}
else {
  XPput(wv, 0);
  xp->u.strv = (const char **) XPptrv(wv);
  xp->str = *xp->u.strv++;
  xp->split = p[1] == '@';     /* ${foo[@]} */
  state = XARG;
}
```

This code is used in section 594.

599. ⟨ Expand a variable 599 ⟩ ≡ /* Can't assign things like $! or $1 */
```
if ((stype & 0x7f) ≡ '=' ∧ (ctype(*sp, C_VAR1) ∨ digit(*sp))) return −1;
xp→var ← global(sp);
xp→str ← str_val(xp→var);
state ← XSUB;
```
This code is used in section 594.

600. Run the command in "$(...)" and read its output.

⟨ eval.c 554 ⟩ +≡
```
static int comsub(Expand *xp, char *cp)
{
  Source *s, *sold;
  struct Op *t;
  struct Shf *shf;
  s ← pushs(SSTRING, ATEMP);
  s→start ← s→str ← cp;
  sold ← source;
  t ← compile(s);
  afree(s, ATEMP);
  source ← sold;
  if (t ≡ Λ) return XBASE;
  if (t ≠ Λ ∧ t→type ≡ TCOM ∧ *t→args ≡ Λ ∧ *t→vars ≡ Λ ∧ t→ioact ≠ Λ) {   /* "$(<⟨file⟩)" */
    struct ioword *io ← *t→ioact;
    char *name;
    if ((io→flag & IOTYPE) ≠ IOREAD) errorf("funny␣$()␣command:␣%s", snptreef(Λ, 32, "%R", io));
    shf ← shf_open(name ← evalstr(io→name, DOTILDE), O_RDONLY, 0, SHF_MAPHI | SHF_CLEXEC);
    if (shf ≡ Λ) warningf(¬Flag(FTALKING), "%s:␣cannot␣open␣$(<)␣input", name);
    xp→split ← 0;   /* no waitlast */
  }
  else {
    int ofd1, pv[2];
    openpipe(pv);
    shf ← shf_fdopen(pv[0], SHF_RD, Λ);
    ofd1 ← savefd(1);
    if (pv[1] ≠ 1) {
      ksh_dup2(pv[1], 1, false);
      close(pv[1]);
    }
    execute(t, XFORK | XXCOM | XPIPEO, Λ);
    restfd(1, ofd1);
    startlast( );
    xp→split ← 1;   /* waitlast */
  }
  xp→u.shf ← shf;
  return XCOM;
}
```

601. Have a trim.

```
⟨ eval.c 554 ⟩ +≡
  static char *trimsub(char *str, char *pat, int how)
  {
    char *end ← strchr(str, 0);
    char *p, c;
    switch (how & 0xff) {      /* UCHAR_MAX maybe? */
    case '#': ⟨ Trim the shortest match at the beginning (#) and break 602 ⟩
    case '#' | 0x80: ⟨ Trim the longest match at the beginning (##) and break 603 ⟩
    case '%': ⟨ Trim the shortest match at the end (%) and break 604 ⟩
    case '%' | 0x80: ⟨ Trim the longest match at the end (%%) and break 605 ⟩
    }
    return str;      /* no match, return string */
  }
```

602. ⟨ Trim the shortest match at the beginning (#) and **break** 602 ⟩ ≡

```
  for (p ← str; p ≤ end; p++) {
    c ← *p;
    *p ← '\0';
    if (gmatch(str, pat, false)) {
      *p ← c;
      return p;
    }
    *p ← c;
  }
  break;
```

This code is used in section 601.

603. ⟨ Trim the longest match at the beginning (##) and **break** 603 ⟩ ≡

```
  for (p ← end; p ≥ str; p--) {
    c ← *p;
    *p ← '\0';
    if (gmatch(str, pat, false)) {
      *p ← c;
      return p;
    }
    *p ← c;
  }
  break;
```

This code is used in section 601.

604. ⟨ Trim the shortest match at the end (%) and **break** 604 ⟩ ≡

```
  for (p ← end; p ≥ str; p--) {
    if (gmatch(p, pat, false)) return str_nsave(str, p − str, ATEMP);
  }
  break;
```

This code is used in section 601.

605. ⟨ Trim the longest match at the end (`%%`) and **break** 605 ⟩ ≡

```
for (p ← str; p ≤ end; p++) {
  if (gmatch(p, pat, false)) return str_nsave(str, p − str, ATEMP);
}
break;
```

This code is used in section 601.

606. Alternation.

```
⟨ eval.c 554 ⟩ +≡
  static void alt_expand(XPtrV *wp, char *start, char *exp_start, char *end, int fdo)
  {
    int count ← 0;
    char *brace_start, *brace_end, *comma ← Λ;
    char *field_start;
    char *p;
    for (p ← exp_start; (p ← strchr(p, MAGIC)) ∧ p[1] ≠ OBRACE; p += 2) ;      /* search for open brace */
    brace_start ← p;
    if (p) {⟨ Find a matching close brace if any 607 ⟩}
    if (¬p ∨ count ≠ 0) {⟨ No valid alternation expansion 609 ⟩}
    brace_end ← p;
    if (¬comma) {
      alt_expand(wp, start, brace_end, end, fdo);
      return;
    }
    ⟨ Expand alternate expressions 610 ⟩
    return;
  }
```

607. ⟨ Find a matching close brace if any 607 ⟩ ≡

```
  comma ← Λ;
  count ← 1;
  for (p += 2; *p ∧ count; p++) {
    if (ISMAGIC(*p)) {
      if (*++p ≡ OBRACE) count++;
      else if (*p ≡ CBRACE) --count;
      else if (*p ≡ ',' ∧ count ≡ 1) comma ← p;
    }
  }
```

This code is used in section 606.

608. Note that given "`a{{b,c}`" we do not expand anything (this is what AT&T ksh does though this may be changed to do the "`{b,c}`" expansion).

609. ⟨ No valid alternation expansion 609 ⟩ ≡

```
  if (fdo & DOGLOB) glob(start, wp, fdo & DOMARKDIRS);
  else XPput(*wp, debunk(start, start, end − start));
  return;
```

This code is used in section 606.

610. ⟨ Expand alternate expressions 610 ⟩ ≡

```
field_start ← brace_start + 2;
count ← 1;
for (p ← brace_start + 2; p ≠ brace_end; p++) {
  if (ISMAGIC(*p)) {
    if (*++p ≡ OBRACE) count++;
    else if ((*p ≡ CBRACE ∧ --count ≡ 0) ∨ (*p ≡ ',' ∧ count ≡ 1)) {
      char *new;
      int l1, l2, l3;
      l1 ← brace_start − start;
      l2 ← (p − 1) − field_start;
      l3 ← end − brace_end;
      new ← alloc(l1 + l2 + l3 + 1, ATEMP);
      memcpy(new, start, l1);
      memcpy(new + l1, field_start, l2);
      memcpy(new + l1 + l2, brace_end, l3);
      new[l1 + l2 + l3] ← '\0';
      alt_expand(wp, new, new + l1, new + l1 + l2 + l3, fdo);
      field_start ← p + 1;
    }
  }
}
```

This code is used in section 606.

611. Tilde Expansion. Check if p is an unquoted name, possibly followed by "[/:]". If so puts the expanded version in $*dsp/dp$ and returns a pointer in p just past the name, otherwise returns 0.

```
⟨ eval.c 554 ⟩ +≡
  static char *maybe_expand_tilde(char *p, XString *dsp, char **dpp, int isassign)
  {
    XString ts;
    char *dp ← *dpp;
    char *tp, *r;
    Xinit(ts, tp, 16, ATEMP);
    while (p[0] ≡ CHAR ∧ p[1] ≠ '/' ∧ (¬isassign ∨ p[1] ≠ ':')) {      /* ":" only for DOASNTILDE form */
      Xcheck(ts, tp);
      *tp++ ← p[1];
      p += 2;
    }
    *tp ← '\0';
    r ← (p[0] ≡ EOS ∨ p[0] ≡ CHAR ∨ p[0] ≡ CSUBST) ? tilde(Xstring(ts, tp)) : Λ;
    Xfree(ts, tp);
    if (r) {
      while (*r) {
        Xcheck(*dsp, dp);
        if (ISMAGIC(*r)) *dp++ ← MAGIC;
        *dp++ ← *r++;
      }
      *dpp ← dp;
      r ← p;
    }
    return r;
  }
```

612. Tilde expansion based on a version by Arnold Robbins.

```
⟨ eval.c 554 ⟩ +≡
  static char *tilde(char *cp)
  {
    char *dp;
    if (cp[0] ≡ '\0') dp ← str_val(global("HOME"));
    else if (cp[0] ≡ '+' ∧ cp[1] ≡ '\0') dp ← str_val(global("PWD"));
    else if (cp[0] ≡ '-' ∧ cp[1] ≡ '\0') dp ← str_val(global("OLDPWD"));
    else dp ← homedir(cp);
    if (dp ≡ null) dp ← Λ;      /* If $HOME, $PWD or $OLDPWD are not set don't expand "~". */
    return dp;
  }
```

613. Map userid to user's home directory. Note that 4.3's getpw adds more than 6K to the shell and the YP version probably adds much more. We might consider our own version of getpwnam() to keep the size down.

```
⟨ eval.c 554 ⟩ +≡
  static char *homedir(char *name)
  {
    struct tbl *ap;
    ap ← ktenter(&homedirs, name, hash(name));
    if (¬(ap→flag & ISSET)) {
      struct passwd *pw;
      pw ← getpwnam(name);
      if (pw ≡ Λ) return Λ;
      ap→val.s ← str_save(pw→pw_dir, APERM);
      ap→flag |= DEFINED | ISSET | ALLOC;
    }
    return ap→val.s;
  }
```

614. Pattern Matching. Was `gmatch.c`, moved to `misc.c`. Match a pattern as in sh(1). Pattern characters are prefixed with `MAGIC` by *expand*.

isfile is false iff no syntax check has been done on the pattern. If the check fails do a straight *strcmp*.

```
⟨ misc.c 9 ⟩ +≡
  int gmatch(const char *s, const char *p, int isfile)
  {
    const char *se, *pe;
    if (s ≡ Λ ∨ p ≡ Λ) return 0;
    se ← s + strlen(s);
    pe ← p + strlen(p);
    if (¬isfile ∧ ¬has_globbing(p, pe)) {
      size_t len ← pe − p + 1;
      char tbuf[64];
      char *t ← len ≤ sizeof (tbuf) ? tbuf : alloc(len, ATEMP);
      debunk(t, p, len);
      return ¬strcmp(t, s);
    }
    return do_gmatch(
        (const unsigned char *) s,
        (const unsigned char *) se,
        (const unsigned char *) p,
        (const unsigned char *) pe);
  }
```

615. Must return either 0 or 1 (assumed by code for "0x80 | '!'").

Pattern-matching ("[*+?@!](...)")) is not protected against inclusion by KSH as it's required when trimming an expansion ("${...[#%]...}").

```
⟨ misc.c 9 ⟩ +≡
  static int do_gmatch(const unsigned char *s, const unsigned char *se,
          const unsigned char *p, const unsigned char *pe)
  {
    int sc, pc;
    const unsigned char *prest, *psub, *pnext;
    const unsigned char *srest;
    if (s ≡ Λ ∨ p ≡ Λ) return 0;
    while (p < pe) {
      pc ← *p++;
      sc ← s < se ? *s : '\0';
      s++;
      if (¬ISMAGIC(pc)) {
        if (sc ≠ pc) return 0;
        continue;
      }
      switch (*p++) {
      case '[': if (sc ≡ 0 ∨ (p ← cclass(p, sc)) ≡ Λ) return 0;
        break;
      case '?': if (sc ≡ 0) return 0;
        break;
      case '*': ⟨ Match a globbing "*" and return 616 ⟩
      case 0x80 | '+': case 0x80 | '*': ⟨ Match 0/1 ("*"/"+") or more times and return 617 ⟩
      case 0x80 | '?': case 0x80 | '@':
      case 0x80 | '␣': /* "␣" is a simile for "@" when a trimming ("[#%]") */
        ⟨ Match once or not ("?") or one of many patterns ("@") and return 618 ⟩
      case 0x80 | '!': ⟨ Match none of the patterns and return 619 ⟩
      default: if (sc ≠ p[−1]) return 0;
        break;
      }
    }
    return s ≡ se;
  }
```

616. ⟨ Match a globbing "*" and return 616 ⟩ ≡

```
  while (ISMAGIC(p[0]) ∧ p[1] ≡ '*') p += 2;    /* collapse consecutive stars */
  if (p ≡ pe) return 1;
  s−−;
  do {
    if (do_gmatch(s, se, p, pe)) return 1;
  } while (s++ < se);
  return 0;
```

This code is used in section 615.

617. ⟨ Match 0/1 ("*"/"+") or more times and **return** 617 ⟩ ≡

```
if (¬(prest ← pat_scan(p, pe, 0))) return 0;
s−−;      /* take care of zero matches */
if (p[−1] ≡ (0x80 | '*') ∧ do_gmatch(s, se, prest, pe)) return 1;
for (psub ← p; ; psub ← pnext) {
  pnext ← pat_scan(psub, pe, 1);
  for (srest ← s; srest ≤ se; srest++) {
    if (do_gmatch(s, srest, psub, pnext − 2) ∧ (do_gmatch(srest, se, prest,
        pe) ∨ (s ≠ srest ∧ do_gmatch(srest, se, p − 2, pe)))) return 1;
  }
  if (pnext ≡ prest) break;
}
return 0;
```

This code is used in section 615.

618. ⟨ Match once or not ("?") or one of many patterns ("@") and **return** 618 ⟩ ≡

```
if (¬(prest ← pat_scan(p, pe, 0))) return 0;
s−−;      /* Take care of zero matches */
if (p[−1] ≡ (0x80 | '?') ∧ do_gmatch(s, se, prest, pe)) return 1;
for (psub ← p; ; psub ← pnext) {
  pnext ← pat_scan(psub, pe, 1);
  srest ← prest ≡ pe ? se : s;
  for ( ; srest ≤ se; srest++) {
    if (do_gmatch(s, srest, psub, pnext − 2) ∧ do_gmatch(srest, se, prest, pe)) return 1;
  }
  if (pnext ≡ prest) break;
}
return 0;
```

This code is used in section 615.

619. ⟨ Match none of the patterns and **return** 619 ⟩ ≡

```
if (¬(prest ← pat_scan(p, pe, 0))) return 0;
s−−;
for (srest ← s; srest ≤ se; srest++) {
  int matched ← 0;
  for (psub ← p; ; psub ← pnext) {
    pnext ← pat_scan(psub, pe, 1);
    if (do_gmatch(s, srest, psub, pnext − 2)) {
      matched ← 1;
      break;
    }
    if (pnext ≡ prest) break;
  }
  if (¬matched ∧ do_gmatch(srest, se, prest, pe)) return 1;
}
return 0;
```

This code is used in section 615.

620. Look for the next "[)|]" (if *match_sep*) in a "`*(foo|bar)`" pattern.

```
⟨ misc.c 9 ⟩ +≡
  const unsigned char *pat_scan(const unsigned char *p, const unsigned char *pe, int match_sep)
  {
    int nest ← 0;
    for ( ; p < pe; p++) {
      if (¬ISMAGIC(*p)) continue;
      if ((*++p ≡ ')' ∧ nest-- ≡ 0) ∨ (*p ≡ '|' ∧ match_sep ∧ nest ≡ 0)) return ++p;
      if ((*p & 0x80) ∧ strchr("*+?@!␣", *p & 0x7f)) nest++;
    }
    return Λ;
  }
```

621.

```
⟨ misc.c 9 ⟩ +≡
  static int posix_cclass(const unsigned char *pattern, int test, const unsigned char **ep)
  {
    const struct CClass *cc;
    const unsigned char *colon;
    size_t len;
    int rval ← 0;
    if ((colon ← strchr(pattern, ':')) ≡ Λ ∨ colon[1] ≠ MAGIC) {
      *ep ← pattern − 2;
      return −1;
    }
    *ep ← colon + 3;        /* skip MAGIC */
    len ← (size_t)(colon − pattern);
    for (cc ← cclasses; cc→name ≠ Λ; cc++) {
      if (¬strncmp(pattern, cc→name, len) ∧ cc→name[len] ≡ '\0') {
        if (cc→isctype(test)) rval ← 1;
        break;
      }
    }
    if (cc→name ≡ Λ) {
      rval ← −2;        /* invalid character class */
    }
    return rval;
  }
```

622. ⟨ misc.c 9 ⟩ +≡

```
static const unsigned char *cclass(const unsigned char *p, int sub)
{
  int c, d, rv, not, found ← 0;
  const unsigned char *orig_p ← p;
  if ((not ← (ISMAGIC(*p) ∧ *++p ≡ '!'))) p++;
  do {      /* check for POSIX character class (eg. "[[:alpha:]]") */
    if ((p[0] ≡ MAGIC ∧ p[1] ≡ '[' ∧ p[2] ≡ ':') ∨ (p[0] ≡ '[' ∧ p[1] ≡ ':')) {
      do {
        const char *pp ← p + (*p ≡ MAGIC) + 2;
        rv ← posix_cclass(pp, sub, &p);
        switch (rv) {
        case 1: found ← 1;
          break;
        case −2: return Λ;
        }
      } while (rv ≠ −1 ∧ p[0] ≡ MAGIC ∧ p[1] ≡ '[' ∧ p[2] ≡ ':');
      if (p[0] ≡ MAGIC ∧ p[1] ≡ ']') break;
    }
    c ← *p++;
    if (ISMAGIC(c)) {
      c ← *p++;
      if ((c & 0x80) ∧ ¬ISMAGIC(c)) {
        c &= 0x7f;      /* extended pattern matching: "*+?@!" */
          /* XXX the "(" char isn't handled as part of "[...]" (? -ed) */
        if (c ≡ '␣') c ← '(';      /* simile for @: plain "(...)" */
      }
    }
    if (c ≡ '\0')      /* No closing "]"—act as if the opening "[" was quoted */
      return sub ≡ '[' ? orig_p : Λ;
    if (ISMAGIC(p[0]) ∧ p[1] ≡ '-' ∧ (¬ISMAGIC(p[2]) ∨ p[3] ≠ ']')) {
      p += 2;      /* "MAGIC-..." */
      d ← *p++;
      if (ISMAGIC(d)) {
        d ← *p++;
        if ((d & 0x80) ∧ ¬ISMAGIC(d)) d &= 0x7f;
      }      /* POSIX says this is an invalid expression */
      if (c > d) return Λ;
    }
    else d ← c;
    if (c ≡ sub ∨ (c ≤ sub ∧ sub ≤ d)) found ← 1;
  } while (¬(ISMAGIC(p[0]) ∧ p[1] ≡ ']'));
  return (found ≠ not) ? p + 2 : Λ;
}
```

623. Subprocess I/O. ie. redirection.

```
⟨ exec.c 461 ⟩ +≡
  static int iosetup(struct ioword *iop, struct tbl *tp)
  {
    int u ← −1;
    char *cp ← iop→name;
    int iotype ← iop→flag & IOTYPE;
    int do_open ← 1, do_close ← 0, flags ← 0;
    struct ioword iotmp;
    struct stat statb;
    if (iotype ≠ IOHERE) cp ← evalonestr(cp, DOTILDE | (Flag(FTALKING_I) ? DOGLOB : 0));
        /* Used for tracing and error messages to print expanded cp */
    iotmp ← *iop;
    iotmp.name ← (iotype ≡ IOHERE) ? Λ : cp;
    iotmp.flag |= IONAMEXP;
    if (Flag(FXTRACE))
      shellf("%s%s\n", PS4_SUBSTITUTE(str_val(global("PS4"))), snptreef(Λ, 32, "%R", &iotmp));
    switch (iotype) {
    case IOREAD: flags ← O_RDONLY; break;
    case IOCAT: flags ← O_WRONLY | O_APPEND | O_CREAT; break;
    case IOWRITE: ⟨ Set up output redirection and break 624 ⟩
    case IORDWR: flags ← O_RDWR | O_CREAT; break;
    case IOHERE: ⟨ Set up redirection from a heredoc and break 625 ⟩
    case IODUP: ⟨ Set up I/O duplication and break 626 ⟩
    }
    if (do_open) {⟨ Open a new file for redirection 627 ⟩}
    if (u < 0) {⟨ Report an error attempting redirection and return 628 ⟩}
    if (genv→savefd[iop→unit] ≡ 0) {⟨ Save a file-descriptor prior to redirection 629 ⟩}
    if (do_close) close(iop→unit);
    else if (u ≠ iop→unit) {⟨ Close freshly duplicated file-descriptors 630 ⟩}
    if (u ≡ 2)      /* Clear any write errors */
      shf_reopen(2, SHF_WR, shl_out);
    return 0;
  }
```

624. **stat** is to allow redirections to things like `/dev/null` without error.

```
⟨ Set up output redirection and break 624 ⟩ ≡
  flags ← O_WRONLY | O_CREAT | O_TRUNC;
  if (Flag(FNOCLOBBER) ∧ ¬(iop→flag & IOCLOB) ∧
        (stat(cp, &statb) ≡ −1 ∨ S_ISREG(statb.st_mode)))
    flags |= O_EXCL;
  break;
```

This code is used in section 623.

625. ⟨ Set up redirection from a heredoc and **break** 625 ⟩ ≡

```
  do_open ← 0;      /* herein returns -2 if an error has been printed */
  u ← herein(iop→heredoc, iop→flag & IOEVAL);      /* cp may have the wrong name (TODO: ? -ed) */
  break;
```

This code is used in section 623.

626. ⟨ Set up I/O duplication and **break** 626 ⟩ ≡

```
{
  const char *emsg;
  do_open ← 0;
  if (*cp ≡ '-' ∧ ¬cp[1]) {
    u ← 1009;      /* prevent error return below */
    do_close ← 1;
  }
  else if ((u ← check_fd(cp, X_OK | ((iop→flag & IORDUP) ? R_OK : W_OK), &emsg)) < 0) {
    warningf(true, "%s: %s", snptreef(Λ, 32, "%R", &iotmp), emsg);
    return −1;
  }
  if (u ≡ iop→unit) return 0;      /* "dup from" ≡ "dup to" */
  break;
}
```

This code is used in section 623.

627. ⟨ Open a new file for redirection 627 ⟩ ≡

```
if (Flag(FRESTRICTED) ∧ (flags & O_CREAT)) {
  warningf(true, "%s: restricted", cp);
  return −1;
}
u ← open(cp, flags, °666);
```

This code is used in section 623.

628. ⟨ Report an error attempting redirection and **return** 628 ⟩ ≡

```
if (u ≡ −1)      /* herein has already printed a message if -2 */
  warningf(true, "cannot %s %s: %s",
      iotype ≡ IODUP ? "dup" : (iotype ≡ IOREAD ∨ iotype ≡ IOHERE) ? "open" : "create", cp,
      strerror(errno));
return −1;
```

This code is used in section 623.

629. Do not save if it has already been redirected (ie. "`cat >x >y`").

c_exec assumes *e→savefd*[*fd*] is set for any redirections. Ask *savefd* not to close *iop→unit*—this allows error messages to be seen if *iop→unit* is 2; also we can't lose the file-descriptor (eg. both *dup2* below and *dup2* in *restfd* failing).

⟨ Save a file-descriptor prior to redirection 629 ⟩ ≡

```
if (u ≡ iop→unit)      /* If these are the same it means unit was already closed */
  genv→savefd[iop→unit] ← −1;
else genv→savefd[iop→unit] ← savefd(iop→unit);
```

This code is used in section 623.

630. ⟨ Close freshly duplicated file-descriptors 630 ⟩ ≡

```
if (ksh_dup2(u, iop→unit, true) < 0) {
  warningf(true, "could not finish (dup) redirection %s: %s",
      snptreef(Λ, 32, "%R", &iotmp),
      strerror(errno));
  if (iotype ≠ IODUP) close(u);
  return −1;
}
if (iotype ≠ IODUP) close(u);
else if (tp ∧ tp→type ≡ CSHELL ∧ tp→val.f ≡ c_exec) {
    /* Touching any co-process file-descriptor in an empty exec! causes the shell to close its copies */
  if (iop→flag & IORDUP) coproc_read_close(u);      /* possible "exec <&p" */
  else coproc_write_close(u);      /* possible "exec >&p" */
}
```

This code is used in section 623.

631. Open heredoc temp file. If the heredoc delimiter is unquoted expand the temp file into a second temp file.

⟨ exec.c 461 ⟩ +≡

```
static int herein(const char *content, int sub)
{
  volatile int fd ← −1;
  struct Source *s, *volatile osource;
  struct Shf *volatile shf;
  struct Temp *h;
  int i;
  if (content ≡ Λ) {      /* "ksh -c 'cat << XYZ'" can cause this */
    warningf(true, "here document missing");
    return −2;      /* special to iosetup: don't print error */
  }
  ⟨ Create temp file to hold heredoc content 632 ⟩
  osource ← source;
  newenv(E_ERRH);
  i ← sigsetjmp(genv→jbuf, 0);
  if (i) {
    source ← osource;
    quitenv(shf);
    close(fd);
    return −2;      /* special to iosetup: don't print error */
  }
  if (sub) {⟨ Do substitutions on the content of the heredoc 633 ⟩}
  else shf_puts(content, shf);
  quitenv(Λ);
  if (shf_close(shf) ≡ EOF) {
    close(fd);
    warningf(true, "error writing %s: %s", h→name, strerror(errno));
    return −2;      /* special to iosetup: don't print error */
  }
  return fd;
}
```

632. This is done before *newenv* so the temp file doesn't get removed too soon.

```
⟨ Create temp file to hold heredoc content 632 ⟩ ≡
  h ← maketemp(ATEMP, TT_HEREDOC_EXP, &genv→temps);
  if (¬(shf ← h→shf) ∨ (fd ← open(h→name, O_RDONLY, 0)) ≡ −1) {
    warningf(true, "can't␣%s␣temporary␣file␣%s:␣%s", ¬shf ? "create" : "open", h→name,
        strerror(errno));
    if (shf) shf_close(shf);
    return −2;      /* special to iosetup: don't print error */
  }
```

This code is used in section 631.

633. ⟨ Do substitutions on the content of the heredoc 633 ⟩ ≡

```
  s ← pushs(SSTRING, ATEMP);
  s→start ← s→str ← content;
  source ← s;
  if (yylex(ONEWORD | HEREDOC) ≠ LWORD) internal_errorf("%s:␣yylex", __func__);
  source ← osource;
  shf_puts(evalstr(yylval.cp, 0), shf);
```

This code is used in section 631.

634. Operating System. Interacting with the operating system, apart from I/O, happens with signals[1]. When a signal is handled by ksh a **Trap** object is created for it with the signal number and ksh command to run when it's received. The flags and signal handlers permit ksh to restore the signal handler state before a call to *fork* and/or *exec*.

```
#define TF_SHELL_USES BIT(0)    /* shell uses signal, user can't change */
#define TF_USER_SET BIT(1)    /* user has (tried to) set trap */
#define TF_ORIG_IGN BIT(2)    /* original action was SIG_IGN */
#define TF_ORIG_DFL BIT(3)    /* original action was SIG_DFL */
#define TF_EXEC_IGN BIT(4)    /* restore SIG_IGN just before exec */
#define TF_EXEC_DFL BIT(5)    /* restore SIG_DFL just before exec */
#define TF_DFL_INTR BIT(6)    /* when received, default action is LINTR */
#define TF_TTY_INTR BIT(7)    /* tty generated signal (see j_waitj) */
#define TF_CHANGED BIT(8)    /* used by runtrap to detect trap changes */
#define TF_FATAL BIT(9)    /* causes termination if not trapped */
```

⟨ Type definitions 17 ⟩ +≡

```
typedef struct Trap {    /* renamed for cosmetic reasons */
  int signal;    /* signal number */
  const char *name;    /* short name */
  const char *mess;    /* descriptive name */
  char *trap;    /* trap command */
  volatile sig_atomic_t set;    /* trap pending */
  int flags;    /* TF_* */
  sig_t cursig;    /* current handler (valid if TF_ORIG_* set) */
  sig_t shtrap;    /* shell signal handler */
} Trap;
```

635. The number of signals NSIG includes zero which is not a real signal[1]. ksh' EXIT pseudo-signal uses this slot instead and ERR the extra slot created at the end of *sigtraps*.

```
#define SIGEXIT_ 0    /* for trap EXIT */
#define SIGERR_ NSIG    /* for trap ERR */
```

⟨ Global variables 5 ⟩ +≡

```
volatile sig_atomic_t trap;
volatile sig_atomic_t intrsig;
volatile sig_atomic_t fatal_trap;
```

636. ⟨ Externally-linked variables 6 ⟩ +≡

```
extern volatile sig_atomic_t trap;    /* traps pending? */
extern volatile sig_atomic_t intrsig;    /* pending trap interrupts command */
extern volatile sig_atomic_t fatal_trap;    /* received a fatal signal */
extern volatile sig_atomic_t got_sigwinch;
extern Trap sigtraps[NSIG + 1];
```

[1] ksh also defines SIGEXIT_ and SIGERR_ which are treated like signals for nearly all purposes except that they are raised by ksh itself—they should be assumed part of the set whenever referring to "signals".

637. ⟨ trap.c 637 ⟩ ≡

```
#include <ctype.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
  Trap sigtraps[NSIG + 1];
  static struct sigaction Sigact_ign , Sigact_trap;
```

See also sections 639, 640, 641, 642, 643, 645, 646, 648, 649, 650, 651, 652, 653, 654, 655, 659, and 660.

638. ⟨ Shared function declarations 4 ⟩ +≡

```
  void inittraps(void);
  void alarm_init(void);
  Trap *gettrap(const char *, int);
  void trapsig(int);
  void intrcheck(void);
  int fatal_trap_check(void);
  int trap_pending(void);
  void runtraps(int intr);
  void runtrap(Trap *);
  void cleartraps(void);
  void restoresigs(void);
  void settrap(Trap *, char *);
  int block_pipe(void);
  void restore_pipe(int);
  int setsig(Trap *, sig_t, int);
  void setexecsig(Trap *, int);
```

639. Signal handling is initialised very early in the lifetime of ksh. First *sigtraps* is populated by filling each slot's long and short name from *sys_siglist* and *sys_signame* respectively. *name* is set for the `EXIT` trap but not the long name which comes from *sys_siglist*. This is probably a bug.

A pair of **sigaction** objects is created for signals which are to be ignored (*Sigact_ign*) or trapped (*Sigact_trap*) by *trapsig*.

The signals `INT`, `QUIT`, `TERM` and `HUP` are always trapped in order to clean up temporary files before exiting. `SIGCHLD` has the bit `TF_SHELL_USES` set so that users are not able to interfere with the shell's handling of subprocesses.

```
⟨ trap.c 637 ⟩ +≡
  void inittraps(void)
  {
    int i;
    for (i ← 0; i ≤ NSIG; i++) {
      sigtraps[i].signal ← i;
      if (i ≡ SIGERR_) {
        sigtraps[i].name ← "ERR";
        sigtraps[i].mess ← "Error␣handler";
      }
      else {
        sigtraps[i].name ← sys_signame[i];
        sigtraps[i].mess ← sys_siglist[i];
      }
    }
    sigtraps[SIGEXIT_].name ← "EXIT";      /* our name for signal 0 */
    sigemptyset(&Sigact_ign.sa_mask);
    Sigact_ign.sa_flags ← 0;      /* interruptible */
    Sigact_ign.sa_handler ← SIG_IGN;
    Sigact_trap ← Sigact_ign;
    Sigact_trap.sa_handler ← trapsig;
    sigtraps[SIGINT].flags |= TF_DFL_INTR | TF_TTY_INTR;
    sigtraps[SIGQUIT].flags |= TF_DFL_INTR | TF_TTY_INTR;
    sigtraps[SIGTERM].flags |= TF_DFL_INTR;      /* not fatal for interactive */
    sigtraps[SIGHUP].flags |= TF_FATAL;
    sigtraps[SIGCHLD].flags |= TF_SHELL_USES;
    setsig(&sigtraps[SIGINT], trapsig, SS_RESTORE_ORIG);
    setsig(&sigtraps[SIGQUIT], trapsig, SS_RESTORE_ORIG);
    setsig(&sigtraps[SIGTERM], trapsig, SS_RESTORE_ORIG);
    setsig(&sigtraps[SIGHUP], trapsig, SS_RESTORE_ORIG);
  }
```

640. When a trapped signal is received *trapsig* will mark that it's been received in its *sigtraps* slot and set flags *intrsig* and *fatal_trap* as appropriate (and *trap*). If an additional signal handler has been saved in *shtrap* then it's invoked—see *setsig*.

```
⟨ trap.c 637 ⟩ +≡
  void trapsig(int i)
  {
    Trap *p ← &sigtraps[i];
    int errno_ ← errno;
    trap ← p→set ← 1;
    if (p→flags & TF_DFL_INTR) intrsig ← 1;
    if ((p→flags & TF_FATAL) ∧ ¬p→trap) {
      fatal_trap ← 1;
      intrsig ← 1;
    }
    if (p→shtrap) (*p→shtrap)(i);
    errno ← errno_;
  }
```

641. The traps in *sigtraps* can be queried by name as well as by number. "SIG" is optional, as is case-sensitivity according to *igncase*.

```
⟨ trap.c 637 ⟩ +≡
  Trap *gettrap(const char *name, int igncase)
  {
    int i;
    Trap *p;
    if (digit(*name)) {      /* name is a number in a string */
      int n;
      if (getn(name, &n) ∧ 0 ≤ n ∧ n < NSIG) return &sigtraps[n];
      return Λ;
    }
    if (igncase ∧ strncasecmp(name, "SIG", 3) ≡ 0) name += 3;
    if (¬igncase ∧ strncmp(name, "SIG", 3) ≡ 0) name += 3;
    for (p ← sigtraps, i ← NSIG + 1; --i ≥ 0; p++)
      if (p→name) {
        if (igncase ∧ strcasecmp(p→name, name) ≡ 0) return p;
        if (¬igncase ∧ strcmp(p→name, name) ≡ 0) return p;
      }
    return Λ;
  }
```

642. A signal's action is defined by a **Trap** object and assigned by *setsig*. If a signal is to remain ignored then no further action is taken otherwise the handler is applied.

There is provision for disabling the additional *shtrap* handler but the only three signals which use it—SIGCHLD, SIGALRM & SIGWINCH—are permanent.

```
⟨ trap.c 637 ⟩ +≡
  int setsig(Trap *p, sig_t f, int flags)
  {
    struct sigaction sigact;
    if (p→signal ≡ SIGEXIT_ ∨ p→signal ≡ SIGERR_) return 1;
    ⟨ Save flags the first time a signal action is set 644 ⟩
    if ((p→flags & TF_ORIG_IGN) ∧
          ¬(flags & SS_FORCE) ∧
          (¬(flags & SS_USER) ∨ ¬Flag(FTALKING))) return 0;
    setexecsig(p, flags & SS_RESTORE_MASK);
    if (¬(flags & SS_USER)) p→shtrap ← Λ;
    if (flags & SS_SHTRAP) {      /* let trapsig do the calling of f */
      p→shtrap ← f;
      f ← trapsig;
    }
    if (p→cursig ≠ f) {
      p→cursig ← f;
      sigemptyset(&sigact.sa_mask);
      sigact.sa_flags ← 0;      /* interruptible */
      sigact.sa_handler ← f;
      sigaction (p→signal, &sigact, Λ);
    }
    return 1;
  }
```

643. When about to call *exec* the signal mask needs to be reset because it will be inherited. When a signal handler is installed flags passed on to *setexecsig* indicate how the state should be restored in that moment.

```
#define SS_RESTORE_MASK 0x3    /* how to restore a signal before an exec */
#define SS_RESTORE_CURR 0      /* leave current handler in place */
#define SS_RESTORE_ORIG 1      /* restore original handler */
#define SS_RESTORE_DFL 2       /* restore to SIG_DFL */
#define SS_RESTORE_IGN 3       /* restore to SIG_IGN */
#define SS_FORCE BIT(3)        /* set signal even if original signal ignored */
#define SS_USER BIT(4)         /* user is doing the set (ie. trap command) */
#define SS_SHTRAP BIT(5)       /* trap for internal use (CHLD/ALRM/WINCH) */
```

⟨ trap.c 637 ⟩ +≡

```
void setexecsig(Trap *p, int restore)
{
  if (¬(p→flags & (TF_ORIG_IGN | TF_ORIG_DFL)))    /* XXX debugging */
    internal_errorf("%s: unset signal %d(%s)", __func__, p→signal, p→name);
  p→flags &= ∼(TF_EXEC_IGN | TF_EXEC_DFL);
  switch (restore & SS_RESTORE_MASK) {
  case SS_RESTORE_CURR: break;    /* leave things as they currently are */
  case SS_RESTORE_ORIG: p→flags |= p→flags & TF_ORIG_IGN ? TF_EXEC_IGN : TF_EXEC_DFL; break;
  case SS_RESTORE_DFL: p→flags |= TF_EXEC_DFL; break;
  case SS_RESTORE_IGN: p→flags |= TF_EXEC_IGN; break;
  }
}
```

644. ⟨ Save flags the first time a signal action is set 644 ⟩ ≡

```
if (¬(p→flags & (TF_ORIG_IGN | TF_ORIG_DFL))) {
  sigaction(p→signal, &Sigact_ign, &sigact);
  p→flags |= sigact.sa_handler ≡ SIG_IGN ? TF_ORIG_IGN : TF_ORIG_DFL;
  p→cursig ← SIG_IGN;
}
```

This code is used in section 642.

645. When a signal is received a bit is set in its *sigtraps* slot but otherwise no action is normally taken. When ksh is able to deal with signals later on it calls *runtraps* to cycle through any trap which has been set in *sigtraps*.

```
⟨ trap.c 637 ⟩ +≡
  void runtraps(int flag)
  {
    int i;
    Trap *p;
    if (ksh_tmout_state ≡ TMOUT_LEAVING) {
      ksh_tmout_state ← TMOUT_EXECUTING;
      warningf(false, "timed␣out␣waiting␣for␣input");
      unwind (LEXIT);
    }
    else ksh_tmout_state ← TMOUT_EXECUTING;      /* XXX: this means the alarm will have no effect if a
        trap is caught after the alarm was started...not good. */
    if (¬flag) trap ← 0;
    if (flag & TF_DFL_INTR) intrsig ← 0;
    if (flag & TF_FATAL) fatal_trap ← 0;
    for (p ← sigtraps, i ← NSIG + 1; −−i ≥ 0; p++)
      if (p→set ∧ (¬flag ∨ ((p→flags & flag) ∧ p→trap ≡ Λ))) runtrap(p);
  }
```

646. The ERR and EXIT pseudo-signals have their trap command cleared (after saving it locally) to avoid infinite recursion if they're raised during their operation. For all signals the trap command is fully parsed (by *command*) before anything is executed so there's no problem freeing *p→trap* (as *settrap* does) while it's in use. This would be done, for example, if a trap handler replaces itself.

```
⟨ trap.c 637 ⟩ +≡
  void runtrap(Trap *p)
  {
    int i ← p→signal;
    char *trapstr ← p→trap;
    int oexstat;
    int old_changed ← 0;
    p→set ← 0;      /* clear the "pending" bit */
    if (trapstr ≡ Λ) {⟨ Perform the default action and unwind or return 647 ⟩}      /* SIG_DFL */
    if (trapstr[0] ≡ '\0') return;      /* SIG_IGN—ignore it */
    if (i ≡ SIGEXIT_ ∨ i ≡ SIGERR_) {
      old_changed ← p→flags & TF_CHANGED;
      p→flags &= ∼TF_CHANGED;
      p→trap ← Λ;
    }
    oexstat ← exstat;
    command(trapstr, current_lineno);
    exstat ← oexstat;      /* discard trap handler's result */
    if (i ≡ SIGEXIT_ ∨ i ≡ SIGERR_) {      /* don't clear TF_CHANGED—it was not set by us */
      if (p→flags & TF_CHANGED) afree(trapstr, APERM);
      else p→trap ← trapstr;
      p→flags |= old_changed;
    }
  }
```

647. ⟨ Perform the default action and **unwind** or **return** 647 ⟩ ≡

```
if (p→flags & TF_FATAL) {      /* eg. SIGHUP */
  exstat ← 128 + i;
  unwind (LLEAVE);
}
if (p→flags & TF_DFL_INTR) {      /* eg. SIGINT/SIGQUIT/SIGTERM/etc. */
  exstat ← 128 + i;
  unwind (LINTR);
}
return;
```

This code is used in section 646.

648. A user trap is applied to a signal through *settrap* which fills in the **Trap** object and calls *setsig*. Traps with TF_DFL_INTR or TF_FATAL set are always trapped so that the shell can clean up—their default action is taken care of by *runtrap* so SIG_DFL is overridden here.

TF_SHELL_USES indicates signals which the user and/or the shell may handle and these are configured so that after the shell has performed its necessary functions the user handler can continue. The actual handler (usually *trapsig* but possibly *j_sigchld* or SIG_IGN) is not changed so that ksh can (not) intervene.

⟨ trap.c 637 ⟩ +≡

```
void settrap(Trap *p, char *s)
{
  sig_t f;
  afree(p→trap, APERM);
  p→trap ← str_save(s, APERM);      /* handles s ≡ Λ */
  p→flags |= TF_CHANGED;
  f ← ¬s ? SIG_DFL : s[0] ? trapsig : SIG_IGN;      /* Λ: default; "": ignore; *: trapsig */
  p→flags |= TF_USER_SET;
  if ((p→flags & (TF_DFL_INTR | TF_FATAL)) ∧ f ≡ SIG_DFL) f ← trapsig;
  else if (p→flags & TF_SHELL_USES) {
    if (¬(p→flags & TF_ORIG_IGN) ∨ Flag(FTALKING)) {
      p→flags &= ∼(TF_EXEC_IGN | TF_EXEC_DFL);      /* do what user wants at exec time */
      if (f ≡ SIG_IGN) p→flags |= TF_EXEC_IGN;
      else p→flags |= TF_EXEC_DFL;
    }
    return;      /* don't change handler to f */
  }      /* todo: should we let user know signal is ignored? how? */
  setsig(p, f, SS_RESTORE_CURR | SS_USER);
}
```

649. Signal/Trap Utilities. Called when we want to allow the user to ^C out of something—won't work if user has trapped SIGINT.

```
⟨ trap.c 637 ⟩ +≡
  void intrcheck(void)
  {
    if (intrsig) runtraps(TF_DFL_INTR | TF_FATAL);
  }
```

650. Called after EINTR to check if a signal which normally causes process termination has been received.

```
⟨ trap.c 637 ⟩ +≡
  int fatal_trap_check(void)
  {
    int i;
    Trap *p;      /* todo: should check if signal is fatal, not the TF_DFL_INTR flag */
    for (p ← sigtraps, i ← NSIG + 1; --i ≥ 0; p++)
      if (p→set ∧ (p→flags & (TF_DFL_INTR | TF_FATAL)))
        return 128 + p→signal;      /* return value is used as an exit code */
    return 0;
  }
```

651. Returns the signal number of any pending traps: ie. a signal which has occurred for which a trap has been set or for which the TF_DFL_INTR flag is set.

```
⟨ trap.c 637 ⟩ +≡
  int trap_pending(void)
  {
    int i;
    Trap *p;
    for (p ← sigtraps, i ← NSIG + 1; --i ≥ 0; p++)
      if (p→set ∧ ((p→trap ∧ p→trap[0]) ∨ ((p→flags & (TF_DFL_INTR | TF_FATAL)) ∧ ¬p→trap)))
        return p→signal;
    return 0;
  }
```

652. Clear pending traps and reset user's trap handlers—used after *fork*.

```
⟨ trap.c 637 ⟩ +≡
  void cleartraps(void)
  {
    int i;
    Trap *p;
    trap ← 0;
    intrsig ← 0;
    fatal_trap ← 0;
    for (i ← NSIG + 1, p ← sigtraps; --i ≥ 0; p++) {
      p→set ← 0;
      if ((p→flags & TF_USER_SET) ∧ (p→trap ∧ p→trap[0])) settrap(p, Λ);
    }
  }
```

653. Restore signals just before an *exec*.

```
⟨ trap.c 637 ⟩ +≡
  void restoresigs(void)
  {
    int i;
    Trap *p;
    for (i ← NSIG + 1, p ← sigtraps; --i ≥ 0; p++)
      if (p→flags & (TF_EXEC_IGN | TF_EXEC_DFL))
        setsig(p, (p→flags & TF_EXEC_IGN) ? SIG_IGN : SIG_DFL, SS_RESTORE_CURR | SS_FORCE);
  }
```

654. Called by *c_print* when writing to a co-process to ensure SIGPIPE won't kill the shell (unless the user catches it and exits).

```
⟨ trap.c 637 ⟩ +≡
  int block_pipe(void)
  {
    int restore_dfl ← 0;
    Trap *p ← &sigtraps[SIGPIPE];
    if (¬(p→flags & (TF_ORIG_IGN | TF_ORIG_DFL))) {
      setsig(p, SIG_IGN, SS_RESTORE_CURR);
      if (p→flags & TF_ORIG_DFL) restore_dfl ← 1;
    }
    else if (p→cursig ≡ SIG_DFL) {
      setsig(p, SIG_IGN, SS_RESTORE_CURR);
      restore_dfl ← 1;      /* restore to SIG_DFL */
    }
    return restore_dfl;
  }
```

655. Called by *c_print* to undo whatever *block_pipe* did.

```
⟨ trap.c 637 ⟩ +≡
  void restore_pipe(int restore_dfl)
  {
    if (restore_dfl) setsig(&sigtraps[SIGPIPE], SIG_DFL, SS_RESTORE_CURR);
  }
```

656. Alarm. By way of example the SIGALRM handler is included here in full.

```
⟨ Type definitions 17 ⟩ +≡
  enum tmout_enum {
    TMOUT_EXECUTING ← 0,      /* executing commands */
    TMOUT_READING,      /* waiting for input */
    TMOUT_LEAVING      /* have timed out */
  };
```

657.
```
⟨ Global variables 5 ⟩ +≡
  unsigned int ksh_tmout;
  enum tmout_enum ksh_tmout_state ← TMOUT_EXECUTING;
```

658.
```
⟨ Externally-linked variables 6 ⟩ +≡
  extern unsigned int ksh_tmout;
  extern enum tmout_enum ksh_tmout_state;
```

659. Initialisation sets the TF_SHELL_USES flag to say that whatever the user wants, run this handler first.

```
⟨ trap.c 637 ⟩ +≡
  static void alarm_catcher(int sig);
  void alarm_init(void)
  {
    sigtraps[SIGALRM].flags |= TF_SHELL_USES;
    setsig(&sigtraps[SIGALRM], alarm_catcher, SS_RESTORE_ORIG | SS_FORCE | SS_SHTRAP);
  }
```

660. The handler (after saving *errno*) checks if the $TMOUT variable has been set and if so, whether that much time has passed. If not enough time has passed the alarm is re-applied. The user's ALRM handler will be invoked after this if there is one.

```
⟨ trap.c 637 ⟩ +≡
  static void alarm_catcher(int sig)
  {
    int errno_ ← errno;
    if (ksh_tmout_state ≡ TMOUT_READING) {
      int left ← alarm(0);
      if (left ≡ 0) {
        ksh_tmout_state ← TMOUT_LEAVING;
        intrsig ← 1;
      }
      else alarm(left);
    }
    errno ← errno_;
  }
```

661. Terminal (tty) Devices. Historically the code to manage a terminal device relied heavily on C preprocessor macros to enable ksh to work on the wide variety of hardware which was available in the 1980s. Now much of these differences are contained within portable libraries and there is little left that's specific to managing a tty device but the heavy-handed code separation technique remains.

```
⟨ tty.h 661 ⟩ ≡
#include <termios.h>
  extern void tty_init(int);
  extern void tty_close(void);
```

662.
```
⟨ Externally-linked variables 6 ⟩ +≡
  extern int tty_fd;
  extern int tty_devtty;
  extern struct termios tty_state;
```

663.
```
⟨ tty.c 663 ⟩ ≡
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
#include "tty.h"
  int tty_fd ← −1;      /* dup'd tty file descriptor */
  int tty_devtty;      /* true if tty_fd is from /dev/tty */
  struct termios tty_state;      /* saved tty state */
```

See also sections 664 and 665.

664. Only two functions are needed now. Closing the terminal comes first because it's the first thing the only other function—opening it—tries to do.

```
⟨ tty.c 663 ⟩ +≡
  void tty_close(void)
  {
    if (tty_fd ≥ 0) {
      close(tty_fd);
      tty_fd ← −1;
    }
  }
```

665. If `/dev/tty` can't be opened then ksh falls back on file descriptor 0 or 2 provided one of them is a `tty`. The file descriptor representing the controlling terminal is copied or moved to the address space above `FDBASE` and the terminal settings are copied into *tty_state*.

```
⟨ tty.c 663 ⟩ +≡
  void tty_init(int init_ttystate)
  {
    int do_close ← 1;      /* TODO: serves no useful purpose */
    int tfd;
    tty_close( );
    tty_devtty ← 1;
    tfd ← open("/dev/tty", O_RDWR, 0);
    if (tfd ≡ −1) {
      tty_devtty ← 0;
      warningf(false, "No␣controlling␣tty␣(open␣/dev/tty:␣%s)", strerror(errno));
      do_close ← 0;
      if (isatty(0)) tfd ← 0;
      else if (isatty(2)) tfd ← 2;
      else {
        warningf(false, "Can't␣find␣tty␣file␣descriptor");
        return;
      }
    }
    if ((tty_fd ← fcntl(tfd, F_DUPFD_CLOEXEC, FDBASE)) ≡ −1) {
      warningf(false, "%s:␣dup␣of␣tty␣fd␣failed:␣%s", __func__, strerror(errno));
    }
    else if (init_ttystate) tcgetattr(tty_fd, &tty_state);
    if (do_close) close(tfd);
  }
```

666. Job Control. This is the original (re)work of Eric Gisin, Ron Natalie, Larry Bouzane, Michael Rendell and numerous OpenBSD developers and contributors covering over 60 revisions.

A global list of all jobs is maintained in *job_list*. Each item in this list has a list of subprocesses which make up the job and share a process group id.

⟨ jobs.c 666 ⟩ ≡
```
#include <sys/resource.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/wait.h>

#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "sh.h"
#include "tty.h"

⟨ jobs.c static functions 668 ⟩ & ⟨ jobs.c static variables 669 ⟩
```
See also sections 672, 677, 683, 688, 689, 695, 696, 701, 703, 707, 708, 709, 715, 716, 717, 718, 719, 720, 721, 722, 725, 726, 728, 732, 737, 738, 739, and 740.

667. ⟨ Shared function declarations 4 ⟩ +≡
```
void j_init(int);
void j_suspend(void);
void j_exit(void);
void j_change(void);
int exchild(struct Op *, int, volatile int *, int);
void startlast(void);
int waitlast(void);
int waitfor(const char *, int *);
int j_kill(const char *, int);
int j_resume(const char *, int);
int j_jobs(const char *, int, int);
int j_njobs(void);
void j_notify(void);
pid_t j_async(void);
int j_stopped_running(void);
```

668. This gets its own section because the `jobs.c` premable is too long to fit comfortably on a single page.

```
⟨ jobs.c static functions 668 ⟩ ≡
  static void j_set_async(Job *);
  static void j_startjob(Job *);
  static int j_waitj(Job *, int, const char *);
  static void j_sigchld(int);
  static void j_print(Job *, int, struct Shf *);
  static Job *j_lookup(const char *, int *);
  static Job *new_job(void);
  static Proc *new_proc(void);
  static void check_job(Job *);
  static void put_job(Job *, int);
  static void remove_job(Job *, const char *);
  static int kill_job(Job *, int);
```

This code is used in section 666.

669.

```
⟨ jobs.c static variables 669 ⟩ ≡
  struct timeval j_systime, j_usrtime;      /* user and system time of last j_waitjed job */
  static Job *job_list;      /* job list */
  static Job *last_job;
  static Job *async_job;
  static pid_t async_pid;
  static Job *free_jobs;
  static Proc *free_procs;
  static int nzombie;      /* No. of zombies owned by this process */
  int njobs;      /* No. of jobs started */
  static int child_max;      /* CHILD_MAX */
  static volatile sig_atomic_t held_sigchld;      /* if SIGCHLD occurs before a job is completely started */
  static struct Shf *shl_j;
  static int ttypgrp_ok;      /* set if can use tty pgrps */
  static pid_t restore_ttypgrp ← -1;
  static pid_t our_pgrp;
  static const char *const lookup_msgs[] ← {
      null,      /* ie. a 1-byte array of '\0' */
      "no such job",
      "ambiguous",
      "argument must be %job or process id",
      Λ
  };
```

See also section 676.

This code is used in section 666.

670. The order of these *state* codes is important—the highest-valued state of any process in a job is taken as the state of the job as a whole. The *status* field records the system return value of a process returned from *waitpid*.

```
#define PRUNNING 0
#define PEXITED 1
#define PSIGNALLED 2
#define PSTOPPED 3
```

⟨ Type definitions 17 ⟩ +≡

```
typedef struct proc Proc;
struct proc {
  Proc *next;     /* next process in pipeline (if any) */
  int state;      /* from above */
  int status;     /* wait status */
  pid_t pid;      /* process id */
  char command[48];     /* process command string */
};
```

671.

```
#define JF_STARTED 0x001     /* set when all processes in job are started */
#define JF_WAITING 0x002     /* set if j_waitj is waiting on job */
#define JF_W_ASYNCNOTIFY 0x004     /* set if waiting and async notification ok */
#define JF_XXCOM 0x008     /* set for ` ... ` jobs */
#define JF_FG 0x010     /* running in foreground (also has tty pgrp) */
#define JF_SAVEDTTY 0x020     /* j->ttystate is valid */
#define JF_CHANGED 0x040     /* process has changed state */
#define JF_KNOWN 0x080     /* $! referenced */
#define JF_ZOMBIE 0x100     /* known, unwaited process */
#define JF_REMOVE 0x200     /* flagged for removal (j_jobs/j_notify) */
#define JF_USETTYMODE 0x400     /* tty mode saved if process exits normally */
#define JF_SAVEDTTYPGRP 0x800     /* j->saved_ttypgrp is valid */
#define JF_PIPEFAIL 0x1000     /* pipefail on when job was started */
```

⟨ Type definitions 17 ⟩ +≡

```
typedef struct Job Job;
```

672. This can't be a global type definition without dragging **termios**/**timeval** into everything.

```
⟨ jobs.c 666 ⟩ +≡
  struct Job {      /* renamed for cosmetic reasons */
    Job *next;      /* next job in list */
    int job;      /* job number: %n */
    int flags;      /* see JF_* */
    int state;      /* job state */
    int status;      /* exit status of last process */
    pid_t pgrp;      /* process group of job */
    pid_t ppid;      /* pid of process that forked job */
    int age;      /* number of jobs started */
    struct timeval systime;      /* system time used by job */
    struct timeval usrtime;      /* user time used by job */
    Proc *proc_list;      /* process list */
    Proc *last_proc;      /* last process in list */
    Coproc_id coproc_id;      /* 0 or id of coprocess output pipe */
    struct termios ttystate;      /* saved tty state for stopped jobs */
    pid_t saved_ttypgrp;      /* saved tty process group for stopped jobs */
  };
```

673. Job control is initialised whether started with `-i` or `+i`. Unlike other flags `FMONITOR` will be 127 if it was not specified on the command line or 0 if it was *disabled* (`+m`) on the command line.

```
⟨ Initialise job control 673 ⟩ ≡
  i ← Flag(FMONITOR) ≠ 127;
  Flag(FMONITOR) ← 0;
  j_init(i);
```

This code is used in section 15.

674. When jobs are running the shell will print "You have stopped jobs" the first time exit is attempted. This state is tracked in *really_exit*.

```
⟨ Global variables 5 ⟩ +≡
  int really_exit;
```

675.

```
⟨ Externally-linked variables 6 ⟩ +≡
  extern int really_exit;
```

676. Signals from the controlling terminal (*tt_sigs*) are trapped by the shell only if it performs job control and so initialisation is completed by *j_change* (which calls *tty_init*).

```
⟨ jobs.c static variables 669 ⟩ +≡
  static int const tt_sigs[] ← {SIGTSTP, SIGTTIN, SIGTTOU};
```

677.

```
⟨ jobs.c 666 ⟩ +≡
  void j_init(int mflagset)
  {
    child_max ← CHILD_MAX;      /* so syscon isn't always being called */
    ⟨ Initialise signal masks & handlers 681 ⟩
    if (¬mflagset ∧ Flag(FTALKING)) Flag(FMONITOR) ← 1;
    ⟨ Prepare for interaction 682 ⟩
    if (Flag(FMONITOR)) j_change();
    else if (Flag(FTALKING)) tty_init(true);
  }
```

678. Ordinarily no signals are blocked (*sm_default*) except when SIGCHLD is received (it is blocked by the operating system until the signal handler returns) or when the *sm_sigchld* set is applied while the job and process lists are being examined or changed.

679. ⟨ Global variables 5 ⟩ +≡
```
sigset_t sm_default, sm_sigchld;
```

680. ⟨ Externally-linked variables 6 ⟩ +≡
```
extern sigset_t sm_default, sm_sigchld;
```

681. ⟨ Initialise signal masks & handlers 681 ⟩ ≡
```
sigemptyset(&sm_default);
sigprocmask(SIG_SETMASK, &sm_default, Λ);
sigemptyset(&sm_sigchld);
sigaddset(&sm_sigchld, SIGCHLD);
setsig(&sigtraps[SIGCHLD], j_sigchld, SS_RESTORE_ORIG | SS_FORCE | SS_SHTRAP);
```
This code is used in section 677.

682. A private **Shf** object is required for asynchronous notification because it's used in an interrupt handler. If the shell is interactive then the signals which respond to a controlling terminal (*tt_sigs*—SIGTSTP, SIGTTIN & SIGTTOU) are ignored.

⟨ Prepare for interaction 682 ⟩ ≡
```
shl_j ← shf_fdopen(2, SHF_WR, Λ);
if (Flag(FMONITOR) ∨ Flag(FTALKING)) {
  int i;
  for (i ← NELEM(tt_sigs); --i ≥ 0; ) {
    sigtraps[tt_sigs[i]].flags |= TF_SHELL_USES;
    setsig(&sigtraps[tt_sigs[i]], SIG_IGN, SS_RESTORE_IGN | SS_FORCE);
      /* j_change sets this to SS_RESTORE_DFL if FMONITOR */
  }
}
```
This code is used in section 677.

683. If FMONITOR is not set then this will turn job control off (this will never happen when called from *j_init*), otherwise it is turned on.

⟨ jobs.c 666 ⟩ +≡
```
void j_change(void)
{
  int i;
  if (Flag(FMONITOR)) {⟨ Turn job control on 685 ⟩}
  else {⟨ Turn job control off 684 ⟩}
}
```

684. To turn job control back off the *tt_sigs* signals are ignored if interactive, or reset otherwise to the default value determined during initialisation.

```
⟨ Turn job control off 684 ⟩ ≡
  ttypgrp_ok ← 0;
  if (Flag(FTALKING))
    for (i ← NELEM(tt_sigs); --i ≥ 0; )
      setsig(&sigtraps[tt_sigs[i]], SIG_IGN, SS_RESTORE_IGN | SS_FORCE);
  else
    for (i ← NELEM(tt_sigs); --i ≥ 0; ) {
      if (sigtraps[tt_sigs[i]].flags & (TF_ORIG_IGN | TF_ORIG_DFL))
        setsig(&sigtraps[tt_sigs[i]],
            (sigtraps[tt_sigs[i]].flags & TF_ORIG_IGN) ? SIG_IGN : SIG_DFL,
            SS_RESTORE_ORIG | SS_FORCE);
    }
  if (¬Flag(FTALKING)) tty_close( );
```

This code is used in section 683.

685. If the shell is interactive then, assuming that `/dev/tty` can be opened,

```
⟨ Turn job control on 685 ⟩ ≡
  int use_tty;
  if (Flag(FTALKING)) {
    use_tty ← 1;      /* remember to, but don't call tcgetattr until we own the tty process group */
    tty_init(false);
  }
  else use_tty ← 0;
  ttypgrp_ok ← use_tty ∧ tty_fd ≥ 0 ∧ tty_devtty;      /* no controlling tty: no SIGT* */
  if (ttypgrp_ok ∧ (our_pgrp ← getpgrp( )) < 0) {
    warningf(false, "%s: getpgrp() failed: %s", __func__, strerror(errno));
    ttypgrp_ok ← 0;
  }
  if (ttypgrp_ok) {⟨ Wait to be given tty 686 ⟩}
  for (i ← NELEM(tt_sigs); --i ≥ 0; ) setsig(&sigtraps[tt_sigs[i]], SIG_IGN, SS_RESTORE_DFL | SS_FORCE);
  if (ttypgrp_ok ∧ our_pgrp ≠ kshpid) {⟨ Set process group ID 687 ⟩}
  if (use_tty) {
    if (¬ttypgrp_ok) warningf(false, "warning: won't have full job control");
  }
  if (tty_fd ≥ 0) tcgetattr(tty_fd, &tty_state);
```

This code is used in section 683.

686. (POSIX.1, B.2, job control)

```
⟨ Wait to be given tty 686 ⟩ ≡
  setsig(&sigtraps[SIGTTIN], SIG_DFL, SS_RESTORE_ORIG | SS_FORCE);
  while (1) {
    pid_t ttypgrp;
    if ((ttypgrp ← tcgetpgrp(tty_fd)) ≡ −1) {
      warningf(false, "%s: tcgetpgrp() failed: %s", __func__, strerror(errno));
      ttypgrp_ok ← 0;
      break;
    }
    if (ttypgrp ≡ our_pgrp) break;
    kill(0, SIGTTIN);
  }
```

This code is used in section 685.

687.

```
⟨ Set process group ID 687 ⟩ ≡
  if (setpgid(0, kshpid) ≡ −1) {
    warningf(false, "%s: setpgid() failed: %s", __func__, strerror(errno));
    ttypgrp_ok ← 0;
  }
  else {
    if (tcsetpgrp(tty_fd, kshpid) ≡ −1) {
      warningf(false, "%s: tcsetpgrp() failed: %s", __func__, strerror(errno));
      ttypgrp_ok ← 0;
    }
    else restore_ttypgrp ← our_pgrp;
    our_pgrp ← kshpid;
  }
```

This code is used in section 685.

688. Job cleanup before shell exit; kill stopped, and possibly running, jobs.
Test says: if job was started by this process and:

* is PSTOPPED,
* or is PRUNNING, either in the foreground or was started by *this* FLOGIN & FNOHUP shell.

```
⟨ jobs.c 666 ⟩ +≡
  void j_exit(void)
  {
    Job *j;
    int killed ← 0;
    for (j ← job_list; j ≠ Λ; j ← j→next) {
      if (j→ppid ≡ procpid ∧
            (j→state ≡ PSTOPPED ∨
                (j→state ≡ PRUNNING ∧
                    ((j→flags & JF_FG) ∨ (Flag(FLOGIN) ∧ ¬Flag(FNOHUP) ∧ procpid ≡ kshpid))))) {
        killed ← 1;
        if (j→pgrp ≡ 0) kill_job(j, SIGHUP);
        else killpg(j→pgrp, SIGHUP);
        if (j→state ≡ PSTOPPED) {
          if (j→pgrp ≡ 0) kill_job(j, SIGCONT);
          else killpg(j→pgrp, SIGCONT);
        }
      }
    }
    if (killed) sleep(1);
    j_notify( );
    if (kshpid ≡ procpid ∧ restore_ttypgrp ≥ 0) {      /* Need to restore the tty pgrp to what it was when
          the * shell started up, so that the process that started us * will be able to access the tty when
          we are done. * Also need to restore our process group in case we are * about to do an exec so
          that both our parent and the * process we are to become will be able to access the tty. */
      tcsetpgrp(tty_fd, restore_ttypgrp);
      setpgid(0, restore_ttypgrp);
    }
    if (Flag(FMONITOR)) {
      Flag(FMONITOR) ← 0;
      j_change( );
    }
  }
```

689. Subprocesses. Execute tree in child subprocess.

```
⟨ jobs.c 666 ⟩ +≡
  int exchild(struct Op *t, int flags, volatile int *xerrok, int close_fd)
  {
    static Proc *last_proc;      /* for pipelines */
    int i;
    sigset_t omask;
    Proc *p;
    Job *j;
    int rv ← 0;
    int forksleep;
    int ischild;
    if (flags & XEXEC)      /* Clear XFORK|XPCLOSE|XCCLOSE|XCOPROC|XPIPEO|XPIPEI|XXCOM|XBGND */
      return execute(t, flags & (XEXEC | XERROK), xerrok);      /* ... also done in another execute below */
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);      /* no SIGCHLDs while using job & process lists */
    ⟨ Link a new process into jobs list 690 ⟩
    snptreef(p→command, sizeof (p→command), "%T", t);
    ⟨ Create subprocess 691 ⟩
    if (Flag(FMONITOR) ∧ ¬(flags & XXCOM)) {⟨ Set up job control 692 ⟩}
    if (close_fd ≥ 0 ∧
          (((flags & XPCLOSE) ∧ ¬ischild) ∨
          ((flags & XCCLOSE) ∧ ischild)))
        close(close_fd);      /* each pipe's input fd */
    if (ischild) {⟨ Clean up and exec in subprocess 693 ⟩ }
    change_random( );      /* Ensure next child gets a (slightly) different $RANDOM sequence */
    if (¬(flags & XPIPEO)) {⟨ Mark the job as started 694 ⟩}      /* last process in a job */
    sigprocmask(SIG_SETMASK, &omask, Λ);
    return rv;
  }
```

690. ⟨ Link a new process into jobs list 690 ⟩ ≡

```
p ← new_proc( );
p→next ← Λ;
p→state ← PRUNNING;
p→status ← 0;
p→pid ← 0;
if (flags & XPIPEI) {      /* continuing with a pipe */
  if (¬last_job) internal_errorf("%s: XPIPEI and no last_job - pid %d", __func__, (int) procpid);
  j ← last_job;
  last_proc→next ← p;
  last_proc ← p;
}
else {
  j ← new_job( );      /* fills in j→job */      /* we don't consider XXCOM's foreground since they don't
      get tty process group and we don't save or restore tty modes. */
  j→flags ← (flags & XXCOM) ? JF_XXCOM : ((flags & XBGND) ? 0 : (JF_FG | JF_USETTYMODE));
  if (Flag(FPIPEFAIL)) j→flags |= JF_PIPEFAIL;
  timerclear(&j→usrtime);
  timerclear(&j→systime);
  j→state ← PRUNNING;
  j→pgrp ← 0;
  j→ppid ← procpid;
  j→age ← ++njobs;
  j→proc_list ← p;
  j→coproc_id ← 0;
  last_job ← j;
  last_proc ← p;
  put_job(j, PJ_PAST_STOPPED);
}
```

This code is used in section 689.

691. ⟨ Create subprocess 691 ⟩ ≡

```
forksleep ← 1;
while ((i ← fork( )) ≡ −1 ∧ errno ≡ EAGAIN ∧ forksleep < 32) {
  if (intrsig) break;      /* allow user to ^C out... */
  sleep(forksleep);
  forksleep ≪= 1;
}
if (i ≡ −1) {
  kill_job(j, SIGKILL);
  remove_job(j, "fork failed");
  sigprocmask(SIG_SETMASK, &omask, Λ);
  errorf("cannot fork - try again");
}
ischild ← i ≡ 0;
if (ischild) p→pid ← procpid ← getpid( );
else p→pid ← i;
```

This code is used in section 689.

692. ⟨ Set up job control 692 ⟩ ≡

```
int dotty ← 0;
if (j→pgrp ≡ 0) {      /* First process */
  j→pgrp ← p→pid;
  dotty ← 1;
}
setpgid(p→pid, j→pgrp);      /* set in both processes to deal with race condition */
#if 0      /* YYY: should this test be used instead? (see also YYY below) */
if (ttypgrp_ok ∧ ischild ∧ ¬(flags & XBGND)) tcsetpgrp(tty_fd, j→pgrp);
#endif
if (ttypgrp_ok ∧ dotty ∧ ¬(flags & XBGND)) tcsetpgrp(tty_fd, j→pgrp);
```

This code is used in section 689.

693. If FMONITOR or FTALKING is set the control terminal signals are ignored. If neither FMONITOR nor FTALKING are set the signals have their inherited values.

⟨ Clean up and *exec* in subprocess 693 ⟩ ≡

```
if (flags & XCOPROC) coproc_cleanup(false);      /* Do this before restoring signal */
sigprocmask(SIG_SETMASK, &omask, Λ);
cleanup_parents_env( );
if (Flag(FMONITOR) ∧ ¬(flags & XXCOM)) {
  for (i ← NELEM(tt_sigs); −−i ≥ 0; )
    setsig(&sigtraps[tt_sigs[i]], SIG_DFL, SS_RESTORE_DFL | SS_FORCE);
}
if (Flag(FBGNICE) ∧ (flags & XBGND)) nice(4);
if ((flags & XBGND) ∧ ¬Flag(FMONITOR)) {
  setsig(&sigtraps[SIGINT], SIG_IGN, SS_RESTORE_IGN | SS_FORCE);
  setsig(&sigtraps[SIGQUIT], SIG_IGN, SS_RESTORE_IGN | SS_FORCE);
  if (¬(flags & (XPIPEI | XCOPROC))) {
    int fd ← open("/dev/null", O_RDONLY);
    if (fd ≠ 0) {
      (void) ksh_dup2(fd, 0, true);
      close(fd);
    }
  }
}
remove_job(j, "child");      /* in case of jobs command */
nzombie ← 0;
ttypgrp_ok ← 0;
Flag(FMONITOR) ← 0;
Flag(FTALKING) ← 0;
tty_close( );
cleartraps( );
execute(t, (flags & XERROK) | XEXEC, Λ);      /* no return */
internal_warningf("%s: execute() returned", __func__);
unwind (LLEAVE);      /* NOTREACHED */
```

This code is used in section 689.

694. All of the processes which make up a pipeline have now been created on the system so the job can be marked as started.

```
⟨ Mark the job as started 694 ⟩ ≡
#if 0    /* YYY: Is this needed? (see also YYY above) */
  if (Flag(FMONITOR) ∧ ¬(flags & (XXCOM | XBGND))) tcsetpgrp(tty_fd, j→pgrp);
#endif
  j_startjob(j);
  if (flags & XCOPROC) {
    j→coproc_id ← coproc.id;
    coproc.njobs++;    /* njobs using co-process output */
    coproc.job ← (void *) j;    /* j using co-process input */
  }
  if (flags & XBGND) {
    j_set_async(j);
    if (Flag(FTALKING)) {
      shf_fprintf(shl_out, "[%d]", j→job);
      for (p ← j→proc_list; p; p ← p→next) shf_fprintf(shl_out, "␣%d", p→pid);
      shf_putchar('\n', shl_out);
      shf_flush(shl_out);
    }
  }
  else rv ← j_waitj(j, JW_NONE, "jw:last␣proc");
```

This code is used in section 689.

695. After a job is started SIGCHLD is re-sent to the current process if one was received while the job was still being created. Expects SIGCHLD to be blocked.

```
⟨ jobs.c 666 ⟩ +≡
  static void j_startjob(Job *j)
  {
    Proc *p;
    j→flags |= JF_STARTED;
    for (p ← j→proc_list; p→next; p ← p→next) ;
    j→last_proc ← p;
    if (held_sigchld) {
      held_sigchld ← 0;
      kill(procpid, SIGCHLD);    /* Don't call j_sigchld as it may remove the job. */
    }
  }
```

696. Move a job between foreground and background. Called only if Flag(FMONITOR) set.

```
⟨ jobs.c 666 ⟩ +≡
  int j_resume(const char *cp, int bg)
  {
    Job *j;
    Proc *p;
    int ecode;
    int running;
    int rv ← 0;
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    if ((j ← j_lookup(cp, &ecode)) ≡ Λ) {
      sigprocmask(SIG_SETMASK, &omask, Λ);
      bi_errorf("%s: %s", cp, lookup_msgs[ecode]);
      return 1;
    }
    if (j→pgrp ≡ 0) {
      sigprocmask(SIG_SETMASK, &omask, Λ);
      bi_errorf("job not job-controlled");
      return 1;
    }
    if (bg) shprintf("[%d] ", j→job);
    ⟨ Get a job's status and print its command 697 ⟩
    put_job(j, PJ_PAST_STOPPED);
    if (bg) j_set_async(j);
    else {⟨ Attach tty to job 698 ⟩}
    if (j→state ≡ PRUNNING ∧ killpg(j→pgrp, SIGCONT) ≡ −1) {⟨ Attempt to SIGCONT a job 699 ⟩}
    if (¬bg) {⟨ Wait for a foregrounded job to finish 700 ⟩}
    sigprocmask(SIG_SETMASK, &omask, Λ);
    return rv;
  }
```

697. ⟨ Get a job's status and print its command 697 ⟩ ≡

```
  running ← 0;
  for (p ← j→proc_list; p ≠ Λ; p ← p→next) {
    if (p→state ≡ PSTOPPED) {
      p→state ← PRUNNING;
      p→status ← 0;
      running ← 1;
    }
    shprintf("%s%s", p→command, p→next ? "| " : "");
  }
  shprintf("\n");
  shf_flush(shl_stdout);
  if (running) j→state ← PRUNNING;
```

This code is used in section 696.

698. ⟨ Attach tty to job 698 ⟩ ≡

```
if (j→state ≡ PRUNNING) {
  if (ttypgrp_ok ∧ (j→flags & JF_SAVEDTTY)) tcsetattr(tty_fd, TCSADRAIN, &j→ttystate);
  if (ttypgrp_ok ∧ tcsetpgrp(tty_fd, (j→flags & JF_SAVEDTTYPGRP) ? j→saved_ttypgrp : j→pgrp) ≡ −1) {
    /* See comment in j_waitj regarding saved_ttypgrp. */
    if (j→flags & JF_SAVEDTTY) tcsetattr(tty_fd, TCSADRAIN, &tty_state);
    sigprocmask(SIG_SETMASK, &omask, Λ);
    bi_errorf("1st␣tcsetpgrp(%d,␣%d)␣failed:␣%s", tty_fd,
        (int)((j→flags & JF_SAVEDTTYPGRP) ? j→saved_ttypgrp : j→pgrp), strerror(errno));
    return 1;
  }
}
j→flags |= JF_FG;
j→flags &= ∼JF_KNOWN;
if (j ≡ async_job) async_job ← Λ;
```

This code is used in section 696.

699. ⟨ Attempt to SIGCONT a job 699 ⟩ ≡

```
int err ← errno;
if (¬bg) {
  j→flags &= ∼JF_FG;
  if (ttypgrp_ok ∧ (j→flags & JF_SAVEDTTY)) tcsetattr(tty_fd, TCSADRAIN, &tty_state);
  if (ttypgrp_ok ∧ tcsetpgrp(tty_fd, our_pgrp) ≡ −1) {
    warningf(true, "fg:␣2nd␣tcsetpgrp(%d,␣%d)␣failed:␣%s", tty_fd, (int) our_pgrp, strerror(errno));
  }
}
sigprocmask(SIG_SETMASK, &omask, Λ);
bi_errorf("cannot␣continue␣job␣%s:␣%s", cp, strerror(err));
return 1;
```

This code is used in section 696.

700. ⟨ Wait for a foregrounded job to finish 700 ⟩ ≡

```
if (ttypgrp_ok) {
  j→flags &= ∼(JF_SAVEDTTY | JF_SAVEDTTYPGRP);
}
rv ← j_waitj(j, JW_NONE, "jw:resume");
```

This code is used in section 696.

701. SIGCHLD Handler.

SIGCHLD handler to reap children and update job states

Expects sigchld to be blocked.

Don't wait for any processes if a job is partially started. This is so we don't do away with the process group leader before all the processes in a pipe line are started (so the *setpgid* won't fail).

Any processes which have state to report will return from *waitpid*; calculate the job usrtime/systime for it then check to see if it means an entire job is done.

```
⟨ jobs.c 666 ⟩ +≡
  static void j_sigchld(int sig)
  {
    int errno_ ← errno;
    Job *j;
    Proc *p ← Λ;
    int pid;
    int status;
    struct rusage ru0, ru1;
    for (j ← job_list; j; j ← j→next)
      if (j→ppid ≡ procpid ∧ ¬(j→flags & JF_STARTED)) {
        held_sigchld ← 1;
        goto finished;
      }
    getrusage(RUSAGE_CHILDREN, &ru0);
    do {
      pid ← waitpid(−1, &status, (WNOHANG | WUNTRACED));
      if (pid ≤ 0) break;      /* return if would block (0) or no children or interrupted (-1) */
      getrusage(RUSAGE_CHILDREN, &ru1);
      ⟨ Find job j and process p structures for this pid 702 ⟩
    found:
      if (j ≡ Λ) {      /* Can occur if process has kids, then execs shell */
#if 0
        warningf(true, "bad process waited for (pid = %d)", pid);
#endif
        ru0 ← ru1;
        continue;
      }
      timeradd(&j→usrtime, &ru1.ru_utime, &j→usrtime);
      timersub(&j→usrtime, &ru0.ru_utime, &j→usrtime);
      timeradd(&j→systime, &ru1.ru_stime, &j→systime);
      timersub(&j→systime, &ru0.ru_stime, &j→systime);
      ru0 ← ru1;
      p→status ← status;
      if (WIFSTOPPED(status)) p→state ← PSTOPPED;
      else if (WIFSIGNALED(status)) p→state ← PSIGNALLED;
      else p→state ← PEXITED;
      check_job(j);      /* check to see if entire job is done */
    } while (1);
  finished:
    errno ← errno_;
  }
```

702. ⟨ Find job j and process p structures for this *pid* 702 ⟩ ≡

```
for (j ← job_list; j ≠ Λ; j ← j→next)
  for (p ← j→proc_list; p ≠ Λ; p ← p→next)
    if (p→pid ≡ pid) goto found;
```

This code is used in section 701.

703. If no processes are running the job status and state are updated, asynchronous job notification is done and, if unneeded, the job is removed.

Expects sigchld to be blocked.

```
⟨ jobs.c 666 ⟩ +≡
static void check_job(Job *j)
{
  int jstate;
  Proc *p;
  if (¬(j→flags & JF_STARTED)) {      /* XXX debugging (nasty—interrupt routine using shl_out) */
    internal_warningf("%s: job started (flags 0x%x)", __func__, j→flags);
    return;
  }
  ⟨ Obtain process status and job state 704 ⟩
  ⟨ Note when co-process dies 705 ⟩
  j→flags |= JF_CHANGED;
  if (Flag(FMONITOR) ∧ ¬(j→flags & JF_XXCOM)) {      /* Only put stopped jobs at the front to avoid
        confusing the user (don't want finished jobs affecting %+ or %-) */
    if (j→state ≡ PSTOPPED) put_job(j, PJ_ON_FRONT);
    if (Flag(FNOTIFY) ∧ (j→flags & (JF_WAITING | JF_W_ASYNCNOTIFY)) ≠ JF_WAITING) {
      ⟨ Look for the real stderr 706 ⟩
      j_print(j, JP_MEDIUM, shl_j);      /* Can't call j_notify as it removes jobs—the job must stay in the
          job list as j_waitj may be running with this job. */
      shf_flush(shl_j);
      if (¬(j→flags & JF_WAITING) ∧ j→state ≠ PSTOPPED) remove_job(j, "notify");
    }
  }
  if (¬Flag(FMONITOR) ∧ ¬(j→flags & (JF_WAITING | JF_FG)) ∧ j→state ≠ PSTOPPED) {
    if (j ≡ async_job ∨ (j→flags & JF_KNOWN)) {
      j→flags |= JF_ZOMBIE;
      j→job ← −1;
      nzombie++;
    }
    else remove_job(j, "checkjob");
  }
}
```

704. ⟨ Obtain process status and job state 704 ⟩ ≡

```
jstate ← PRUNNING;
for (p ← j→proc_list; p ≠ Λ; p ← p→next) {
  if (p→state ≡ PRUNNING) return;      /* some processes still running */
  if (p→state > jstate) jstate ← p→state;      /* This is why that order mattered */
}
j→state ← jstate;
switch (j→last_proc→state) {
case PEXITED: j→status ← WEXITSTATUS(j→last_proc→status); break;
case PSIGNALLED: j→status ← 128 + WTERMSIG(j→last_proc→status); break;
default: j→status ← 0; break;
}
```

This code is used in section 703.

705. Note when co-process dies: can't be done in *j_wait* nor *remove_job* since neither may be called for non-interactive shells.

TODO: would be nice to get the closes out of here so they aren't done in the signal handler. Would mean a check in *coproc_getfd* to do "`if job == 0 && write >= 0, close write`".

⟨ Note when co-process dies 705 ⟩ ≡

```
if (j→state ≡ PEXITED ∨ j→state ≡ PSIGNALLED) {
    /* No need to keep co-process input any more (at least, this is what ksh93d thinks) */
  if (coproc.job ≡ j) {
    coproc.job ← Λ;
    coproc_write_close(coproc.write);
  }
  if (j→coproc_id ∧ j→coproc_id ≡ coproc.id ∧ −− coproc.njobs ≡ 0)
      /* Do we need to keep the output? */
    coproc_readw_close(coproc.read);
}
```

This code is used in section 703.

706. This should probably use the symbol *stderr* but it's safe to say that the meaning of file descriptors 0, 1 & 2 is unlikely to change any time soon.

⟨ Look for the real *stderr* 706 ⟩ ≡

```
{
  struct Env *ep;
  int fd ← 2;
  for (ep ← genv; ep; ep ← ep→oenv)
    if (ep→savefd ∧ ep→savefd[2]) fd ← ep→savefd[2];
  shf_reopen(fd, SHF_WR, shl_j);
}
```

This code is used in section 703.

707. Signalling. There are two sides to process signalling, sending and receiving. Signals are sent using the wonderfully-named *kill* and *killpg* system calls used by *j_kill*. This also implements the ksh `kill` built-in. `SIGCHLD` is blocked while the job list is in use.

```
⟨ jobs.c 666 ⟩ +≡
  int j_kill(const char *cp, int sig)
  {
    Job *j;
    int rv ← 0;
    int ecode;
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    if ((j ← j_lookup(cp, &ecode)) ≡ Λ) {
      sigprocmask(SIG_SETMASK, &omask, Λ);
      bi_errorf("%s: %s", cp, lookup_msgs[ecode]);
      return 1;
    }
    if (j→pgrp ≡ 0) {      /* started when job control was disabled */
      if (kill_job(j, sig) < 0) {
        bi_errorf("%s: %s", cp, strerror(errno));
        rv ← 1;
      }
    }
    else {
      if (j→state ≡ PSTOPPED ∧ (sig ≡ SIGTERM ∨ sig ≡ SIGHUP)) (void) killpg(j→pgrp, SIGCONT);
      if (killpg(j→pgrp, sig) ≡ −1) {
        bi_errorf("%s: %s", cp, strerror(errno));
        rv ← 1;
      }
    }
    sigprocmask(SIG_SETMASK, &omask, Λ);
    return rv;
  }
```

708. A job which was started without or prior to job control being enabled are not contained within a process group so each process in a job is signalled individually. This is used by *j_kill* when `SIGCHLD` is already blocked.

```
⟨ jobs.c 666 ⟩ +≡
  static int kill_job(Job *j, int sig)
  {
    Proc *p;
    int rval ← 0;
    for (p ← j→proc_list; p ≠ Λ; p ← p→next)
      if (p→pid ≠ 0)
        if (kill(p→pid, sig) ≡ −1) rval ← −1;
    return rval;
  }
```

709. A subprocess doesn't (directly) send a signal back—the operating system reacts to a change in its running status by sending SIGCHLD on its behalf. When the shell needs to put itself in the background until it receives this signal *j_waitj* is called; this doesn't interfere with the SIGCHLD handler but instead expects it to have run and updated the job flags.

```
#define JW_NONE 0x00
#define JW_INTERRUPT 0x01 /* ^C will stop the wait */
#define JW_ASYNCNOTIFY 0x02 /* asynchronous notification during wait ok */
#define JW_STOPPEDWAIT 0x04 /* wait even if job stopped */
```

⟨ jobs.c 666 ⟩ +≡

```
static int j_waitj(Job *j, int flags, const char *where)
{
  int rv;
  j→flags |= JF_WAITING;
  if (flags & JW_ASYNCNOTIFY) j→flags |= JF_W_ASYNCNOTIFY; /* no auto-notify on this job */
  if (¬Flag(FMONITOR)) flags |= JW_STOPPEDWAIT; /* wait for stopped jobs also */
  ⟨ Wait while the job's running 710 ⟩ /* returns if a signal is trapped or fatal */
  j→flags &= ∼(JF_WAITING | JF_W_ASYNCNOTIFY); /* turn these back off */
  if (j→flags & JF_FG) { /* the job ended while in the foreground */
    int status;
    j→flags &= ∼JF_FG;
    if (Flag(FMONITOR) ∧ ttypgrp_ok ∧ j→pgrp) {⟨ Save the tty's current process group 711 ⟩}
    if (tty_fd ≥ 0) {⟨ Restore tty settings if they are unchanged 712 ⟩}
    ⟨ Simulate ^C to loops &c. 713 ⟩
  }
  j_usrtime ← j→usrtime;
  j_systime ← j→systime;
  if (j→flags & JF_PIPEFAIL) {⟨ Set rv from a job started under JF_PIPEFAIL 714 ⟩}
  else rv ← j→status;
  if (¬(flags & JW_ASYNCNOTIFY) ∧ /* ie. notification was not previously permitted */
      (¬Flag(FMONITOR) ∨ j→state ≠ PSTOPPED)) {
    j_print(j, JP_SHORT, shl_out);
    shf_flush(shl_out);
  }
  if (j→state ≠ PSTOPPED ∧ /* ie. PEXITED/PSIGNALLED */
      (¬Flag(FMONITOR) ∨ ¬(flags & JW_ASYNCNOTIFY)))
    remove_job(j, where);
  return rv;
}
```

710. ⟨ Wait while the job's running 710 ⟩ ≡

```
while ((volatile int) j→state ≡ PRUNNING ∨
        ((flags & JW_STOPPEDWAIT) ∧ (volatile int) j→state ≡ PSTOPPED)) {
  sigsuspend (&sm_default);      /* blocks until signal handlers have finished */
  if (fatal_trap) {
    int oldf ← j→flags & (JF_WAITING | JF_W_ASYNCNOTIFY);
    j→flags &= ∼(JF_WAITING | JF_W_ASYNCNOTIFY);
    runtraps(TF_FATAL);
    j→flags |= oldf;      /* not reached... */
  }
  if ((flags & JW_INTERRUPT) ∧ (rv ← trap_pending( ))) {
    j→flags &= ∼(JF_WAITING | JF_W_ASYNCNOTIFY);
    return −rv;
  }
}
```

This code is used in section 709.

711. The `tty`'s current process group is restored when the job is put in the foreground. This is to deal with things like the GNU su which does a *fork* & *exec* instead of an *exec* (the *fork* means the *exec*ed shell gets a different pid from its pgrp, so naturally it sets its pgrp and gets hosed when it gets foregrounded by the parent shell, which has restored the `tty`'s pgrp to that of the su process).

⟨ Save the `tty`'s current process group 711 ⟩ ≡

```
if (j→state ≡ PSTOPPED ∧ (j→saved_ttypgrp ← tcgetpgrp(tty_fd)) ≥ 0)  j→flags |= JF_SAVEDTTYPGRP;
if (tcsetpgrp(tty_fd, our_pgrp) ≡ −1) {
  warningf(true, "%s: tcsetpgrp(%d, %d) failed: %s", __func__, tty_fd, (int) our_pgrp,
      strerror(errno));
}
if (j→state ≡ PSTOPPED) {
  j→flags |= JF_SAVEDTTY;
  tcgetattr(tty_fd, &j→ttystate);
}
```

This code is used in section 709.

712. Only restore tty settings if job was originally started in the foreground. Problems can be caused by things like "`more foobar &`" which will typically get and save the shell's vi/emacs tty settings before setting up the tty for itself; when `more` exits, it restores the 'original' settings, and things go down hill from there...

If the job is stopped and later restarted and then exits the saved `tty` mode is not restored. Consider the sequence:

* `vi foo` (stopped)
* `...`
* `stty` ⟨`something`⟩
* `...`
* `fg` (vi; later quits normally)

The restored mode should be that set by the `stty` command not what it was before `vi` was started.

```
⟨ Restore tty settings if they are unchanged 712 ⟩ ≡
  if (j→state ≡ PEXITED ∧ j→status ≡ 0 ∧ (j→flags & JF_USETTYMODE)) {
    tcgetattr(tty_fd, &tty_state);
  }
  else {
    tcsetattr(tty_fd, TCSADRAIN, &tty_state);
    if (j→state ≡ PSTOPPED) j→flags &= ∼JF_USETTYMODE;
  }
```

This code is used in section 709.

713. If it looks like user hit `^C` to kill a job, pretend we got one too to break out of for loops, etc. (AT&T ksh does this even when not monitoring, but this doesn't make sense since a tty generated `^C` goes to the whole process group).

```
⟨ Simulate ^C to loops &c. 713 ⟩ ≡
  status ← j→last_proc→status;
  if (Flag(FMONITOR) ∧ j→state ≡ PSIGNALLED ∧
        WIFSIGNALED(status) ∧ (sigtraps[WTERMSIG(status)].flags & TF_TTY_INTR))
    trapsig(WTERMSIG(status));
```

This code is used in section 709.

714. If a job is a pipeline (and even if it's not) then if it was started while `FPIPEFAIL` (`set -o pipefail`) was in effect the exit status code is that of the last process in the list.

```
⟨ Set rv from a job started under JF_PIPEFAIL 714 ⟩ ≡
  Proc *p;
  int status;
  rv ← 0;
  for (p ← j→proc_list; p ≠ Λ; p ← p→next) {
    switch (p→state) {
    case PEXITED: status ← WEXITSTATUS(p→status); break;
    case PSIGNALLED: status ← 128 + WTERMSIG(p→status); break;
    default: status ← 0; break;
    }
    if (status) rv ← status;
  }
```

This code is used in section 709.

715. The most recently started job is saved in *last_job* and this—used after the last character of a command substitution[1] has been parsed—starts it. The following function waits for it to finish.

```
⟨ jobs.c 666 ⟩ +≡
  void startlast(void)
  {
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    if (last_job) {      /* no need to report error—waitlast will do it */
      last_job→flags |= JF_WAITING;      /* ensure it isn't removed by check_job */
      j_startjob(last_job);
    }
    sigprocmask(SIG_SETMASK, &omask, Λ);
  }
```

716.
```
⟨ jobs.c 666 ⟩ +≡
  int waitlast(void)
  {
    int rv;
    Job *j;
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    j ← last_job;
    if (¬j ∨ ¬(j→flags & JF_STARTED)) {
      if (¬j) warningf(true, "%s: no last job", __func__);
      else internal_warningf("%s: not started", __func__);
      sigprocmask(SIG_SETMASK, &omask, Λ);
      return 125;      /* a not-so-arbitrary non-zero value */
    }
    rv ← j_waitj(j, JW_NONE, "jw:waitlast");
    sigprocmask(SIG_SETMASK, &omask, Λ);
    return rv;
  }
```

[1] `$( ... )` / `` ` ... ` ``

717. Alternatively this waits for a particular process or the first process in the job list if $cp \equiv \Lambda$.

```
⟨ jobs.c 666 ⟩ +≡
  int waitfor(const char *cp, int *sigp)
  {
    int rv;
    Job *j;
    int ecode;
    int flags ← JW_INTERRUPT | JW_ASYNCNOTIFY;
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    *sigp ← 0;
    if (cp ≡ Λ) {      /* always returns 0 so no need to worry about exited/signaled jobs */
      for (j ← job_list; j; j ← j→next)      /* AT&T ksh will wait for stopped jobs—we don't */
        if (j→ppid ≡ procpid ∧ j→state ≡ PRUNNING) break;
      if (¬j) {
        sigprocmask(SIG_SETMASK, &omask, Λ);
        return −1;
      }
    }
    else if ((j ← j_lookup(cp, &ecode))) {
      flags &= ∼JW_ASYNCNOTIFY;      /* don't report normal job completion */
      if (j→ppid ≠ procpid) {
        sigprocmask(SIG_SETMASK, &omask, Λ);
        return −1;
      }
    }
    else {
      sigprocmask(SIG_SETMASK, &omask, Λ);
      if (ecode ≠ JL_NOSUCH) bi_errorf("%s: %s", cp, lookup_msgs[ecode]);
      return −1;
    }
    rv ← j_waitj(j, flags, "jw:waitfor");
    sigprocmask(SIG_SETMASK, &omask, Λ);
    if (rv < 0) *sigp ← 128 + −rv;      /* we were interrupted */
    return rv;
  }
```

718. Job List. Allocate a new job and fill in the job number. Takes a job from *free_jobs* if there is one; does not add the job to *job_list*. Expects SIGCHLD to be blocked.

```
⟨ jobs.c 666 ⟩ +≡
  static Job *new_job(void)
  {
    int i;
    Job *newj, *j;
    if (free_jobs ≠ Λ) {
      newj ← free_jobs;
      free_jobs ← free_jobs→next;
    }
    else newj ← alloc(sizeof(Job), APERM);      /* brute force method */
    for (i ← 1; ; i++) {
      for (j ← job_list; j ∧ j→job ≠ i; j ← j→next) ;
      if (j ≡ Λ) break;
    }
    newj→job ← i;
    return newj;
  }
```

719. Likewise, allocate a new process object while SIGCHLD is blocked.

```
⟨ jobs.c 666 ⟩ +≡
  static Proc *new_proc(void)
  {
    Proc *p;
    if (free_procs ≠ Λ) {
      p ← free_procs;
      free_procs ← free_procs→next;
    }
    else p ← alloc(sizeof(Proc), APERM);
    return p;
  }
```

720. Put j in a particular location (taking it out of *job_list* if it is there already). Expects SIGCHLD to be blocked.

```
#define PJ_ON_FRONT 0 /* at very front */
#define PJ_PAST_STOPPED 1 /* just past any stopped jobs */
```

⟨ jobs.c 666 ⟩ +≡

```
static void put_job(Job *j, int where)
{
  Job **prev, *curr; /* Remove job from list (if there) */
  prev ← &job_list;
  curr ← job_list;
  for ( ; curr ∧ curr ≠ j; prev ← &curr→next, curr ← *prev) ;
  if (curr ≡ j) *prev ← curr→next;
  switch (where) {
  case PJ_ON_FRONT: j→next ← job_list;
    job_list ← j;
    break;
  case PJ_PAST_STOPPED: prev ← &job_list;
    curr ← job_list;
    for ( ; curr ∧ curr→state ≡ PSTOPPED; prev ← &curr→next, curr ← *prev) ;
    j→next ← curr;
    *prev ← j;
    break;
  }
}
```

721. Take job out of *job_list* and put old structures into free list *free_jobs*. Keeps *nzombies*, *last_job* and *async_job* up to date. Expects `SIGCHLD` to be blocked.

```
⟨ jobs.c 666 ⟩ +≡
  static void remove_job(Job *j, const char *where)
  {
    Proc *p, *tmp;
    Job **prev, *curr;
    prev ← &job_list;
    curr ← *prev;
    for ( ; curr ≠ Λ ∧ curr ≠ j; prev ← &curr→next, curr ← *prev) ;
    if (curr ≠ j) {
      internal_warningf("%s: job not found (%s)", __func__, where);
      return;
    }
    *prev ← curr→next;
    for (p ← j→proc_list; p ≠ Λ; ) {      /* free up proc structures */
      tmp ← p;
      p ← p→next;
      tmp→next ← free_procs;
      free_procs ← tmp;
    }
    if ((j→flags & JF_ZOMBIE) ∧ j→ppid ≡ procpid) --nzombie;
    j→next ← free_jobs;
    free_jobs ← j;
    if (j ≡ last_job) last_job ← Λ;
    if (j ≡ async_job) async_job ← Λ;
  }
```

722. Convert a `%...` sequence to **Job**. Expects `SIGCHLD` to be blocked.

```
#define JL_OK 0
#define JL_NOSUCH 1 /* no such job */
#define JL_AMBIG 2 /* %foo or %?foo is ambiguous */
#define JL_INVALID 3 /* non-pid, non-% job id */
```

⟨ jobs.c 666 ⟩ +≡

```
static Job *j_lookup(const char *cp, int *ecodep)
{
  Job *j, *last_match;
  const char *errstr;
  Proc *p;
  int len, job ← 0;
  if (digit(*cp)) {
    job ← strtonum(cp, 1, INT_MAX, &errstr);
    if (errstr) {
      if (ecodep) *ecodep ← JL_NOSUCH;
      return Λ;
    } /* Look for last_proc→pid (what $! returns) first... */
    for (j ← job_list; j ≠ Λ; j ← j→next)
      if (j→last_proc ∧ j→last_proc→pid ≡ job) return j; /* ...then look for process group (this is
          non-POSIX) but should not break anything (so FPOSIX isn't used). */
    for (j ← job_list; j ≠ Λ; j ← j→next)
      if (j→pgrp ∧ j→pgrp ≡ job) return j;
    if (ecodep) *ecodep ← JL_NOSUCH;
    return Λ;
  }
  if (*cp ≠ '%') {
    if (ecodep) *ecodep ← JL_INVALID;
    return Λ;
  }
  switch (*++cp) {
  case '\0': /* non-standard */
  case '+': case '%':
    if (job_list ≠ Λ) return job_list;
    break;
  case '-':
    if (job_list ≠ Λ ∧ job_list→next) return job_list→next;
    break;
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    job ← strtonum(cp, 1, INT_MAX, &errstr);
    if (errstr) break;
    for (j ← job_list; j ≠ Λ; j ← j→next)
      if (j→job ≡ job) return j;
    break;
  case '?': ⟨ Look for a process matching %?⟨string⟩ 723 ⟩
    break;
  default: ⟨ Look for a process matching %⟨string⟩ 724 ⟩
    break;
  }
  if (ecodep) *ecodep ← JL_NOSUCH;
```

```
    return Λ;
  }
```

723. ⟨ Look for a process matching %?⟨*string*⟩ 723 ⟩ ≡

```
  last_match ← Λ;
  for (j ← job_list; j ≠ Λ; j ← j→next)
    for (p ← j→proc_list; p ≠ Λ; p ← p→next)
      if (strstr(p→command, cp + 1) ≠ Λ) {
        if (last_match) {
          if (ecodep) *ecodep ← JL_AMBIG;
          return Λ;
        }
        last_match ← j;
      }
  if (last_match) return last_match;
```

This code is used in section 722.

724. ⟨ Look for a process matching %⟨*string*⟩ 724 ⟩ ≡

```
  len ← strlen(cp);
  last_match ← Λ;
  for (j ← job_list; j ≠ Λ; j ← j→next)
    if (strncmp(cp, j→proc_list→command, len) ≡ 0) {
      if (last_match) {
        if (ecodep) *ecodep ← JL_AMBIG;
        return Λ;
      }
      last_match ← j;
    }
  if (last_match) return last_match;
```

This code is used in section 722.

725. Asynchronous Jobs. Return the pid of the last process in the last asynchronous **Job**.

⟨ jobs.c 666 ⟩ +≡

```
  pid_t j_async(void)
  {
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    if (async_job) async_job→flags |= JF_KNOWN;
    sigprocmask(SIG_SETMASK, &omask, Λ);
    return async_pid;
  }
```

726. Make j the last async process. * * Expects sigchld to be blocked. */

```
⟨ jobs.c 666 ⟩ +≡
  static void j_set_async(Job *j)
  {
    Job *jl, *oldest;
    if (async_job ∧ (async_job→flags & (JF_KNOWN | JF_ZOMBIE)) ≡ JF_ZOMBIE)
      remove_job(async_job, "async");
    if (¬(j→flags & JF_STARTED)) {
      internal_warningf("%s: job not started", __func__);
      return;
    }
    async_job ← j;
    async_pid ← j→last_proc→pid;
    while (nzombie > child_max) {⟨ Remove the oldest zombie from the job list 727 ⟩}
  }
```

727. It's not clear why zombies are tracked at all. The mean reason for *nzombies*, for example, seems to be to complain here if *nzombies* wasn't tracked correctly.

```
⟨ Remove the oldest zombie from the job list 727 ⟩ ≡
  oldest ← Λ;
  for (jl ← job_list; jl; jl ← jl→next)
    if (jl ≠ async_job ∧ (jl→flags & JF_ZOMBIE) ∧ (¬oldest ∨ jl→age < oldest→age)) oldest ← jl;
  if (¬oldest) {      /* XXX debugging */
    if (¬(async_job→flags & JF_ZOMBIE) ∨ nzombie ≠ 1) {
      internal_warningf("%s: bad nzombie (%d)", __func__, nzombie);
      nzombie ← 0;
    }
    break;
  }
  remove_job(oldest, "zombie");
```

This code is used in section 726.

728. Job Control Utilities. Suspend the shell.

```
⟨ jobs.c 666 ⟩ +≡
  void j_suspend(void)
  {
    struct sigaction sa, osa;
    ⟨ Restore tty & pgrp 729 ⟩
    ⟨ Suspend the shell 730 ⟩
    ⟨ Back from suspend—reset signals, pgrp & tty 731 ⟩
  }
```

729. ⟨ Restore tty & pgrp 729 ⟩ ≡

```
  if (ttypgrp_ok) {
    tcsetattr(tty_fd, TCSADRAIN, &tty_state);
    if (restore_ttypgrp ≥ 0) {
      if (tcsetpgrp(tty_fd, restore_ttypgrp) ≡ −1) {
        warningf(false, "%s: tcsetpgrp() failed: %s", __func__, strerror(errno));
      }
      else {
        if (setpgid(0, restore_ttypgrp) ≡ −1) {
          warningf(false, "%s: setpgid() failed: %s", __func__, strerror(errno));
        }
      }
    }
  }
```

This code is used in section 728.

730. ⟨ Suspend the shell 730 ⟩ ≡

```
  memset(&sa, 0, sizeof (sa));
  sigemptyset(&sa.sa_mask);
  sa.sa_handler ← SIG_DFL;
  sigaction (SIGTSTP, &sa, &osa);
  kill(0, SIGTSTP);
```

This code is used in section 728.

731. ⟨ Back from suspend—reset signals, pgrp & tty 731 ⟩ ≡

```
  sigaction (SIGTSTP, &osa, Λ);
  if (ttypgrp_ok) {
    if (restore_ttypgrp ≥ 0) {
      if (setpgid(0, kshpid) ≡ −1) {
        warningf(false, "%s: setpgid() failed: %s", __func__, strerror(errno));
        ttypgrp_ok ← 0;
      }
      else {
        if (tcsetpgrp(tty_fd, kshpid) ≡ −1) {
          warningf(false, "%s: tcsetpgrp() failed: %s", __func__, strerror(errno));
          ttypgrp_ok ← 0;
        }
      }
    }
    tty_init(true);
  }
```

This code is used in section 728.

732. Print job status in either short, medium or long format. POSIX doesn't say what to do if there is no process group leader (ie, ¬FMONITOR). We arbitrarily return last pid (which is what $! returns). Expects SIGCHLD to be blocked.

```
#define JP_NONE 0 /* don't print anything */
#define JP_SHORT 1 /* print signals that processes were killed by */
#define JP_MEDIUM 2 /* print "[job-num] ± command" */
#define JP_LONG 3 /* print "[job-num] ± pid command" */
#define JP_PGRP 4 /* print pgrp */
```

```
⟨ jobs.c 666 ⟩ +≡
  static void j_print(Job *j, int how, struct Shf *shf)
  {
    Proc *p;
    int state;
    int status;
    int coredumped;
    char jobchar ← ' ';
    char buf[64];
    const char *filler;
    int output ← 0;
    if (how ≡ JP_PGRP) {
      shf_fprintf(shf, "%d\n", j→pgrp ? j→pgrp : (j→last_proc ? j→last_proc→pid : 0));
      return;
    }
    j→flags &= ~JF_CHANGED;
    filler ← j→job > 10 ? "\n        " : "\n      ";
    if (j ≡ job_list) jobchar ← '+'; else if (j ≡ job_list→next) jobchar ← '-'; /* or ' ' */
    for (p ← j→proc_list; p ≠ Λ; ) {
      coredumped ← 0;
      ⟨ Stringify process state into buf 733 ⟩
      ⟨ Print a job ID and status 734 ⟩
      ⟨ Print a job's first line 735 ⟩
      state ← p→state;
      status ← p→status;
      p ← p→next;
      while (p ∧ p→state ≡ state ∧ p→status ≡ status) {
        ⟨ Print a job's remaining pipeline 736 ⟩
        p ← p→next;
      }
    }
    if (output) shf_fprintf(shf, "\n");
  }
```

733. ⟨ Stringify process state into *buf* 733 ⟩ ≡
```
switch (p→state) {
case PRUNNING:
  strlcpy(buf, "Running", sizeof buf);
  break;
case PSTOPPED:
  strlcpy(buf, sigtraps[WSTOPSIG(p→status)].mess, sizeof buf);
  break;
case PEXITED:
  if (how ≡ JP_SHORT) buf[0] ← '\0';
  else if (WEXITSTATUS(p→status) ≡ 0) strlcpy(buf, "Done", sizeof buf);
  else shf_snprintf(buf, sizeof (buf), "Done␣(%d)", WEXITSTATUS(p→status));
  break;
case PSIGNALLED:
  if (WCOREDUMP(p→status)) coredumped ← 1;
  if (how ≡ JP_SHORT ∧ ¬coredumped ∧
        (WTERMSIG(p→status) ≡ SIGINT ∨ WTERMSIG(p→status) ≡ SIGPIPE)) {
    buf[0] ← '\0';      /* kludge for not reporting "normal" termination signals */
  }
  else strlcpy(buf, sigtraps[WTERMSIG(p→status)].mess, sizeof buf);
  break;
}
```
This code is used in section 732.

734. ⟨ Print a job ID and status 734 ⟩ ≡
```
if (how ≠ JP_SHORT) {
  if (p ≡ j→proc_list) shf_fprintf(shf, "[%d]␣%c␣", j→job, jobchar);
  else shf_fprintf(shf, "%s", filler);
}
```
This code is used in section 732.

735. ⟨ Print a job's first line 735 ⟩ ≡
```
if (how ≡ JP_LONG) shf_fprintf(shf, "%5d␣", p→pid);
if (how ≡ JP_SHORT) {
  if (buf[0]) {
    output ← 1;
    shf_fprintf(shf, "%s%s␣", buf, coredumped ? "␣(core␣dumped)" : "");
  }
}
else {
  output ← 1;
  shf_fprintf(shf, "%-20s␣%s%s%s", buf, p→command, p→next ? "|" : "",
      coredumped ? "␣(core␣dumped)" : "");
}
```
This code is used in section 732.

736. ⟨ Print a job's remaining pipeline 736 ⟩ ≡
```
if (how ≡ JP_LONG)
  shf_fprintf(shf, "%s%5d␣%-20s␣%s%s", filler, p→pid, "␣", p→command, p→next ? "|" : "");
else if (how ≡ JP_MEDIUM)
  shf_fprintf(shf, "␣%s%s", p→command, p→next ? "|" : "");
```
This code is used in section 732.

737. Are there any running or stopped jobs?

```
⟨ jobs.c 666 ⟩ +≡
  int j_stopped_running(void)
  {
    Job *j;
    int which ← 0;
    for (j ← job_list; j ≠ Λ; j ← j→next) {
      if (j→ppid ≡ procpid ∧ j→state ≡ PSTOPPED) which |= 1;
      if (Flag(FLOGIN) ∧ ¬Flag(FNOHUP) ∧ procpid ≡ kshpid ∧ j→ppid ≡ procpid ∧ j→state ≡ PRUNNING)
        which |= 2;
    }
    if (which) {
      shellf("You␣have␣%s%s%s␣jobs\n",
          which & 1 ? "stopped" : "",
          which ≡ 3 ? "␣and␣" : "",
          which & 2 ? "running" : "");
      return 1;
    }
    return 0;
  }
```

738. Returns the number of jobs for printing.

```
⟨ jobs.c 666 ⟩ +≡
  int j_njobs(void)
  {
    Job *j;
    int nj ← 0;
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    for (j ← job_list; j; j ← j→next) nj++;
    sigprocmask(SIG_SETMASK, &omask, Λ);
    return nj;
  }
```

739. List jobs for `jobs` built-in.

```
⟨ jobs.c 666 ⟩ +≡
  int j_jobs(const char *cp, int slp, int nflag)      /* slp – 0: short, 1: long, 2: pgrp */
  {
    Job *j, *tmp;
    int how;
    int zflag ← 0;
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    if (nflag < 0) {      /* kludge: print zombies */
      nflag ← 0;
      zflag ← 1;
    }
    if (cp) {
      int ecode;
      if ((j ← j_lookup(cp, &ecode)) ≡ Λ) {
        sigprocmask(SIG_SETMASK, &omask, Λ);
        bi_errorf("%s: %s", cp, lookup_msgs[ecode]);
        return 1;
      }
    }
    else j ← job_list;
    how ← slp ≡ 0 ? JP_MEDIUM : (slp ≡ 1 ? JP_LONG : JP_PGRP);
    for ( ; j; j ← j→next) {
      if ((¬(j→flags & JF_ZOMBIE) ∨ zflag) ∧ (¬nflag ∨ (j→flags & JF_CHANGED))) {
        j_print(j, how, shl_stdout);
        if (j→state ≡ PEXITED ∨ j→state ≡ PSIGNALLED) j→flags |= JF_REMOVE;
      }
      if (cp) break;
    }
    for (j ← job_list; j; j ← tmp) {
      /* Remove jobs after printing so there won't be multiple + or - jobs */
      tmp ← j→next;
      if (j→flags & JF_REMOVE) remove_job(j, "jobs");
    }
    sigprocmask(SIG_SETMASK, &omask, Λ);
    return 0;
  }
```

740. List jobs for top-level notification.

```
⟨ jobs.c 666 ⟩ +≡
  void j_notify(void)
  {
    Job *j, *tmp;
    sigset_t omask;
    sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);
    for (j ← job_list; j; j ← j→next) {
      if (Flag(FMONITOR) ∧ (j→flags & JF_CHANGED)) j_print(j, JP_MEDIUM, shl_out);
      if (j→state ≡ PEXITED ∨ j→state ≡ PSIGNALLED) j→flags |= JF_REMOVE;
          /* Remove jobs after doing reports so there won't be multiple + or - jobs */
    }
    for (j ← job_list; j; j ← tmp) {
      tmp ← j→next;
      if (j→flags & JF_REMOVE) remove_job(j, "notify");
    }
    shf_flush(shl_out);
    sigprocmask(SIG_SETMASK, &omask, Λ);
  }
```

741. Filesystem. Handling path names and directories gets its own file, originally by Larry Bouzane.

⟨ path.c 741 ⟩ ≡
```
#include <sys/stat.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
  static char *do_phys_path(XString *, char *, const char *);
```
See also sections 745, 747, 753, 757, and 758.

742. ⟨ Shared function declarations 4 ⟩ +≡
```
  int make_path(const char *, const char *, char **, XString *, int *);
  void simplify_path(char *);
  char *get_phys_path(const char *);
  void set_current_wd(char *);
```

743. The current working directory is a property known to the kernel. The shell keeps a local copy, insofar as it can, in *current_wd*.

⟨ Global variables 5 ⟩ +≡
```
  char *current_wd;
  int current_wd_size;
```

744. ⟨ Externally-linked variables 6 ⟩ +≡
```
  extern char *current_wd;
  extern int current_wd_size;
```

745. Changes to the current working directory (ie. *chdir*) are recorded by *set_current_wd*.

⟨ path.c 741 ⟩ +≡
```
  void set_current_wd(char *path)
  {
    int len;
    char *p ← path;
    if (¬p ∧ ¬(p ← ksh_get_wd(Λ, 0))) p ← null;
    len ← strlen(p) + 1;
    if (len > current_wd_size) current_wd ← aresize(current_wd, current_wd_size ← len, APERM);
    memcpy(current_wd, p, len);
    if (p ≠ path ∧ p ≠ null) afree(p, ATEMP);
  }
```

746. Like *getcwd* except *bsize* is ignored if *buf* is 0 (PATH_MAX is used). Using *getcwd* would require that space always be allocated for the result.

```
⟨ misc.c 9 ⟩ +≡
  char *ksh_get_wd(char *buf, int bsize)
  {
    char *b;
    char *ret;
    if (¬buf) {
      bsize ← PATH_MAX;
      b ← alloc(PATH_MAX + 1, ATEMP);
    }
    else b ← buf;
    ret ← getcwd(b, bsize);
    if (¬buf) {
      if (ret) ret ← aresize(b, strlen(b) + 1, ATEMP);
      else afree(b, ATEMP);
    }
    return ret;
  }
```

747. Path Names. Makes a filename (saving it into *xsp*) using the following algorithm:

* Start with the empty string.
* If *file* starts with "/", append *file* and set *cdpathp* ← Λ.
* If *file* starts with "./" or "../" (or is exactly "." or ".."), append *cwd* & *file* and set *cdpathp* ← Λ.
* If the first element of *cdpathp* doesn't start with a "/" or ".", *cwd* is appended.
* The first element of *cdpathp* is appended.
* *file* is appended.
* *cdpathp* is set to the start of the next element in *cdpathp* (or Λ if there are no more).

The return value indicates whether a non-zero length element from *cdpathp* was appended to result.

```
⟨ path.c 741 ⟩ +≡
  int make_path(const char *cwd, const char *file, char **cdpathp, XString *xsp, int *phys_pathp)
  {
    int rval ← 0;
    int use_cdpath ← 1;
    char *plist;
    int len;
    int plen ← 0;
    char *xp ← Xstring(*xsp, xp);
    if (¬file) file ← null;
    if (file[0] ≡ '/') {
      *phys_pathp ← 0;
      use_cdpath ← 0;
    }
    else {
      if (file[0] ≡ '.') {⟨ Check for ./ or ../ 748 ⟩}
      if (¬(plist ← *cdpathp)) use_cdpath ← 0;
      else if (use_cdpath) {⟨ "Extract" the first element of cdpathp 749 ⟩}
      if ((use_cdpath ≡ 0 ∨ ¬plen ∨ plist[0] ≠ '/') ∧ (cwd ∧ *cwd)) {⟨ Path is relative—append cwd 750 ⟩}
      *phys_pathp ← Xlength(*xsp, xp);
      if (use_cdpath ∧ plen) {⟨ Append the extraction from cdpathp 751 ⟩}
    }
    ⟨ Append file 752 ⟩
    if (¬use_cdpath) *cdpathp ← Λ;
    return rval;
  }
```

748. ⟨ Check for ./ or ../ 748 ⟩ ≡

```
  char c ← file[1];
  if (c ≡ '.') c ← file[2];
  if (c ≡ '/' ∨ c ≡ '\0') use_cdpath ← 0;
```

This code is used in section 747.

749. Find the next ":" in *cdpathp* and replace it with '\0'.

⟨ "Extract" the first element of *cdpathp* 749 ⟩ ≡

```
  char *pend;
  for (pend ← plist; *pend ∧ *pend ≠ ':'; pend++) ;
  plen ← pend − plist;
  *cdpathp ← *pend ? ++pend : Λ;
```

This code is used in section 747.

750. ⟨ Path is relative—append *cwd* 750 ⟩ ≡

```
len ← strlen(cwd);
XcheckN(*xsp, xp, len);
memcpy(xp, cwd, len);
xp += len;
if (cwd[len − 1] ≠ '/') Xput(*xsp, xp, '/');      /* Tack on a / if necessary */
```

This code is used in section 747.

751. ⟨ Append the extraction from *cdpathp* 751 ⟩ ≡

```
XcheckN(*xsp, xp, plen);
memcpy(xp, plist, plen);
xp += plen;
if (plist[plen − 1] ≠ '/') Xput(*xsp, xp, '/');      /* — „ — */
rval ← 1;
```

This code is used in section 747.

752. ⟨ Append *file* 752 ⟩ ≡

```
len ← strlen(file) + 1;
XcheckN(*xsp, xp, len);
memcpy(xp, file, len);
```

This code is used in section 747.

753. Simplify in-place pathnames containing "`.`" and "`..`" entries, ie. *simplify_path*(`"/a/b/c/./../d/.."`) returns "`/a/b`". Also squashes long strings of `/`s.

`/foo/` → `/foo`
`/foo/../../bar` → `/bar`
`/foo/./bar/..` → `/foo`
`.` → `.`
`..` → `..`
`./foo` → `foo`
`foo/../../../bar` → `../../bar`

To perform, *cur* and *t* are set to the first character (after `/` if the path is absolute, which is noted). *cur* marks the position characters will be written to while *t* where to examine next.

```
⟨ path.c 741 ⟩ +≡
  void simplify_path(char *path)
  {
    char *cur;
    char *t;
    int isrooted;
    char *very_start ← path;
    char *start;
    if (¬*path) return;
    if ((isrooted ← (path[0] ≡ '/'))) very_start++;
    for (cur ← t ← start ← very_start; ; ) {
      while (*t ≡ '/') t++;      /* skip extra /s */
      if (*t ≡ '\0') {⟨ Convert an empty path to "." and break 754 ⟩}
      if (t[0] ≡ '.') {
        if (¬t[1] ∨ t[1] ≡ '/') {⟨ Skip "./" and continue 755 ⟩}
        else if (t[1] ≡ '.' ∧ (¬t[2] ∨ t[2] ≡ '/')) {⟨ Remove a "../" component and continue 756 ⟩}
      }
      if (cur ≠ very_start) *cur++ ← '/';      /* Append "/" except at the beginning */
      while (*t ∧ *t ≠ '/') *cur++ ← *t++;      /* Move the path component and locate the next */
    }
  }
```

754. ⟨ Convert an empty path to "." and break 754 ⟩ ≡

```
  if (cur ≡ path) *cur++ ← '.';
  *cur ← '\0';
  break;
```

This code is used in section 753.

755. ⟨ Skip "./" and continue 755 ⟩ ≡

```
  t += 1;
  continue;
```

This code is used in section 753.

756. ⟨ Remove a "../" component and **continue** 756 ⟩ ≡

```
if (¬isrooted ∧ cur ≡ start) {
  if (cur ≠ very_start) *cur++ ← '/';
  *cur++ ← '.';
  *cur++ ← '.';
  start ← cur;
}
else if (cur ≠ start)
  while (--cur > start ∧ *cur ≠ '/') ;
t += 2;
continue;
```

This code is used in section 753.

757. Find the physical name from a path which may include symbolic links.

⟨ path.c 741 ⟩ +≡

```
char *get_phys_path(const char *path)
{
  XString xs;
  char *xp;
  Xinit(xs, xp, strlen(path) + 1, ATEMP);
  xp ← do_phys_path(&xs, xp, path);
  if (¬xp) return Λ;
  if (Xlength(xs, xp) ≡ 0) Xput(xs, xp, '/');
  Xput(xs, xp, '\0');
  return Xclose(xs, xp);
}
```

758. There is no need for *xsp* to be a **XString** ∗.

⟨ path.c 741 ⟩ +≡

```
static char *do_phys_path(XString *xsp, char *xp, const char *path)
{
  const char *p, *q;
  int len, llen;
  int savepos;
  char lbuf[PATH_MAX];
  Xcheck(*xsp, xp);
  for (p ← path; p; p ← q) {
    while (*p ≡ '/') p++;      /* skip extra /s */
    if (¬*p) break;
    len ← (q ← strchr(p, '/')) ? (size_t)(q − p) : strlen(p);
    if (len ≡ 1 ∧ p[0] ≡ '.') continue;      /* skip "./" */
    if (len ≡ 2 ∧ p[0] ≡ '.' ∧ p[1] ≡ '.') {⟨ Remove a path component at xp and continue 759 ⟩}
    ⟨ Append the path component at xp 760 ⟩
    ⟨ Resolve a symlink in the path so far or continue 761 ⟩
    ⟨ Reset xp if the symlink is absolute 762 ⟩
  }
  return xp;
}
```

759. ⟨ Remove a path component at *xp* and **continue** 759 ⟩ ≡
```
while (xp > Xstring(∗xsp, xp)) {
  xp −−;
  if (∗xp ≡ '/') break;
}
continue;
```
This code is used in section 758.

760. ⟨ Append the path component at *xp* 760 ⟩ ≡
```
savepos ← Xsavepos(∗xsp, xp);
Xput(∗xsp, xp, '/');
XcheckN(∗xsp, xp, len + 1);
memcpy(xp, p, len);
xp += len;
∗xp ← '\0';
```
This code is used in section 758.

761. ⟨ Resolve a symlink in the path so far or **continue** 761 ⟩ ≡
```
llen ← readlink(Xstring(∗xsp, xp), lbuf, sizeof (lbuf) − 1);
if (llen ≡ −1) {
  if (errno ≠ EINVAL) return Λ;
  continue;      /∗ wasn't a symlink ∗/
}
lbuf[llen] ← '\0';
```
This code is used in section 758.

762. ⟨ Reset *xp* if the symlink is absolute 762 ⟩ ≡
```
xp ← lbuf[0] ≡ '/' ? Xstring(∗xsp, xp) : Xrestpos(∗xsp, xp, savepos);
if (¬(xp ← do_phys_path(xsp, xp, lbuf))) return Λ;
```
This code is used in section 758.

763. To search for a command in `$PATH` or for a function in `$FPATH` *search* checks each item in *path*.

```
⟨ exec.c 461 ⟩ +≡
  char *search(const char *name, const char *path,
  int mode,      /* R_OK or X_OK */
  int *errnop)      /* set if a candidate is found but unsuitable */
  {
    const char *sp, *p;
    char *xp;
    XString xs;
    int namelen;
    if (errnop) *errnop ← 0;
    if (strchr(name, '/')) {
      if (search_access(name, mode, errnop) ≡ 0) return (char *) name;
      return Λ;
    }
    namelen ← strlen(name) + 1;
    Xinit(xs, xp, 128, ATEMP);
    sp ← path;
    while (sp ≠ Λ) {⟨ Search for name in each entry in path and return it 764 ⟩}
    Xfree(xs, xp);
    return Λ;
  }
```

764. Characters are copied from *path* until "`:`" or the end, then the desired filename is appended and the full pathname is checked by *search_access*.

```
⟨ Search for name in each entry in path and return it 764 ⟩ ≡
  xp ← Xstring(xs, xp);
  if (¬(p ← strchr(sp, ':'))) p ← sp + strlen(sp);
  if (p ≠ sp) {
    XcheckN(xs, xp, p − sp);
    memcpy(xp, sp, p − sp);
    xp += p − sp;
    *xp++ ← '/';
  }
  sp ← p;
  XcheckN(xs, xp, namelen);
  memcpy(xp, name, namelen);
  if (search_access(Xstring(xs, xp), mode, errnop) ≡ 0) return Xclose(xs, xp + namelen);
  if (*sp++ ≡ '\0') sp ← Λ;
```

This code is used in section 763.

765. A file merely existing is not enough, it must also be accessible and usable. *errnop* is set if a candidate is found but is unsuitable.

statb.st_mode is checked twice because *access* says root can execute everything.

```
⟨ exec.c 461 ⟩ +≡
  int search_access(const char *path, int mode, int *errnop)
  {
    int ret, err ← 0;
    struct stat statb;
    if (stat(path, &statb) ≡ −1) return −1;
    ret ← access(path, mode);
    if (ret ≡ −1) err ← errno;      /* File exists, but we can't access it */
    else if (mode ≡ X_OK ∧
          (¬S_ISREG(statb.st_mode) ∨
          ¬(statb.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH)))) {
      ret ← −1;
      err ← S_ISDIR(statb.st_mode) ? EISDIR : EACCES;
    }
    if (err ∧ errnop ∧ ¬*errnop) *errnop ← err;
    return ret;
  }
```

766. Globbing. If there were a `glob.c` this would be at the top of it. Or perhaps into `eval.c`.

```
#define XCF_COMMAND BIT(0) /* Do command completion */
#define XCF_FILE BIT(1) /* Do file completion */
#define XCF_FULLPATH BIT(2) /* command completion: store full path */
#define XCF_COMMAND_FILE (XCF_COMMAND | XCF_FILE)
```

⟨ Globbing 766 ⟩ ≡

```
static char *add_glob(const char *str, int slen);
static void glob_table(const char *pat, XPtrV *wp, struct table *tp);
static void glob_path(int flags, const char *pat, XPtrV *wp, const char *path);
static int path_order_cmp(const void *aa, const void *bb);
```

See also sections 770, 771, 773, 775, 778, 779, 781, 791, 796, 799, 802, 803, and 811.

This code is used in section 866.

767. In order to glob a string the lexical analyser injects `MAGIC` whenever it encounters a globbing character in a location where it can perform.

`MAGIC` is a character that might be printed to make bugs more obvious but not a character that is used often. Also can't use the high bit as it causes portability problems (calling *strchr* (*x*, 0*x*80|'x') is error prone)[1].

```
#define MAGIC (7) /* prefix for *?[!{,} during expand */
#define ISMAGIC(c) ((unsigned char)(c) ≡ MAGIC)
```

⟨ Define `MAGIC` 767 ⟩ ≡

This code is used in section 7.

768. A string is *debunk*ed to remove its `MAGIC` without applying it.

⟨ `eval.c` 554 ⟩ +≡

```
char *debunk(char *dp, const char *sp, size_t dlen)
{
  char *d, *s;
  if ((s ← strchr(sp, MAGIC))) {
    size_t slen ← s − sp;
    if (slen ≥ dlen) return dp;
    memcpy(dp, sp, slen);
    for (d ← dp + slen; *s ∧ (d < dp + dlen); s++)
      if (¬ISMAGIC(*s) ∨ ¬(*++s & 0x80) ∨ ¬strchr("*+?@!␣", *s & 0x7f)) *d++ ← *s;
      else { /* extended pattern operators: [*+?@!␣] */
        if ((*s & 0x7f) ≠ '␣') *d++ ← *s & 0x7f;
        if (d < dp + dlen) *d++ ← '(';
      }
    *d ← '\0';
  }
  else if (dp ≠ sp) strlcpy(dp, sp, dlen);
  return dp;
}
```

[1] ie. It's already used for something else.

769. String expansion uses the eponymously named[1] *glob* as its sole entry-point to the globbing routines. TODO: *cp* is not **const** because slashes are temporarily replaced with `'\0'`.

```
⟨ eval.c 554 ⟩ +≡
  static void glob(char *cp, XPtrV *wp, int markdirs)
  {
    int oldsize ← XPsize(*wp);
    if (glob_str(cp, wp, markdirs) ≡ 0) XPput(*wp, debunk(cp, cp, strlen(cp) + 1));
    else qsortp(XPptrv(*wp) + oldsize, (size_t)(XPsize(*wp) − oldsize), xstrcmp);
  }
```

770. Don't do command globing on zero length strings—it takes too long and isn't very useful. File globs are more likely to be useful, so allow these.

```
⟨ Globbing 766 ⟩ +≡
  static int x_try_array(const char *, int, const char *, int, int *, char ***);
  int x_cf_glob(int flags, const char *buf, int buflen, int pos, int *startp, int *endp, char ***wordsp, int
          *is_commandp)
  {
    int len;
    int nwords;
    char **words ← Λ;
    int is_command;
    len ← x_locate_word(buf, buflen, pos, startp, &is_command);
    if (¬(flags & XCF_COMMAND)) is_command ← 0;
    if (len ≡ 0 ∧ is_command) return 0;
    if (is_command) nwords ← x_command_glob(flags, buf + *startp, len, &words);
    else if (¬x_try_array(buf, buflen, buf + *startp, len, &nwords, &words))
      nwords ← x_file_glob(flags, buf + *startp, len, &words);
    if (nwords ≡ 0) {
      *wordsp ← Λ;
      return 0;
    }
    if (is_commandp) *is_commandp ← is_command;
    *wordsp ← words;
    *endp ← *startp + len;
    return nwords;
  }
```

[1] Derived from Version 6 Unix's `/etc/glob` program that expanded filenames.

771. Before searching the filesystem ksh searches for built-ins etc.

```
⟨ Globbing 766 ⟩ +≡
  static int x_command_glob(int flags, const char *str, int slen, char ***wordsp)
  {
    char *toglob;
    char *pat;
    char *fpath;
    int nwords;
    XPtrV w;
    struct Block *l;
    if (slen < 0) return 0;
    ⟨ Convert the search string into a pattern 772 ⟩
    XPinit(w, 32);
    ⟨ Search tables for an internal command or a function 774 ⟩
    nwords ← XPsize(w);
    if (¬nwords) {
      *wordsp ← Λ;
      XPfree(w);
      return 0;
    }
    if (flags & XCF_FULLPATH) {⟨ Sort by basename, then path order 776 ⟩}
    else {⟨ Sort and remove duplicate entries 780 ⟩}
    XPput(w, Λ);
    *wordsp ← (char **) XPclose(w);
    return nwords;
  }
```

772. See also the implementation od *add_glob* in the next section.

```
⟨ Convert the search string into a pattern 772 ⟩ ≡
  toglob ← add_glob(str, slen);
  pat ← evalstr(toglob, DOPAT | DOTILDE);
  afree(toglob, ATEMP);
```

This code is used in section 771.

773. Given a string, copy it and possibly add a "*" to the end. The new string is returned.

If the pathname contains a wildcard (an unquoted "*", "?" or "["), parameter expansion ("$") or "~username" with no trailing slash, then it is globbed based on that value (ie. without the appended "*").

```
⟨ Globbing 766 ⟩ +≡
  static char *add_glob(const char *str, int slen)
  {
    char *toglob;
    char *s;
    bool saw_slash ← false;
    if (slen < 0) return Λ;
    toglob ← str_nsave(str, slen + 1, ATEMP);      /* One longer for the * */
    toglob[slen] ← '\0';
    for (s ← toglob; *s; s++) {
      if (*s ≡ '\\' ∧ s[1]) s++;
      else if (*s ≡ '*' ∨ *s ≡ '[' ∨ *s ≡ '?' ∨ *s ≡ '$' ∨ (s[1] ≡ '(' ∧ strchr("+@!", *s))) break;
      else if (*s ≡ '/') saw_slash ← true;
    }
    if (¬*s ∧ (*toglob ≠ '~' ∨ saw_slash)) {
      toglob[slen] ← '*';
      toglob[slen + 1] ← '\0';
    }
    return toglob;
  }
```

774. ⟨ Search tables for an internal command or a function 774 ⟩ ≡

```
  glob_table(pat, &w, &keywords);
  glob_table(pat, &w, &aliases);
  glob_table(pat, &w, &builtins);
  for (l ← genv→loc; l; l ← l→next) glob_table(pat, &w, &l→funs);
  glob_path(flags, pat, &w, search_path);
  if ((fpath ← str_val(global("FPATH"))) ≠ null) glob_path(flags, pat, &w, fpath);
```

This code is used in section 771.

775. Apply pattern matching to a table; all table entries that match a pattern are added to *wp*.

```
⟨ Globbing 766 ⟩ +≡
  static void glob_table(const char *pat, XPtrV *wp, struct table *tp)
  {
    struct tstate ts;
    struct tbl *te;
    for (ktwalk(&ts, tp); (te ← ktnext(&ts)); ) {
      if (gmatch(te→name, pat, false)) XPput(*wp, str_save(te→name, ATEMP));
    }
  }
```

776. ⟨ Sort by basename, then path order 776 ⟩ ≡

```
struct path_order_info *info;
struct path_order_info *last_info ← Λ;
char **words ← (char **) XPptrv(w);
int path_order ← 0;
int i;
info ← areallocarray(Λ, nwords, sizeof(struct path_order_info), ATEMP);
for (i ← 0; i < nwords; i++) {
  info[i].word ← words[i];
  info[i].base ← x_basename(words[i], Λ);
  if (¬last_info ∨ info[i].base ≠ last_info→base ∨ strncmp(words[i], last_info→word, info[i].base) ≠ 0) {
    last_info ← &info[i];
    path_order++;
  }
  info[i].path_order ← path_order;
}
qsort(info, nwords, sizeof(struct path_order_info), path_order_cmp);
for (i ← 0; i < nwords; i++) words[i] ← info[i].word;
afree(info, ATEMP);
```

This code is used in section 771.

777. To compare results uses this object.

⟨ Type definitions 17 ⟩ +≡

```
struct path_order_info {
  char *word;
  int base;
  int path_order;
};
```

778. ⟨ Globbing 766 ⟩ +≡

```
static int path_order_cmp(const void *aa, const void *bb)
{
  const struct path_order_info *a ← (const struct path_order_info *) aa;
  const struct path_order_info *b ← (const struct path_order_info *) bb;
  int t;
  t ← strcmp(a→word + a→base, b→word + b→base);
  return t ? t : a→path_order − b→path_order;
}
```

779.
Return the offset of the basename of string s which need not be `'\0'`-terminated if it ends at se. Trailing slashes are ignored. If s is just a slash then the offset is 0 (actually, length - 1).

* "`/etc`": 1
* "`/etc/`": 1
* "`/etc//`": 1
* "`/etc/fo`": 5
* "`foo`": 0
* "`///`": 2
* "": 0

```
⟨ Globbing 766 ⟩ +≡
  int x_basename(const char *s, const char *se)
  {
    const char *p;
    if (se ≡ Λ) se ← s + strlen(s);
    if (s ≡ se) return 0;
    for (p ← se − 1; p > s ∧ *p ≡ '/'; p−−) ;      /* Skip trailing slashes */
    for ( ; p > s ∧ *p ≠ '/'; p−−) ;      /* Skip over everything that isn't a slash */
    if (*p ≡ '/' ∧ p + 1 < se) p++;
    return p − s;
  }
```

780. ⟨ Sort and remove duplicate entries 780 ⟩ ≡

```
  char **words ← (char **) XPptrv(w);
  int i, j;
  qsortp(XPptrv(w), (size_t) nwords, xstrcmp);
  for (i ← j ← 0; i < nwords − 1; i++) {
    if (strcmp(words[i], words[i + 1])) words[j++] ← words[i];
    else afree(words[i], ATEMP);
  }
  words[j++] ← words[i];
  nwords ← j;
  w.cur ← (void **) &words[j];
```

This code is used in section 771.

781. ⟨ Globbing 766 ⟩ +≡

```
static void glob_path(int flags, const char *pat, XPtrV *wp, const char *path)
{
  const char *sp, *p;
  char *xp;
  int staterr;
  int pathlen;
  int patlen;
  int oldsize, newsize, i, j;
  char **words;
  XString xs;
  patlen ← strlen(pat) + 1;
  sp ← path;
  Xinit(xs, xp, patlen + 128, ATEMP);
  while (sp) {
    xp ← Xstring(xs, xp);
    if (¬(p ← strchr(sp, ':'))) p ← sp + strlen(sp);
    pathlen ← p − sp;
    if (pathlen) {⟨ Copy the next entry from path into xp 782 ⟩}
    sp ← p;
    XcheckN(xs, xp, patlen);
    memcpy(xp, pat, patlen);
    oldsize ← XPsize(*wp);
    glob_str(Xstring(xs, xp), wp, 1);      /* mark dirs */
    newsize ← XPsize(*wp);
    ⟨ Filter the globbed list for executables 783 ⟩
    if (¬*sp++) break;
  }
  Xfree(xs, xp);
}
```

782. Copy sp into xp, stuffing any MAGIC characters on the way (Making the MAGIC itself MAGICal?)

⟨ Copy the next entry from *path* into *xp* 782 ⟩ ≡

```
const char *s ← sp;
XcheckN(xs, xp, pathlen * 2);
while (s < p) {
  if (ISMAGIC(*s)) *xp++ ← MAGIC;
  *xp++ ← *s++;
}
*xp++ ← '/';
pathlen++;
```

This code is used in section 781.

783. ⟨ Filter the globbed list for executables 783 ⟩ ≡

```
words ← (char **) XPptrv(*wp);
for (i ← j ← oldsize; i < newsize; i++) {
  staterr ← 0;
  if ((search_access(words[i], X_OK, &staterr) ≥ 0) ∨ (staterr ≡ EISDIR)) {
    words[j] ← words[i];
    if (¬(flags & XCF_FULLPATH)) memmove(words[j], words[j] + pathlen, strlen(words[j] + pathlen) + 1);
    j++;
  }
  else afree(words[i], ATEMP);
}
wp→cur ← (void **) &words[j];
```

This code is used in section 781.

784. Apply file globbing to cp and store the matching files in wp. Returns the number of matches found.

⟨ eval.c 554 ⟩ +≡

```
int glob_str(char *cp, XPtrV *wp, int markdirs)
{
  int oldsize ← XPsize(*wp);
  XString xs;
  char *xp;
  Xinit(xs, xp, 256, ATEMP);
  globit(&xs, &xp, cp, wp, markdirs ? GF_MARKDIR : GF_NONE);
  Xfree(xs, xp);
  return XPsize(*wp) − oldsize;
}
```

785. ... Implemented here.

```
#define GF_NONE 0
#define GF_EXCHECK BIT(0)      /* do existence check on file */
#define GF_GLOBBED BIT(1)      /* some globbing has been done */
#define GF_MARKDIR BIT(2)      /* add trailing / to directories */
```

⟨ eval.c 554 ⟩ +≡

```
static void globit(XString *xs,      /* destination string */
char **xpp,      /* pointer to destination end */
char *sp,      /* source path */
XPtrV *wp,      /* output list */
int check)      /* GF_* flags */
{
  char *np;      /* next source component */
  char *xp ← *xpp;
  char *se;
  char odirsep;      /* don't assume "/"—can[not][1] be multiple kinds */
  intrcheck( );      /* allow long-running expansions to be interrupted */
  if (sp ≡ Λ) {⟨ End of source path and return 786 ⟩}
  if (xp > Xstring(*xs, xp)) *xp++ ← '/';      /* Append "/" */
  while (*sp ≡ '/') {      /* Copy "/" from sp until there are none left */
    Xcheck(*xs, xp);
    *xp++ ← *sp++;
  }
  np ← strchr(sp, '/');      /* Find the next "/" */
  if (np ≠ Λ) {      /* Replace the "/" with '\0' */
    se ← np;
    odirsep ← *np;
    *np++ ← '\0';
  }
  else {
    odirsep ← '\0';      /* keep gcc quiet */
    se ← sp + strlen(sp);
  }
  if (¬has_globbing(sp, se)) {⟨ Remove MAGIC and globit again 789 ⟩}
  else {⟨ Apply the MAGIC glob 790 ⟩}
  if (np ≠ Λ) *--np ← odirsep;
}
```

[1] This looks like legacy code—this appears to be the only place this assumption is **not** made.

786. We only need to check if the file exists if a pattern is followed by a non-pattern (eg. "`foo*x/bar`"; no check is needed for "`foo*`" since the match must exist) or if any patterns were expanded and the markdirs option is set. Symlinks make things a bit tricky...

```
#define stat_check()
        (stat_done ? stat_done : (stat_done ← stat(Xstring(*xs, xp), &statb) ≡ −1 ? −1 : 1))
⟨ End of source path and return 786 ⟩ ≡
  if ((check & GF_EXCHECK) ∨ ((check & GF_MARKDIR) ∧ (check & GF_GLOBBED))) {
    struct stat lstatb, statb;
    int stat_done ← 0;      /* -1: failed, 1 ok */
    if (lstat(Xstring(*xs, xp), &lstatb) ≡ −1) return;
    ⟨ Permit files to look like directories 787 ⟩
    ⟨ Make directories look like directories 788 ⟩
  }
  XPput(*wp, str_nsave(Xstring(*xs, xp), Xlength(*xs, xp), ATEMP));
  return;
```

This code is used in section 785.

787. special case for systems which strip trailing slashes from regular files (eg. `/etc/passwd/`). SunOS 4.1.3 does this...

```
⟨ Permit files to look like directories 787 ⟩ ≡
  if ((check & GF_EXCHECK) ∧ xp > Xstring(*xs, xp) ∧
        xp[−1] ≡ '/' ∧ ¬S_ISDIR(lstatb.st_mode) ∧
        (¬S_ISLNK(lstatb.st_mode) ∨ stat_check( ) < 0 ∨ ¬S_ISDIR(statb.st_mode)))
    return;
```

This code is used in section 786.

788. Possibly tack on a trailing / if there isn't already one and if the file is a directory or a symlink to a directory

```
⟨ Make directories look like directories 788 ⟩ ≡
  if (((check & GF_MARKDIR) ∧ (check & GF_GLOBBED)) ∧ xp > Xstring(*xs, xp) ∧ xp[−1] ≠ '/' ∧
    (S_ISDIR(lstatb.st_mode) ∨
        (S_ISLNK(lstatb.st_mode) ∧ stat_check( ) > 0 ∧ S_ISDIR(statb.st_mode)))) {
    *xp++ ← '/';
    *xp ← '\0';
  }
```

This code is used in section 786.

789. Avoid pattern checks for strings containing `MAGIC`, open "`[`" without the closing "`]`", etc. otherwise the call to *opendir* may fail due to an invalid directory name. Only execute permission should be required as per POSIX.

```
⟨ Remove MAGIC and globit again 789 ⟩ ≡
  XcheckN(*xs, xp, se − sp + 1);
  debunk(xp, sp, Xnleft(*xs, xp));
  xp += strlen(xp);
  *xpp ← xp;
  globit(xs, xpp, np, wp, check);
```

This code is used in section 785.

790. ⟨ Apply the MAGIC glob 790 ⟩ ≡

```
DIR * dirp;
struct dirent *d;
char *name;
int len;
int prefix_len;
*xp ← '\0';
prefix_len ← Xlength(*xs, xp);
dirp ← opendir(prefix_len ? Xstring(*xs, xp) : ".");
if (dirp ≡ Λ) goto Nodir;
while ((d ← readdir(dirp)) ≠ Λ) {
  name ← d→d_name;
  if (name[0] ≡ '.' ∧ (name[1] ≡ 0 ∨ (name[1] ≡ '.' ∧ name[2] ≡ 0))) continue;
      /* always ignore . and .. */
  if ((*name ≡ '.' ∧ *sp ≠ '.') ∨ ¬gmatch(name, sp, true)) continue;
  len ← strlen(d→d_name) + 1;
  XcheckN(*xs, xp, len);
  memcpy(xp, name, len);
  *xpp ← xp + len − 1;
  globit(xs, xpp, np, wp, (check & GF_MARKDIR) | GF_GLOBBED | (np ? GF_EXCHECK : GF_NONE));
  xp ← Xstring(*xs, xp) + prefix_len;
}
closedir(dirp);
Nodir: ;
```

This code is used in section 785.

791. If not looking up a command, try looking an argument for a comment.

```
⟨ Globbing 766 ⟩ +≡
static int x_try_array(const char *buf, int buflen, const char *want, int wantlen, int *nwords, char
        ***words)
{
  const char *cmd, *cp;
  int cmdlen, n, i, slen;
  char *name, *s;
  struct tbl *v, *vp;
  *nwords ← 0;
  *words ← Λ;
  ⟨ Walk back to find start of command 792 ⟩
  ⟨ Take a stab at argument count from here 793 ⟩
  ⟨ Try to find the array 794 ⟩
  ⟨ Walk the array and build words list 795 ⟩
  return *nwords ≠ 0;
}
```

792. ⟨ Walk back to find start of command 792 ⟩ ≡
```
if (want ≡ buf) return 0;
for (cmd ← want; cmd > buf; cmd−−) {
  if (strchr(";|&()‘", cmd[−1]) ≠ Λ) break;
}
while (cmd < want ∧ isspace((u_char) ∗ cmd)) cmd++;
cmdlen ← 0;
while (cmd + cmdlen < want ∧ ¬isspace((u_char)cmd[cmdlen])) cmdlen++;
for (i ← 0; i < cmdlen; i++) {
  if (¬isalnum((u_char)cmd[i]) ∧ cmd[i] ≠ '_') return 0;
}
```
This code is used in section 791.

793. ⟨ Take a stab at argument count from here 793 ⟩ ≡
```
n ← 1;
for (cp ← cmd + cmdlen + 1; cp < want; cp++) {
  if (¬isspace((u_char)cp[−1]) ∧ isspace((u_char) ∗ cp)) n++;
}
```
This code is used in section 791.

794. ⟨ Try to find the array 794 ⟩ ≡
```
if (asprintf(&name, "complete_%.*s_%d", cmdlen, cmd, n) ≡ −1)
  internal_errorf("unable to allocate memory");
v ← global(name);
free(name);
if (∼v→flag & (ISSET | ARRAY)) {
  if (asprintf(&name, "complete_%.*s", cmdlen, cmd) ≡ −1)
    internal_errorf("unable to allocate memory");
  v ← global(name);
  free(name);
  if (∼v→flag & (ISSET | ARRAY)) return 0;
}
```
This code is used in section 791.

795. ⟨ Walk the array and build words list 795 ⟩ ≡
```
for (vp ← v; vp; vp ← vp→u.array) {
  if (∼vp→flag & ISSET) continue;
  s ← str_val(vp);
  slen ← strlen(s);
  if (slen < wantlen) continue;
  if (slen > wantlen) slen ← wantlen;
  if (slen ≠ 0 ∧ strncmp(s, want, slen) ≠ 0) continue;
  ∗words ← areallocarray(∗words, (∗nwords) + 2, sizeof ∗∗words, ATEMP);
  (∗words)[(∗nwords)++] ← str_save(s, ATEMP);
}
if (∗nwords ≠ 0) (∗words)[∗nwords] ← Λ;
```
This code is used in section 791.

796. Do file globbing:

* appends "*" to (a copy of) *str* if no globbing chararacters found
* does expansion, checks for no match, etc.
* sets $*wordsp$ to the array of matching strings
* **return**s the number of matching strings

```
⟨ Globbing 766 ⟩ +≡
  static int x_file_glob(int flags, const char *str, int slen, char ***wordsp)
  {
    char *toglob;
    char **words;
    int nwords;
    XPtrV w;
    struct Source *s, *sold;
    if (slen < 0) return 0;
    toglob ← add_glob(str, slen);
    ⟨ Glob "str"* to an array of words 797 ⟩
    for (nwords ← 0; words[nwords]; nwords++) ;
    if (nwords ≡ 1) {⟨ Check if the globbed file exists or for the empty string 798 ⟩}
    afree(toglob, ATEMP);
    if (nwords) {
      *wordsp ← words;
    }
    else if (words) {
      x_free_words(nwords, words);
      *wordsp ← Λ;
    }
    return nwords;
  }
```

797. Uses *expand* to do it which recurses into these routines (I think...).

```
⟨ Glob "str"* to an array of words 797 ⟩ ≡
  sold ← source;
  s ← pushs(SWSTR, ATEMP);
  s→start ← s→str ← toglob;
  source ← s;
  if (yylex(ONEWORD | UNESCAPE) ≠ LWORD) {
    source ← sold;
    internal_warningf("%s: substitute error", __func__);
    return 0;
  }
  source ← sold;
  XPinit(w, 32);
  expand(yylval.cp, &w, DOGLOB | DOTILDE | DOMARKDIRS);
  XPput(w, Λ);
  words ← (char **) XPclose(w);
```

This code is used in section 796.

798. An empty string results if we tried to glob something which evaluated to an empty string, eg. "`$FOO`" when there is no `FOO` etc.

⟨ Check if the globbed file exists or for the empty string 798 ⟩ ≡

```
struct stat statb;
if ((lstat(words[0], &statb) ≡ −1) ∨ words[0][0] ≡ '\0') {
  x_free_words(nwords, words);
  words ← Λ;
  nwords ← 0;
}
```

This code is used in section 796.

799. Globbing Utilities.

```
⟨ Globbing 766 ⟩ +≡
  void x_print_expansions(int nwords, char *const *words, int is_command)
  {
    int prefix_len;
    if (¬is_command ∧ (prefix_len ← x_longest_prefix(nwords, words)) > 0) {
      int i;
      if (nwords ≡ 1)      /* Special case for 1 match: prefix is the whole word */
        prefix_len ← x_basename(words[0], Λ);
      for (i ← 0; i < nwords; i++)
        if (x_basename(words[i] + prefix_len, Λ) > prefix_len)
            /* Any (non-trailing) slashes in non-common word suffixes? */
          break;
      if (i ≡ nwords) {      /* All in same directory? */
        ⟨ Strip identical directory names and return 801 ⟩
      }
    }
    x_putc('\r');
    x_putc('\n');
    pr_list(words);      /* Enumerate expansions */
  }
```

800. If all matches are in the same directory we want to omit the directory name.

801.

```
⟨ Strip identical directory names and return 801 ⟩ ≡
  XPtrV l;
  while (prefix_len > 0 ∧ words[0][prefix_len − 1] ≠ '/') prefix_len −−;
  XPinit(l, nwords + 1);
  for (i ← 0; i < nwords; i++) XPput(l, words[i] + prefix_len);
  XPput(l, Λ);
  x_putc('\r');
  x_putc('\n');
  pr_list((char **) XPptrv(l));      /* Enumerate expansions */
  XPfree(l);      /* not x_free_words */
  return;
```

This code is used in section 799.

802. Find the longest common prefix.

```
⟨ Globbing 766 ⟩ +≡
  int x_longest_prefix(int nwords, char *const *words)
  {
    int i, j;
    int prefix_len;
    char *p;
    if (nwords ≤ 0) return 0;
    prefix_len ← strlen(words[0]);
    for (i ← 1; i < nwords; i++)
      for (j ← 0, p ← words[i]; j < prefix_len; j++)
        if (p[j] ≠ words[0][j]) {
          prefix_len ← j;
          break;
        }
    return prefix_len;
  }
```

803. Locate the word in *buf* surrounding cursor position *pos*.

```
#define IS_WORDC(c) ¬(ctype(c, C_LEX1) ∨ (c) ≡ '\'' ∨ (c) ≡ '"' ∨ (c) ≡ '`' ∨ (c) ≡ '=' ∨ (c) ≡ ':')
⟨ Globbing 766 ⟩ +≡
  static int x_locate_word(const char *buf, int buflen, int pos, int *startp, int *is_commandp)
  {
    int p;
    int start, end;
    if (pos < 0 ∨ pos > buflen) {      /* Bad call? Probably should report error */
      *startp ← pos;
      *is_commandp ← 0;
      return 0;
    }
    start ← pos;
    ⟨ Locate a word's beginning in start 804 ⟩
    ⟨ Locate the end of a word in end 805 ⟩
    if (is_commandp) {⟨ Figure out if this is a command 806 ⟩}
    *startp ← start;
    return end − start;
  }
```

804. (has effect of allowing one blank after the end of a word)

```
⟨ Locate a word's beginning in start 804 ⟩ ≡
  for ( ;
        (start > 0 ∧ IS_WORDC(buf[start − 1])) ∨ (start > 1 ∧ buf[start − 2] ≡ '\');
        start−− )
    ;     /* Keep going backwards to start of word */
```

This code is used in section 803.

805. ⟨ Locate the end of a word in *end* 805 ⟩ ≡

```
  for (end ← start; end < buflen ∧ IS_WORDC(buf[end]); end++) {
    if (buf[end] ≡ '\' ∧ (end + 1) < buflen) end++;
  }
```

This code is used in section 803.

806. If command has a "/" $PATH is not searched—only the current directory which is just like file globbing.

```
⟨ Figure out if this is a command 806 ⟩ ≡
  int iscmd;
  for (p ← start − 1; p ≥ 0 ∧ isspace((unsigned char) buf[p]); p−−) ;
  iscmd ← p < 0 ∨ strchr(";|&()‘", buf[p]);
  if (iscmd) {
    for (p ← start; p < end; p++)
      if (buf[p] ≡ '/') break;
    iscmd ← p ≡ end;
  }
  ∗is_commandp ← iscmd;
```

This code is used in section 803.

807. Detect whether a string has syntactically correct globbing pattern(s) or not, or contains a syntax error:

* "[" with no closing "]".
* Imbalanced "$(...)" expression.
* "[...]" and "*(...)" improperly nested (eg. "[a$(b|]c)" or "*(a[b|c]d))").

TODO:

* if no magic: if dest given, copy to dst; else return ?
* if *magic* ∧ (*no globbing* ∨ *syntax error*) debunk to dst; else return ?
* else return ?

TODO: Tab? Why is ␣ a globber?

```
⟨ misc.c 9 ⟩ +≡
  int has_globbing(const char ∗xp, const char ∗xpe)
  {
    const unsigned char ∗p ← (const unsigned char ∗) xp;
    const unsigned char ∗pe ← (const unsigned char ∗) xpe;
    int c;
    int nest ← 0, bnest ← 0;
    int saw_glob ← 0;
    int in_bracket ← 0;      /∗ inside "[...]" ∗/
    for ( ; p < pe; p++) {
      if (¬ISMAGIC(∗p)) continue;
      if ((c ← ∗++p) ≡ '*' ∨ c ≡ '?') saw_glob ← 1;
      else if (c ≡ '[') {⟨ Detected "[" in has_globbing 808 ⟩}
      else if (c ≡ ']') {⟨ Detected "]" in has_globbing 809 ⟩}
      else if ((c & 0x80) ∧ strchr("*+?@!␣", c & 0x7f)) {⟨ Detected pattern in has_globbing 810 ⟩}
      else if (c ≡ '|') { if (in_bracket ∧ ¬bnest) return 0; }      /∗ "*(a[foo|bar])" ∗/
      else if (c ≡ ')') {
        if (in_bracket) { if (¬bnest−−) return 0;      /∗ "*(a[b)c]" ∗/ }
        else if (nest) nest−−;
      }      /∗ else must be "MAGIC-MAGIC" or "MAGIC-[!-]{,}]" ∗/
    }
    return saw_glob ∧ ¬in_bracket ∧ ¬nest;
  }
```

808. ⟨ Detected "[" in *has_globbing* 808 ⟩ ≡

```
if (¬in_bracket) {
  saw_glob ← 1;
  in_bracket ← 1;
  if (ISMAGIC(p[1]) ∧ p[2] ≡ '!') p += 2;
  if (ISMAGIC(p[1]) ∧ p[2] ≡ ']') p += 2;
}   /* XXX Do we need to check ranges here? POSIX Q */
```

This code is used in section 807.

809. ⟨ Detected "]" in *has_globbing* 809 ⟩ ≡

```
if (in_bracket) {
  if (bnest)   /* [a*(b]) */
    return 0;
  in_bracket ← 0;
}
```

This code is used in section 807.

810. ⟨ Detected pattern in *has_globbing* 810 ⟩ ≡

```
saw_glob ← 1;
if (in_bracket) bnest++;
else nest++;
```

This code is used in section 807.

811. ⟨ Globbing 766 ⟩ +≡

```
void x_free_words(int nwords, char **words)
{
  int i;
  for (i ← 0; i < nwords; i++) afree(words[i], ATEMP);
  afree(words, ATEMP);
}
```

812. History. This file contains

a) the original in-memory history mechanism

b) a more complicated mechanism done by pc@hillside.co.uk that more closely follows the real ksh way of doing things.

history, *histptr* and *histsize* are defined in lex.c.

TODO: Make these fall under a single mechanism.

⟨ history.c 812 ⟩ ≡

```
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <vis.h>
#include "sh.h"
static void history_write(void);
static FILE *history_open(void);
static void history_load(Source *);
static void history_close(void);
static int hist_execute(char *);
static int hist_replace(char **, const char *, const char *, int);
static char **hist_get(const char *, int, int);
static char **hist_get_oldest(void);
static void histbackup(void);
static FILE *histfh;
static char **histbase;      /* actual start of the history allocation */
static char **current;       /* current position in history */
static char *hname;          /* current name of history file */
static int hstarted;         /* set after hist_init called */
static int ignoredups;       /* ditch duplicated history lines? */
static int ignorespace;      /* ditch lines starting with a space? */
static Source *hist_source;
static uint32_t line_co;
static struct stat last_sb;
static volatile sig_atomic_t c_fc_depth;
static void history_lock(int);
```

See also sections 814, 815, 816, 817, 818, 819, 821, 825, 826, 827, 829, 830, 831, 832, 833, 834, 837, 838, 839, 840, 841, 842, 843, 844, and 857.

813. ⟨ Shared function declarations 4 ⟩ +≡

```
void init_histvec(void);
void hist_init(Source *);
void hist_finish(void);
void histsave(int, const char *, int);
int c_fc(char **);
void c_fc_reset(void);
void sethistcontrol(const char *);
void sethistsize(int);
void sethistfile(const char *);
char **histpos(void);
int histnum(int);
int findhist(int, int, const char *, int);
int findhistrel(const char *);
char **hist_get_newest(int);
```

814. `#define HISTORYSIZE 500 /* size of saved history */`

⟨ history.c 812 ⟩ +≡

```
void init_histvec(void)
{
  if (histbase ≡ Λ) {
    histsize ← HISTORYSIZE;
      /* * allocate one extra element so that histptr always * lies within array bounds */
    histbase ← reallocarray(Λ, histsize + 1, sizeof(char *));
    if (histbase ≡ Λ) internal_errorf("allocating␣history␣storage:␣%s", strerror(errno));
    *histbase ← Λ;
    history ← histbase + 1;
    histptr ← history − 1;
  }
}
```

815. As *hist_init* is only called when the shell is initialised and returns immediately unless FTALKING, *s* will always be that created by ⟨ Initialise to read from standard input 29 ⟩.

```
#define HMAGIC1 0xab
#define HMAGIC2 0xcd
⟨ history.c 812 ⟩ +≡
  void hist_init(Source *s)
  {
    int oldmagic1, oldmagic2;
    if (Flag(FTALKING) ≡ 0) return;
    hstarted ← 1;
    hist_source ← s;
    if (str_val(global("HISTFILE")) ≡ null) return;
    hname ← str_save(str_val(global("HISTFILE")), APERM);
    histfh ← history_open( );
    if (histfh ≡ Λ) return;
    oldmagic1 ← fgetc(histfh);
    oldmagic2 ← fgetc(histfh);
    ⟨ Close a broken or old style history file and return 820 ⟩
    history_load(s);
    history_lock(LOCK_UN);
  }
```

816.
```
⟨ history.c 812 ⟩ +≡
  static void histreset(void)
  {
    char **hp;
    for (hp ← history; hp ≤ histptr; hp++) afree(*hp, APERM);
    histptr ← history − 1;
    hist_source¬line ← 0;
  }
```

817.
```
⟨ history.c 812 ⟩ +≡
  void sethistcontrol(const char *str)
  {
    char *spec, *tok, *state;
    ignorespace ← 0;
    ignoredups ← 0;
    if (str ≡ Λ) return;
    spec ← str_save(str, ATEMP);
    for (tok ← strtok_r(spec, ":", &state); tok ≠ Λ; tok ← strtok_r(Λ, ":", &state)) {
      if (strcmp(tok, "ignoredups") ≡ 0) ignoredups ← 1;
      else if (strcmp(tok, "ignorespace") ≡ 0) ignorespace ← 1;
    }
    afree(spec, ATEMP);
  }
```

818. ⟨ history.c 812 ⟩ +≡

```
void sethistsize(int n)
{
  if (n > 0 ∧ (uint32_t) n ≠ histsize) {
    char **tmp;
    int offset ← histptr − history;      /* save most recent history */
    if (offset > n − 1) {
      char **hp;
      offset ← n − 1;
      for (hp ← history; hp < histptr − offset; hp++) afree(*hp, APERM);
      memmove(history, histptr − offset, n * sizeof(char *));
    }
    tmp ← reallocarray(histbase, n + 1, sizeof(char *));
    if (tmp ≠ Λ) {
      histbase ← tmp;
      histsize ← n;
      history ← histbase + 1;
      histptr ← history + offset;
    }
    else warningf(false, "resizing␣history␣storage:␣%s", strerror(errno));
  }
}
```

819. ⟨ history.c 812 ⟩ +≡

```
void sethistfile(const char *name)
{
  if (hstarted ≡ 0) return;      /* if not started then nothing to do */
  if (hname ∧ strcmp(hname, name) ≡ 0) return;
      /* also if the name is the same as the name we already have */
  if (hname) {      /* its a new name - possibly */
    afree(hname, APERM);
    hname ← Λ;
    histreset( );
  }
  history_close( );
  hist_init(hist_source);
}
```

820. An actual end-of-file is OK since that just means there isn't any history yet.

```
⟨ Close a broken or old style history file and return 820 ⟩ ≡
  if (oldmagic1 ≡ EOF ∨ oldmagic2 ≡ EOF) {
    if (¬feof(histfh) ∧ ferror(histfh)) {
      history_close();
      return;
    }
  }
  else if (oldmagic1 ≡ HMAGIC1 ∧ oldmagic2 ≡ HMAGIC2) {
    bi_errorf("ignoring␣old␣style␣history␣file");
    history_close();
    return;
  }
```

This code is used in section 815.

821. This "ghastly hackery" was added by Peter Collinson to permit `$HISTSIZE` to control the number of lines of history stored and to maintain a real history file.

TODO: *lno* is unused.

```
⟨ history.c 812 ⟩ +≡
  void histsave(int lno, const char *cmd, int dowrite)
  {
    char *c, *cp;
    if (ignorespace ∧ cmd[0] ≡ '␣') return;
    c ← str_save(cmd, APERM);
    if ((cp ← strrchr(c, '\n')) ≠ Λ) *cp ← '\0';
        /* XXX to properly check for duplicated lines we should first reload the histfile if needed */
    if (ignoredups ∧ histptr ≥ history ∧ strcmp(*histptr, c) ≡ 0) {
      afree(c, APERM);
      return;
    }
    if (dowrite ∧ histfh) {⟨ Reload the history file if it's changed (after locking) 823 ⟩}
    ⟨ Maintain the history file's size 822 ⟩
    *histptr ← c;
    if (dowrite ∧ histfh) {⟨ Append an entry to the history file (and unlock) 824 ⟩}
  }
```

822. Remove the oldest command if the size of the history is greater than `$HISTSIZE` or increment *histptr*.

```
⟨ Maintain the history file's size 822 ⟩ ≡
  if (histptr < history + histsize − 1) histptr++;
  else {
    afree(*history, APERM);
    memmove(history, history + 1, (histsize − 1) * sizeof (*history));
  }
```

This code is used in section 821.

823. ⟨ Reload the history file if it's changed (after locking) 823 ⟩ ≡

```
#ifndef SMALL
  struct stat sb;
  history_lock(LOCK_EX);
  if (fstat(fileno(histfh), &sb) ≠ −1) {
    if (timespeccmp(&sb.st_mtim, &last_sb.st_mtim, ≡)) ;      /* file is unchanged */
    else {
      histreset( );
      history_load(hist_source);
    }
  }
#endif
```

This code is used in section 821.

824. ⟨ Append an entry to the history file (and unlock) 824 ⟩ ≡

```
#ifndef SMALL
  char *encoded;
  if (fseeko(histfh, 0, SEEK_END) ≡ 0 ∧ stravis(&encoded, c, VIS_SAFE | VIS_NL) ≠ −1) {
    fprintf(histfh, "%s\n", encoded);
    fflush(histfh);
    fstat(fileno(histfh), &last_sb);
    line_co++;
    history_write( );
    free(encoded);
  }
  history_lock(LOCK_UN);
#endif
```

This code is used in section 821.

825. ⟨ history.c 812 ⟩ +≡

```
  static void history_lock(int operation)
  {
    while (flock(fileno(histfh), operation) ≠ 0) {
      if (errno ≡ EINTR ∨ errno ≡ EAGAIN) continue;
      else break;
    }
  }
```

826. Run a (possibly fixed) history entry.

/* Commands are executed here instead of pushing them onto the * input 'cause POSIX says the redirection and variable assignments * in * "`X=y fc -e - 42 2> /dev/null`" * are to effect the repeated commands environment. */

Which input?

```
⟨ history.c 812 ⟩ +≡
  static int hist_execute(char *cmd)
  {
    Source *sold;
    int ret;
    char *p, *q;
    histbackup( );      /* aka. ⟨ Back out the last history entry 828 ⟩ */
    for (p ← cmd; p; p ← q) {
      if ((q ← strchr(p, '\n'))) {
        *q++ ← '\0';      /* kill the ~~nl~~ */
        if (¬*q) q ← Λ;      /* ignore trailing ~~nl~~ */
      }
      histsave(++(hist_source→line), p, 1);
      shellf("%s\n", p);      /* POSIX doesn't say this is done... */
      if ((p ← q)) q[−1] ← '\n';      /* restore ~~nl~~ (trailing ~~nl~~ not restored) */
    }
    sold ← source;
    ret ← command(cmd, 0);      /* XXX: source should not get trashed by this... */
    source ← sold;
    return ret;
  }
```

827.

```
⟨ history.c 812 ⟩ +≡
  static void histbackup(void){ ⟨ Back out the last history entry 828 ⟩ }
```

828.

```
⟨ Back out the last history entry 828 ⟩ ≡
  static int last_line ← −1;
  if (histptr ≥ history ∧ last_line ≠ hist_source→line) {
    hist_source→line −−;
    afree(*histptr, APERM);
    histptr −−;
    last_line ← hist_source→line;
  }
```

This code is cited in section 826.

This code is used in section 827.

829. ⟨ history.c 812 ⟩ +≡

```
static int hist_replace(char **hp, const char *pat, const char *rep, int global)
{
  char *line;
  if (¬pat) line ← str_save(*hp, ATEMP);
  else {
    char *s, *s1;
    int pat_len ← strlen(pat);
    int rep_len ← strlen(rep);
    int len;
    XString xs;
    char *xp;
    int any_subst ← 0;
    Xinit(xs, xp, 128, ATEMP);
    for (s ← *hp; (s1 ← strstr(s, pat)) ∧ (¬any_subst ∨ global); s ← s1 + pat_len) {
      any_subst ← 1;
      len ← s1 − s;
      XcheckN(xs, xp, len + rep_len);
      memcpy(xp, s, len);      /* first part */
      xp += len;
      memcpy(xp, rep, rep_len);      /* replacement */
      xp += rep_len;
    }
    if (¬any_subst) {
      bi_errorf("substitution␣failed");
      return 1;
    }
    len ← strlen(s) + 1;
    XcheckN(xs, xp, len);
    memcpy(xp, s, len);
    xp += len;
    line ← Xclose(xs, xp);
  }
  return hist_execute(line);
}
```

830. History Navigation. Return the current history position.

```
⟨ history.c 812 ⟩ +≡
  char **histpos(void)
  {
    return current;
  }
```

831. Move to the given history position or the last if there aren't enough.

```
⟨ history.c 812 ⟩ +≡
  int histnum(int n)
  {
    int last ← histptr − history;
    if (n < 0 ∨ n ≥ last) {
      current ← histptr;
      return last;
    }
    else {
      current ← &history[n];
      return n;
    }
  }
```

832. Return a pointer to the newest command in the history.

```
⟨ history.c 812 ⟩ +≡
  char **hist_get_newest(int allow_cur)
  {
    if (histptr < history ∨ (¬allow_cur ∧ histptr ≡ history)) {
      bi_errorf("no␣history␣(yet)");
      return Λ;
    }
    if (allow_cur) return histptr;
    return histptr − 1;
  }
```

833. Return a pointer to the oldest command in the history.

```
⟨ history.c 812 ⟩ +≡
  static char **hist_get_oldest(void)
  {
    if (histptr ≤ history) {
      bi_errorf("no␣history␣(yet)");
      return Λ;
    }
    return history;
  }
```

834. Return a pointer to a history item matching the given number or string pattern.

```
⟨ history.c 812 ⟩ +≡
  static char **hist_get(const char *str, int approx, int allow_cur)
  {
    char **hp ← Λ;
    int n;
    if (getn(str, &n)) {⟨ Get the history item matching a number 835 ⟩}
    else {⟨ Get the history item matching a string 836 ⟩}
    return hp;
  }
```

835. Returns the history item n beyond or behind the *histptr*, which is the end...

```
⟨ Get the history item matching a number 835 ⟩ ≡
  hp ← histptr + (n < 0 ? n : (n − hist_source→line));
  if ((long) hp < (long) history) {
    if (approx) hp ← hist_get_oldest( );
    else {
      bi_errorf("%s: not in history", str);
      hp ← Λ;
    }
  }
  else if (hp > histptr) {
    if (approx) hp ← hist_get_newest(allow_cur);
    else {
      bi_errorf("%s: not in history", str);
      hp ← Λ;
    }
  }
  else if (¬allow_cur ∧ hp ≡ histptr) {
    bi_errorf("%s: invalid range", str);
    hp ← Λ;
  }
```

This code is used in section 834.

836. ⟨ Get the history item matching a string 836 ⟩ ≡

```
  int anchored ← *str ≡ '?' ? (++str, 0) : 1;      /* the -1 is to avoid the current fc command */
  n ← findhist(histptr − history − 1, 0, str, anchored);
  if (n < 0) {
    bi_errorf("%s: not in history", str);
    hp ← Λ;
  }
  else hp ← &history[n];
```

This code is used in section 834.

837. These will become unnecessary if *hist_get* is modified to allow searching from positions other than the end, and in either direction.

```
⟨ history.c 812 ⟩ +≡
  int findhist(int start, int fwd, const char *str, int anchored)
  {
    char **hp;
    int maxhist ← histptr − history;
    int incr ← fwd ? 1 : −1;
    int len ← strlen(str);
    if (start < 0 ∨ start ≥ maxhist) start ← maxhist;
    hp ← &history[start];
    for ( ; hp ≥ history ∧ hp ≤ histptr; hp += incr)
      if ((anchored ∧ strncmp(*hp, str, len) ≡ 0) ∨ (¬anchored ∧ strstr(*hp, str))) return hp − history;
    return −1;
  }
```

838. ⟨ history.c 812 ⟩ +≡

```
  int findhistrel(const char *str)
  {
    int maxhist ← histptr − history;
    int start ← maxhist − 1;
    int rec ← atoi(str);
    if (rec ≡ 0) return −1;
    if (rec > 0) {
      if (rec > maxhist) return −1;
      return rec − 1;
    }
    if (rec > maxhist) return −1;
    return start + rec + 1;
  }
```

839. History File I/O. The history file is accessed using C's standard I/O library and not the **Shf** object described previously.

```
⟨ history.c 812 ⟩ +≡
  static FILE *history_open(void)
  {
    FILE *f ← Λ;
#ifndef SMALL
    struct stat sb;
    int fd, fddup;
    if ((fd ← open(hname, O_RDWR | O_CREAT | O_EXLOCK, °600)) ≡ −1) return Λ;
    if (fstat(fd, &sb) ≡ −1 ∨ sb.st_uid ≠ getuid( )) { close(fd); return Λ; }
    fddup ← savefd(fd);
    if (fddup ≠ fd) close(fd);
    if ((f ← fdopen(fddup, "r+")) ≡ Λ) close(fddup);
    else last_sb ← sb;
#endif
    return f;
  }
```

840. If the saved history file is too large later commands will push older ones out by virtue of being appended to *history* by *histsave*.

```
⟨ history.c 812 ⟩ +≡
  static void history_load(Source *s)
  {
    char *p, encoded[LINE + 1], line[LINE + 1];
    int toolongseen ← 0;
    rewind(histfh);
    line_co ← 1;      /* just read it all; will auto resize history upon next command */
    while (fgets(encoded, sizeof (encoded), histfh)) {
      if ((p ← strchr(encoded, '\n')) ≡ Λ) {      /* discard overlong line */
        do {      /* maybe a missing trailing newline? */
          if (strlen(encoded) ≠ sizeof (encoded) − 1) {
            bi_errorf("history␣file␣is␣corrupt");
            return;
          }
        } while (fgets(encoded, sizeof (encoded), histfh) ∧ strchr(encoded, '\n') ≡ Λ);
        if (¬toolongseen) {
          toolongseen ← 1;
          bi_errorf("ignored␣history␣line(s)␣longer␣than""␣%d␣bytes", LINE);
        }
        continue;
      }
      *p ← '\0';
      s→line ← line_co;
      s→cmd_offset ← line_co;
      strunvis(line, encoded);
      histsave(line_co, line, 0);
      line_co++;
    }
    history_write( );
  }
```

841. ⟨ history.c 812 ⟩ +≡

```
static void history_close(void)
{
  if (histfh) {
    fflush(histfh);
    fclose(histfh);
    histfh ← Λ;
  }
}
```

842. ⟨ history.c 812 ⟩ +≡

```
void hist_finish(void)
{
  history_close( );
}
```

843. ⟨ history.c 812 ⟩ +≡

```
static void history_write(void)
{
  char **hp, *encoded;
  if (line_co < histsize + (histsize/4)) return;      /* if file has grown over 25% */
  rewind(histfh);      /* rewrite the whole caboodle */
  if (ftruncate(fileno(histfh), 0) ≡ -1) {
    bi_errorf("failed to rewrite history file - %s", strerror(errno));
  }
  for (hp ← history; hp ≤ histptr; hp++) {
    if (stravis(&encoded, *hp, VIS_SAFE | VIS_NL) ≠ -1) {
      if (fprintf(histfh, "%s\n", encoded) ≡ -1) {
        free(encoded);
        return;
      }
      free(encoded);
    }
  }
  line_co ← histsize;
  fflush(histfh);
  fstat(fileno(histfh), &last_sb);
}
```

844. `fc/history` **Built-in Command.** Actually many similar sub-commands in one. Stands for Fix Command.

```
⟨ history.c 812 ⟩ +≡
  int c_fc(char **wp)
  {
    struct Shf *shf;
    struct Temp *tf ← Λ;
    char *p, *editor ← Λ;
    int gflag ← 0, lflag ← 0, nflag ← 0, sflag ← 0, rflag ← 0;
    int optc, ret;
    char *first ← Λ, *last ← Λ;
    char **hfirst, **hlast, **hp;
    if (c_fc_depth ≠ 0) {
      bi_errorf("history␣function␣called␣recursively");
      return 1;
    }
    if (¬Flag(FTALKING_I)) {
      bi_errorf("history␣functions␣not␣available");
      return 1;
    }
    while ((optc ← ksh_getopt(wp, &builtin_opt, "e:glnrs0,1,2,3,4,5,6,7,8,9,")) ≠ −1)
      ⟨ Process an fc option 845 ⟩
    wp += builtin_opt.optind;
    if (sflag) {⟨ Substitute and execute command 846 ⟩}
    if (editor ∧ (lflag ∨ nflag)) {
      bi_errorf("can't␣use␣-l,␣-n␣with␣-e");
      return 1;
    }
    if (¬first ∧ (first ← *wp)) wp++;
    if (¬last ∧ (last ← *wp)) wp++;
    if (*wp) {
      bi_errorf("too␣many␣arguments");
      return 1;
    }
    if (¬first) {⟨ Calculate fc's default hfirst & hlast 850 ⟩}
    else {⟨ Validate fc's supplied hfirst & hlast 851 ⟩}
    if (hfirst > hlast) {⟨ Make the hlast hfirst and the hfirst hlast 852 ⟩}
    if (lflag) {⟨ List history 853 ⟩}
    ⟨ Save selected history lines to a temporary file 854 ⟩
    setstr(local("_", false), tf→name, KSH_RETURN_ERROR);      /* arbitrarily ignore setstr errors here */
    {⟨ Edit the temporarily-saved history 855 ⟩}
    {⟨ Run the fixed history commands 856 ⟩}
  }
```

845. ⟨ Process an `fc` option 845 ⟩ ≡

```
switch (optc) {
case 'e': p ← builtin_opt.optarg;
  if (strcmp(p, "-") ≡ 0) sflag++;
  else {
    size_t len ← strlen(p) + 4;
    editor ← str_nsave(p, len, ATEMP);
    strlcat(editor, "␣$_", len);
  }
  break;
case 'g': gflag++; break;      /* non-AT&T ksh */
case 'l': lflag++; break;
case 'n': nflag++; break;
case 'r': rflag++; break;
case 's': sflag++; break;      /* POSIX version of "-e -" */
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
    /* kludge city—accept "-num" as "-- -num" (kind of) */
  p ← shf_smprintf("-%c%s", optc, builtin_opt.optarg);
  if (¬first) first ← p;
  else if (¬last) last ← p;
  else {
    bi_errorf("too␣many␣arguments");
    return 1;
  }
  break;
case '?': return 1;
}
```

This code is used in section 844.

846. ⟨ Substitute and execute command 846 ⟩ ≡

```
char *pat ← Λ, *rep ← Λ;
if (editor ∨ lflag ∨ nflag ∨ rflag) {
  bi_errorf("can't␣use␣-e,␣-l,␣-n,␣-r␣with␣-s␣(-e␣-)");
  return 1;
}
if (*wp ∧ **wp ∧ (p ← strchr(*wp + 1, '='))) {⟨ Check for pattern replacement argument 847 ⟩}
⟨ Check for search prefix 848 ⟩
⟨ Search for an historic command 849 ⟩
```

This code is used in section 844.

847. ⟨ Check for pattern replacement argument 847 ⟩ ≡

```
pat ← str_save(*wp, ATEMP);
p ← pat + (p − *wp);
*p++ ← '\0';
rep ← p;
wp++;
```

This code is used in section 846.

848. ⟨ Check for search prefix 848 ⟩ ≡
```
if (¬first ∧ (first ← ∗wp)) wp++;
if (last ∨ ∗wp) {
  bi_errorf("too␣many␣arguments");
  return 1;
}
```
This code is used in section 846.

849. ⟨ Search for an historic command 849 ⟩ ≡
```
hp ← first ? hist_get(first, false, false) : hist_get_newest(false);
if (¬hp) return 1;
c_fc_depth++;
ret ← hist_replace(hp, pat, rep, gflag);
c_fc_reset();
return ret;
```
This code is used in section 846.

850. ⟨ Calculate fc's default *hfirst* & *hlast* 850 ⟩ ≡
```
hfirst ← lflag ? hist_get("-16", true, true) : hist_get_newest(false);
if (¬hfirst) return 1;      /∗ can't fail if hfirst didn't fail ∗/
hlast ← hist_get_newest(false);
```
This code is used in section 844.

851. ⟨ Validate fc's supplied *hfirst* & *hlast* 851 ⟩ ≡ /∗ POSIX says not an error if first/last out of bounds when range is specified; AT&T ksh and pdksh allow out of bounds for -l as well. ∗/
```
hfirst ← hist_get(first, (lflag ∨ last) ? true : false, lflag ? true : false);
if (¬hfirst) return 1;
hlast ← last ? hist_get(last, true, lflag ? true : false) : (lflag ? hist_get_newest(false) : hfirst);
if (¬hlast) return 1;
```
This code is used in section 844.

852. ⟨ Make the *hlast* *hfirst* and the *hfirst* *hlast* 852 ⟩ ≡
```
char ∗∗temp;
temp ← hfirst;
hfirst ← hlast;
hlast ← temp;
rflag ← ¬rflag;      /∗ POSIX ∗/
```
This code is used in section 844.

853. ⟨ List history 853 ⟩ ≡
```
char ∗s, ∗t;
const char ∗nfmt ← nflag ? "\t" : "%d\t";
for (hp ← rflag ? hlast : hfirst; hp ≥ hfirst ∧ hp ≤ hlast; hp += rflag ? −1 : 1) {
  shf_fprintf(shl_stdout, nfmt, hist_source→line − (int)(histptr − hp));
  for (s ← ∗hp; (t ← strchr(s, '\n')); s ← t)      /∗ print multi-line commands correctly ∗/
    shf_fprintf(shl_stdout, "%.*s\t", ++t − s, s);
  shf_fprintf(shl_stdout, "%s\n", s);
}
shf_flush(shl_stdout);
return 0;
```
This code is used in section 844.

854. XXX: source should not get trashed by this.

```
⟨ Save selected history lines to a temporary file 854 ⟩ ≡
  tf ← maketemp(ATEMP, TT_HIST_EDIT, &genv→temps);
  if (¬(shf ← tf→shf)) {
    bi_errorf("cannot␣create␣temp␣file␣%s␣-␣%s", tf→name, strerror(errno));
    return 1;
  }
  for (hp ← rflag ? hlast : hfirst; hp ≥ hfirst ∧ hp ≤ hlast; hp += rflag ? −1 : 1)
    shf_fprintf(shf, "%s\n", ∗hp);
  if (shf_close(shf) ≡ EOF) {
    bi_errorf("error␣writing␣temporary␣file␣-␣%s", strerror(errno));
    return 1;
  }
```

This code is used in section 844.

855.

```
⟨ Edit the temporarily-saved history 855 ⟩ ≡
  Source ∗sold ← source;
  ret ← command(editor ? editor : "${FCEDIT:-/bin/ed}␣$_", 0);
  source ← sold;
  if (ret) return ret;
```

This code is used in section 844.

856.

```
⟨ Run the fixed history commands 856 ⟩ ≡
  struct stat statb;
  XString xs;
  char ∗xp;
  int n;
  if (¬(shf ← shf_open(tf→name, O_RDONLY, 0, 0))) {
    bi_errorf("cannot␣open␣temp␣file␣%s", tf→name);
    return 1;
  }
  n ← fstat(shf→fd, &statb) ≡ −1 ? 128 : statb.st_size + 1;
  Xinit(xs, xp, n, hist_source→areap);
  while ((n ← shf_read(xp, Xnleft(xs, xp), shf)) > 0) {
    xp += n;
    if (Xnleft(xs, xp) ≤ 0) XcheckN(xs, xp, Xlength(xs, xp));
  }
  if (n < 0) {
    bi_errorf("error␣reading␣temp␣file␣%s␣-␣%s", tf→name, strerror(shf→errno_));
    shf_close(shf);
    return 1;
  }
  shf_close(shf);
  ∗xp ← '\0';
  strip_nuls(Xstring(xs, xp), Xlength(xs, xp));
  c_fc_depth++;
  ret ← hist_execute(Xstring(xs, xp));
  c_fc_reset();
  return ret;
```

This code is used in section 844.

857. Reset the *c_fc* recursion depth counter, for when an `fc` call is interrupted.

```
⟨ history.c 812 ⟩ +≡
  void c_fc_reset(void)
  {
    c_fc_depth ← 0;
  }
```

858. Interactive Editor.

859. tty driver characters we are interested in.

```
⟨ Type definitions 17 ⟩ +≡
  typedef struct {
    int erase;
    int kill;
    int werase;
    int intr;
    int quit;
    int eof;
  } X_chars;
```

860. The width of the terminal in *x_cols* is updated by setting `$COLUMNS` or by *check_sigwinch*. The minimum value for *x_cols* includes 2 for the prompt and 3 for "␣<␣" at the end of the line.

If the prompt leaves less than `MIN_EDIT_SPACE` clear on the line, the prompt is truncated.

```
#define MIN_EDIT_SPACE 7
#define MIN_COLS (2 + MIN_EDIT_SPACE + 3)
⟨ Global variables 5 ⟩ +≡
  int x_cols ← 80;    /* width of tty and the value of $COLUMNS */
```

861.
```
⟨ Externally-linked variables 6 ⟩ +≡
  extern int x_cols;
```

862.
```
⟨ edit.c 862 ⟩ ≡
#include "config.h"
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <ctype.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
#include "edit.h"
#include "tty.h"
  X_chars edchars;
  static void x_sigwinch(int);
  volatile sig_atomic_t got_sigwinch;
  static void check_sigwinch(void);
  static int x_file_glob(int, const char *, int, char ***);
  static int x_command_glob(int, const char *, int, char ***);
  static int x_locate_word(const char *, int, int, int *, int *);
```

See also sections 866, 867, 868, 869, 871, 872, 873, 874, 875, 876, 877, 878, and 881.

863. #define BEL 0x07

```
⟨ edit.h 863 ⟩ ≡
  int x_getc(void);
  void x_flush(void);
  int x_putc(int);
  void x_puts(const char *);
  bool x_mode(bool);
  int promptlen(const char *, const char **);
  int x_do_comment(char *, int, int *);
  void x_print_expansions(int, char *const *, int);
  int x_cf_glob(int, const char *, int, int, int *, int *, char ***, int *);
  int x_longest_prefix(int, char *const *);
  int x_basename(const char *, const char *);
  void x_free_words(int, char **);
  int x_escape(const char *, size_t, int(*)(const char *, size_t));
  int x_emacs(char *, size_t);      /* emacs.c */
  void x_init_emacs(void);      /* emacs.c */
  void x_emacs_keys(X_chars *);      /* emacs.c */
  int x_vi(char *, size_t);      /* vi.c */
```

864. ⟨ Shared function declarations 4 ⟩ +≡

```
  void x_init(void);
  int x_read(char *, size_t);
  void set_editmode(const char *);
```

865. ⟨ Externally-linked variables 6 ⟩ +≡

```
  extern X_chars edchars;
```

866. The globbing implementation is included in edit.c.

```
⟨ edit.c 862 ⟩ +≡
  ⟨ Globbing 766 ⟩
```

867. This is where *main* initialises interactive editing. The tty driver character are set to -2 "to force initial binding" except that nothing does that. Also *werase* is set to °*27* or ^W "for deficient systems".

```
⟨ edit.c 862 ⟩ +≡
  void x_init(void)
  {
    edchars.erase ← edchars.kill ← edchars.intr ← edchars.quit ← edchars.eof ← −2;
    edchars.werase ← °27;
    if (setsig(&sigtraps[SIGWINCH], x_sigwinch, SS_RESTORE_ORIG | SS_SHTRAP))
      sigtraps[SIGWINCH].flags |= TF_SHELL_USES;
    got_sigwinch ← 1;      /* force initial check */
    check_sigwinch( );
#ifdef EMACS
    x_init_emacs( );
#endif      /* EMACS */
  }
```

868. Choose an interactive editing mode. Only sets flags—doesn't active the mode.

```
⟨ edit.c 862 ⟩ +≡
  void set_editmode(const char *ed)
  {
    static const enum sh_flag edit_flags[] ← {
#ifdef EMACS
    FEMACS, FGMACS,
#endif
#ifdef VI
    FVI,
#endif
    };
    char *rcp;
    unsigned int ele;
    if ((rcp ← strrchr(ed, '/'))) ed ← ++rcp;
    for (ele ← 0; ele < NELEM(edit_flags); ele++)
      if (strstr(ed, sh_options[(int) edit_flags[ele]].name)) {
        change_flag(edit_flags[ele], OF_SPECIAL, 1);
        return;
      }
  }
```

869. The *edchars* object is next (and only) touched by *x_mode*, called when enabling or disabling interactivity for a line or character.

```
⟨ edit.c 862 ⟩ +≡
  bool x_mode(bool onoff)
  {
    static bool x_cur_mode;
    bool prev;
    if (x_cur_mode ≡ onoff) return x_cur_mode;
    prev ← x_cur_mode;
    x_cur_mode ← onoff;
    if (onoff) {
      struct termios cb;
      X_chars oldchars;
      oldchars ← edchars;
      cb ← tty_state;
      edchars.erase ← cb.c_cc[VERASE];
      edchars.kill ← cb.c_cc[VKILL];
      edchars.intr ← cb.c_cc[VINTR];
      edchars.quit ← cb.c_cc[VQUIT];
      edchars.eof ← cb.c_cc[VEOF];
      edchars.werase ← cb.c_cc[VWERASE];
      cb.c_iflag &= ∼(INLCR | ICRNL);
      cb.c_lflag &= ∼(ISIG | ICANON | ECHO);
      cb.c_cc[VLNEXT] ← _POSIX_VDISABLE;      /* osf/1 processes lnext when ¬icanon */
      cb.c_cc[VDISCARD] ← _POSIX_VDISABLE;
        /* sunos 4.1.x & osf/1 processes discard(flush) when ¬icanon */
      cb.c_cc[VTIME] ← 0;
      cb.c_cc[VMIN] ← 1;
      tcsetattr(tty_fd, TCSADRAIN, &cb);
      ⟨ Convert unset tty characters to internal 'unset' value 870 ⟩
    }
    else {
      tcsetattr(tty_fd, TCSADRAIN, &tty_state);
    }
    return prev;
  }
```

870. ⟨ Convert unset tty characters to internal 'unset' value 870 ⟩ ≡

```
  if (edchars.erase ≡ _POSIX_VDISABLE) edchars.erase ← −1;
  if (edchars.kill ≡ _POSIX_VDISABLE) edchars.kill ← −1;
  if (edchars.intr ≡ _POSIX_VDISABLE) edchars.intr ← −1;
  if (edchars.quit ≡ _POSIX_VDISABLE) edchars.quit ← −1;
  if (edchars.eof ≡ _POSIX_VDISABLE) edchars.eof ← −1;
  if (edchars.werase ≡ _POSIX_VDISABLE) edchars.werase ← −1;
  if (memcmp(&edchars, &oldchars, sizeof (edchars)) ≠ 0) {
#ifdef EMACS
    x_emacs_keys(&edchars);
#endif
  }
```

This code is used in section 869.

871. When a terminal window is resized ksh will receive the SIGWINCH signal (if it's in the foreground process group) but the necessary work cannot be carried out in the signal handler.

```
⟨ edit.c 862 ⟩ +≡
  static void x_sigwinch(int sig)
  {
    got_sigwinch ← 1;
  }
```

872. Do not export $COLUMNS/$LINES. Many applications check these before checking *ws_col/row* in *winsize*, so if the application is started with $COLUMNS or $LINES in the environment and the window is then resized, it won't see the change because the environment doesn't change.

```
⟨ edit.c 862 ⟩ +≡
  static void check_sigwinch(void)
  {
    if (got_sigwinch) {
      struct winsize ws;
      got_sigwinch ← 0;
      if (procpid ≡ kshpid ∧ ioctl(tty_fd, TIOCGWINSZ, &ws) ≡ 0) {
        struct tbl *vp;
        if (ws.ws_col) {
          x_cols ← ws.ws_col < MIN_COLS ? MIN_COLS : ws.ws_col;
          if ((vp ← typeset("COLUMNS", 0, 0, 0, 0))) setint(vp, (int64_t) ws.ws_col);
        }
        if (ws.ws_row ∧ (vp ← typeset("LINES", 0, 0, 0, 0))) setint(vp, (int64_t) ws.ws_row);
      }
    }
  }
```

873. Read an edited command line.

```
⟨ edit.c 862 ⟩ +≡
  int x_read(char *buf, size_t len)
  {
    int i;
    x_mode(true);
#ifdef EMACS
    if (Flag(FEMACS) ∨ Flag(FGMACS)) i ← x_emacs(buf, len);
    else
#endif
#ifdef VI
      if (Flag(FVI)) i ← x_vi(buf, len);
      else
#endif
        i ← −1;      /* internal error */
    x_mode(false);
    check_sigwinch( );
    return i;
  }
```

874. Block and read a single character turning edit mode off to handle and traps.

```
⟨ edit.c 862 ⟩ +≡
  int x_getc(void)
  {
    char c;
    int n;
    while ((n ← blocking_read(STDIN_FILENO, &c, 1)) < 0 ∧ errno ≡ EINTR)
      if (trap) {
        x_mode(false);
        runtraps(0);
        x_mode(true);
      }
    if (n ≠ 1) return −1;
    return (int)(unsigned char) c;
  }
```

875.

```
⟨ edit.c 862 ⟩ +≡
  int x_putc(int c)
  {
    return shf_putc(c, shl_out);
  }
```

876.

```
⟨ edit.c 862 ⟩ +≡
  void x_puts(const char *s)
  {
    while (*s ≠ 0) shf_putc(*s++, shl_out);
  }
```

877.

```
⟨ edit.c 862 ⟩ +≡
  void x_flush(void)
  {
    shf_flush(shl_out);
  }
```

878. Shared Routines.

Handle the commenting/uncommenting of a line.

Returns:

* 1 if a carriage return is indicated (comment added)
* 0 if no return (comment removed)
* -1 if there is an error (not enough room for comment chars)

If successful, $*lenp$ contains the new length. Note: cursor should be moved to the start of the line after (un)commenting.

```
⟨ edit.c 862 ⟩ +≡
  int x_do_comment(char *buf, int bsize, int *lenp)
  {
    int i, j;
    int len ← *lenp;
    if (len ≡ 0) return 1;      /* somewhat arbitrary—it's what AT&T ksh does */
    if (buf[0] ≡ '#') {⟨ Uncomment the input and return 0 879 ⟩}
    else {⟨ Try to comment out the input and return 880 ⟩}
  }
```

879. ⟨ Uncomment the input and **return** 0 879 ⟩ ≡

```
  int saw_nl ← 0;
  for (j ← 0, i ← 1; i < len; i++) {
    if (¬saw_nl ∨ buf[i] ≠ '#') buf[j++] ← buf[i];
    saw_nl ← buf[i] ≡ '\n';
  }
  *lenp ← j;
  return 0;
```

This code is used in section 878.

880. ⟨ Try to comment out the input and **return** 880 ⟩ ≡

```
  int n ← 1;
  for (i ← 0; i < len; i++)      /* See if there's room for the #s—1 per ~~nl~~ */
    if (buf[i] ≡ '\n') n++;
  if (len + n ≥ bsize) return −1;
  for (i ← len, j ← len + n; −−i ≥ 0; ) {      /* Now insert them */
    if (buf[i] ≡ '\n') buf[−−j] ← '#';
    buf[−−j] ← buf[i];
  }
  buf[0] ← '#';
  *lenp += n;
  return 1;
```

This code is used in section 878.

881. If the argument string contains any special characters they will be escaped and the result will be put into the edit buffer by a keybinding-specific function.

```
⟨ edit.c 862 ⟩ +≡
  int x_escape(const char *s, size_t len, int(*putbuf_func)(const char *, size_t))
  {
    size_t add, wlen;
    const char *ifs ← str_val(local("IFS", 0));
    int rval ← 0;
    for (add ← 0, wlen ← len; wlen − add > 0; add++) {
      if (strchr("!\"#$&'()*:;<=>?[\\]`{|}", s[add]) ∨ strchr(ifs, s[add])) {
        if (putbuf_func(s, add) ≠ 0) {
          rval ← −1;
          break;
        }
        putbuf_func("\\", 1);
        putbuf_func(&s[add], 1);
        add++;
        wlen −= add;
        s += add;
        add ← −1;      /* after the increment it will go to 0 */
      }
    }
    if (wlen > 0 ∧ rval ≡ 0) rval ← putbuf_func(s, wlen);
    return (rval);
  }
```

882. Emacs.

```
⟨ emacs.c 882 ⟩ ≡
#include "config.h"
#ifdef EMACS
#include <sys/queue.h>
#include <sys/stat.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifndef SMALL
#include <term.h>
#include <curses.h>
#endif    /* SMALL */
#include "sh.h"
#include "edit.h"
#undef CTRL
#define CTRL(x)  ((x) ≡ '?' ? 0x7F : (x) & 0x1F)    /* ASCII */
#define UNCTRL(x)  ((x) ≡ 0x7F ? '?' : (x) | 0x40)    /* ASCII */
  ⟨ Emacs mode static functions 884 ⟩
  ⟨ Emacs mode static variables 886 ⟩
  ⟨ Emacs mode source 890 ⟩
#endif    /* EMACS */
```

883. ⟨ Shared function declarations 4 ⟩ +≡

```
int x_bind(const char *, const char *, int, int);
```

884. Loadsa functions.

```
⟨ Emacs mode static functions 884 ⟩ ≡
  static int x_ins(char *);
  static void x_delete(int, int);
  static int x_bword(void);
  static int x_fword(void);
  static void x_goto(char *);
  static void x_bs(int);
  static int x_size_str(char *);
  static int x_size(int);
  static void x_zots(char *);
  static void x_zotc(int);
  static void x_load_hist(char **);
  static int x_search(char *, int, int);
  static int x_match(char *, char *);
  static void x_redraw(int);
  static void x_push(int);
  static void x_adjust(void);
  static void x_e_ungetc(int);
  static int x_e_getc(void);
  static int x_e_getu8(char *, int);
  static void x_e_putc(int);
  static void x_e_puts(const char *);
  static int x_comment(int);
  static int x_fold_case(int);
  static char *x_lastcp(void);
  static void do_complete(int, Comp_type);
  static int isu8cont(unsigned char);
```

See also section 885.

This code is used in section 882.

885. Prototypes for keybindings.

⟨ Emacs mode static functions 884 ⟩ +≡

```
static int x_abort(int);
static int x_beg_hist(int);
static int x_clear_screen(int);
static int x_comp_comm(int);
static int x_comp_file(int);
static int x_complete(int);
static int x_del_back(int);
static int x_del_bword(int);
static int x_del_char(int);
static int x_del_fword(int);
static int x_del_line(int);
static int x_draw_line(int);
static int x_end_hist(int);
static int x_end_of_text(int);
static int x_enumerate(int);
static int x_eot_del(int);
static int x_error(int);
static int x_goto_hist(int);
static int x_ins_string(int);
static int x_insert(int);
static int x_kill(int);
static int x_kill_region(int);
static int x_list_comm(int);
static int x_list_file(int);
static int x_literal(int);
static int x_meta_yank(int);
static int x_mv_back(int);
static int x_mv_begin(int);
static int x_mv_bword(int);
static int x_mv_end(int);
static int x_mv_forw(int);
static int x_mv_fword(int);
static int x_newline(int);
static int x_next_com(int);
static int x_nl_next_com(int);
static int x_noop(int);
static int x_prev_com(int);
static int x_prev_histword(int);
static int x_search_char_forw(int);
static int x_search_char_back(int);
static int x_search_hist(int);
static int x_set_mark(int);
static int x_transpose(int);
static int x_xchg_point_mark(int);
static int x_yank(int);
static int x_comp_list(int);
static int x_expand(int);
static int x_fold_capitalize(int);
static int x_fold_lower(int);
static int x_fold_upper(int);
```

```
  static int x_set_arg(int);
  static int x_comment(int);
#ifdef DEBUG
  static int x_debug_info(int);
#endif
```

886. Horizontal scrolling stuff.

```
⟨ Emacs mode static variables 886 ⟩ ≡
  static char *xbuf;      /* beg input buffer */
  static char *xend;      /* end input buffer */
  static char *xcp;       /* current position */
  static char *xep;       /* current end */
  static char *xbp;       /* start of visible portion of input buffer */
  static char *xlp;       /* last byte visible on screen */
  static int x_adj_ok;
  static int x_adj_done;
    /* so that functions can tell whether x_adjust has been called while they are active */
  static int xx_cols;     /* cached copy of x_cols—tty width */
  static int x_col;
  static int x_displen;
  static int x_arg;       /* general purpose arg */
  static int x_arg_defaulted;     /* x_arg not explicitly set; defaulted to 1 */
```

See also sections 887, 889, 898, and 1008.

This code is used in section 882.

887. Other stuff...

```
#define KILLSIZE 20
⟨ Emacs mode static variables 886 ⟩ +≡
  static int xlp_valid;
  static int x_tty;       /* are we on a tty? */
  static int x_bind_quiet;      /* not an error if key is bound—x_emacs_keys during x_mode(true) */
  static int (*x_last_command)(int);
  static char **x_histp;      /* history position */
  static int x_nextcmd;       /* for newline-and-next */
  static char *xmp;       /* mark pointer */
  static char *killstack[KILLSIZE];
  static int killsp, killtp;
  static int x_literal_set;
  static int x_arg_set;
  static char *macro_args;
  static int prompt_skip;
  static int prompt_redraw;
```

888. Emacs functions.

```
#define XF_ARG 1 /* command takes number prefix */
#define XF_NOBIND 2 /* not allowed to bind to function */
#define XF_PREFIX 4 /* function sets prefix */
```

⟨ Type definitions 17 ⟩ +≡

```
typedef int (*kb_func)(int);
struct x_ftab {
  kb_func xf_func;
  const char *xf_name;
  short xf_flags;
};
```

889. Too early for this? Too big; how to break it up?

⟨ Emacs mode static variables 886 ⟩ +≡

```
static const struct x_ftab x_ftab[] ← {
    {x_abort, "abort", 0},
    {x_beg_hist, "beginning-of-history", 0},
    {x_clear_screen, "clear-screen", 0},
    {x_comp_comm, "complete-command", 0},
    {x_comp_file, "complete-file", 0},
    {x_complete, "complete", 0},
    {x_del_back, "delete-char-backward", XF_ARG},
    {x_del_bword, "delete-word-backward", XF_ARG},
    {x_del_char, "delete-char-forward", XF_ARG},
    {x_del_fword, "delete-word-forward", XF_ARG},
    {x_del_line, "kill-line", 0},
    {x_draw_line, "redraw", 0},
    {x_end_hist, "end-of-history", 0},
    {x_end_of_text, "eot", 0},
    {x_enumerate, "list", 0},
    {x_eot_del, "eot-or-delete", XF_ARG},
    {x_error, "error", 0},
    {x_goto_hist, "goto-history", XF_ARG},
    {x_ins_string, "macro-string", XF_NOBIND},
    {x_insert, "auto-insert", XF_ARG},
    {x_kill, "kill-to-eol", XF_ARG},
    {x_kill_region, "kill-region", 0},
    {x_list_comm, "list-command", 0},
    {x_list_file, "list-file", 0},
    {x_literal, "quote", 0},
    {x_meta_yank, "yank-pop", 0},
    {x_mv_back, "backward-char", XF_ARG},
    {x_mv_begin, "beginning-of-line", 0},
    {x_mv_bword, "backward-word", XF_ARG},
    {x_mv_end, "end-of-line", 0},
    {x_mv_forw, "forward-char", XF_ARG},
    {x_mv_fword, "forward-word", XF_ARG},
    {x_newline, "newline", 0},
    {x_next_com, "down-history", XF_ARG},
    {x_nl_next_com, "newline-and-next", 0},
    {x_noop, "no-op", 0},
    {x_prev_com, "up-history", XF_ARG},
    {x_prev_histword, "prev-hist-word", XF_ARG},
    {x_search_char_forw, "search-character-forward", XF_ARG},
    {x_search_char_back, "search-character-backward", XF_ARG},
    {x_search_hist, "search-history", 0},
    {x_set_mark, "set-mark-command", 0},
    {x_transpose, "transpose-chars", 0},
    {x_xchg_point_mark, "exchange-point-and-mark", 0},
    {x_yank, "yank", 0},
    {x_comp_list, "complete-list", 0},
    {x_expand, "expand-file", 0},
    {x_fold_capitalize, "capitalize-word", XF_ARG},
    {x_fold_lower, "downcase-word", XF_ARG},
```

```
    {x_fold_upper, "upcase-word", XF_ARG},
    {x_set_arg, "set-arg", XF_NOBIND},
    {x_comment, "comment", 0},
    {0, 0, 0},
#ifdef DEBUG
  {x_debug_info, "debug-info", 0},
#else
  {0, 0, 0},
#endif
  {0, 0, 0},
  };
```

890. Emacs editing mode requires an **Area** for the kill ring and macro definitions.

```
#define AEDIT &aedit
⟨ Emacs mode source 890 ⟩ ≡
  static Area aedit;
```

See also sections 897, 899, 900, 901, 902, 903, 905, 906, 907, 914, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 955, 956, 957, 958, 961, 962, 963, 964, 965, 966, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 986, 987, 988, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1006, 1007, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, and 1020.

This code is used in section 882.

891. First lot in man page order.

⟨ Emacs mode keybindings 891 ⟩ ≡

```
kb_add(x_abort, CTRL('G'), 0);
kb_add(x_mv_back, CTRL('B'), 0);
kb_add(x_mv_back, CTRL('X'), CTRL('D'), 0);
kb_add(x_mv_bword, CTRL('['), 'b', 0);
kb_add(x_beg_hist, CTRL('['), '<', 0);
kb_add(x_mv_begin, CTRL('A'), 0);
kb_add(x_fold_capitalize, CTRL('['), 'C', 0);
kb_add(x_fold_capitalize, CTRL('['), 'c', 0);
kb_add(x_comment, CTRL('['), '#', 0);
kb_add(x_complete, CTRL('['), CTRL('['), 0);
kb_add(x_comp_comm, CTRL('X'), CTRL('['), 0);
kb_add(x_comp_file, CTRL('['), CTRL('X'), 0);
kb_add(x_comp_list, CTRL('I'), 0);
kb_add(x_comp_list, CTRL('['), '=', 0);
kb_add(x_del_back, CTRL('?'), 0);
kb_add(x_del_back, CTRL('H'), 0);
kb_add(x_del_char, CTRL('['), '[', '3', '~', 0);      /* delete */
kb_add(x_del_bword, CTRL('W'), 0);
kb_add(x_del_bword, CTRL('['), CTRL('?'), 0);
kb_add(x_del_bword, CTRL('['), CTRL('H'), 0);
kb_add(x_del_bword, CTRL('['), 'h', 0);
kb_add(x_del_fword, CTRL('['), 'd', 0);
kb_add(x_next_com, CTRL('N'), 0);
kb_add(x_next_com, CTRL('X'), 'B', 0);
kb_add(x_fold_lower, CTRL('['), 'L', 0);
kb_add(x_fold_lower, CTRL('['), 'l', 0);
kb_add(x_end_hist, CTRL('['), '>', 0);
kb_add(x_mv_end, CTRL('E'), 0);
```

See also sections 892, 893, 894, 895, and 896.

This code is used in section 1016.

892. The rest of the (original? -ed) man page.

⟨ Emacs mode keybindings 891 ⟩ +≡

```
/* How to handle: eot: ^_, underneath copied from original keybindings: */
kb_add(x_end_of_text, CTRL('_'), 0);
kb_add(x_eot_del, CTRL('D'), 0);
kb_add(x_xchg_point_mark, CTRL('X'), CTRL('X'), 0);      /* error */
kb_add(x_expand, CTRL('['), '*', 0);
kb_add(x_mv_forw, CTRL('F'), 0);
kb_add(x_mv_forw, CTRL('X'), 'C', 0);
kb_add(x_mv_fword, CTRL('['), 'f', 0);
kb_add(x_goto_hist, CTRL('['), 'g', 0);
kb_add(x_kill, CTRL('K'), 0);      /* kill-line */
kb_add(x_enumerate, CTRL('['), '?', 0);
kb_add(x_list_comm, CTRL('X'), '?', 0);
kb_add(x_list_file, CTRL('X'), CTRL('Y'), 0);
kb_add(x_newline, CTRL('J'), 0);
kb_add(x_newline, CTRL('M'), 0);
kb_add(x_nl_next_com, CTRL('O'), 0);
kb_add(x_prev_histword, CTRL('['), '.', 0);      /* no-op */
kb_add(x_prev_histword, CTRL('['), '_', 0);
kb_add(x_literal, CTRL('^'), 0);      /* how to handle: quote: ^^ */
kb_add(x_clear_screen, CTRL('L'), 0);
kb_add(x_search_char_back, CTRL('['), CTRL(']'), 0);
kb_add(x_search_char_forw, CTRL(']'), 0);
kb_add(x_search_hist, CTRL('R'), 0);
kb_add(x_set_mark, CTRL('['), '␣', 0);
kb_add(x_transpose, CTRL('T'), 0);
kb_add(x_prev_com, CTRL('P'), 0);
kb_add(x_prev_com, CTRL('X'), 'A', 0);
kb_add(x_fold_upper, CTRL('['), 'U', 0);
kb_add(x_fold_upper, CTRL('['), 'u', 0);
kb_add(x_literal, CTRL('V'), 0);
kb_add(x_yank, CTRL('Y'), 0);
kb_add(x_meta_yank, CTRL('['), 'y', 0);
```

893. Arrow keys.

⟨ Emacs mode keybindings 891 ⟩ +≡

```
kb_add(x_prev_com, CTRL('['), '[', 'A', 0);      /* up */
kb_add(x_next_com, CTRL('['), '[', 'B', 0);      /* down */
kb_add(x_mv_forw, CTRL('['), '[', 'C', 0);      /* right */
kb_add(x_mv_back, CTRL('['), '[', 'D', 0);      /* left */
kb_add(x_prev_com, CTRL('['), 'O', 'A', 0);      /* up */
kb_add(x_next_com, CTRL('['), 'O', 'B', 0);      /* down */
kb_add(x_mv_forw, CTRL('['), 'O', 'C', 0);      /* right */
kb_add(x_mv_back, CTRL('['), 'O', 'D', 0);      /* left */
```

894. More navigation keys.

⟨ Emacs mode keybindings 891 ⟩ +≡

```
kb_add(x_mv_begin, CTRL('['), '[', 'H', 0);      /* home */
kb_add(x_mv_end, CTRL('['), '[', 'F', 0);      /* end */
kb_add(x_mv_begin, CTRL('['), 'O', 'H', 0);      /* home */
kb_add(x_mv_end, CTRL('['), 'O', 'F', 0);      /* end */
kb_add(x_mv_begin, CTRL('['), '[', '1', '~', 0);      /* home */
kb_add(x_mv_end, CTRL('['), '[', '4', '~', 0);      /* end */
kb_add(x_mv_begin, CTRL('['), '[', '7', '~', 0);      /* home */
kb_add(x_mv_end, CTRL('['), '[', '8', '~', 0);      /* end */
```

895. Can't be bound.

⟨ Emacs mode keybindings 891 ⟩ +≡

```
kb_add(x_set_arg, CTRL('['), '0', 0);
kb_add(x_set_arg, CTRL('['), '1', 0);
kb_add(x_set_arg, CTRL('['), '2', 0);
kb_add(x_set_arg, CTRL('['), '3', 0);
kb_add(x_set_arg, CTRL('['), '4', 0);
kb_add(x_set_arg, CTRL('['), '5', 0);
kb_add(x_set_arg, CTRL('['), '6', 0);
kb_add(x_set_arg, CTRL('['), '7', 0);
kb_add(x_set_arg, CTRL('['), '8', 0);
kb_add(x_set_arg, CTRL('['), '9', 0);
```

896. Ctrl arrow keys.

⟨ Emacs mode keybindings 891 ⟩ +≡

```
kb_add(x_mv_end, CTRL('['), '[', '1', ';', '5', 'A', 0);      /* ctrl up */
kb_add(x_mv_begin, CTRL('['), '[', '1', ';', '5', 'B', 0);      /* ctrl down */
kb_add(x_mv_fword, CTRL('['), '[', '1', ';', '5', 'C', 0);      /* ctrl right */
kb_add(x_mv_bword, CTRL('['), '[', '1', ';', '5', 'D', 0);      /* ctrl left */
```

897. Emacs Mode Key Bindings. See queue(3).

seq is a pointer to the byte after this structure containing the string naming the function. Could instead have it last as **unsigned char** *seq*[ ] and save a pointer.

Not in ⟨ Type definitions 17 ⟩ because `TAILQ_ENTRY` requires `sys/queue.h`.

```
⟨ Emacs mode source 890 ⟩ +≡
  struct kb_entry {
    TAILQ_ENTRY(kb_entry)entry;
    unsigned char *seq;
    int len;
    struct x_ftab *ftab;
    void *args;
  };
```

898.
```
⟨ Emacs mode static variables 886 ⟩ +≡
  TAILQ_HEAD(kb_list, kb_entry);
  struct kb_list kblist ← TAILQ_HEAD_INITIALIZER(kblist);
```

899. Check if a key is bound. TODO: Why not return the pointer? Could avoid some duplication (eg. ⟨ Find a key-binding definition 921 ⟩).

```
⟨ Emacs mode source 890 ⟩ +≡
  static int kb_match(char *s)
  {
    int len ← strlen(s);
    struct kb_entry *k;
    TAILQ_FOREACH (k, &kblist, entry) {
      if (len > k→len) continue;
      if (memcmp(k→seq, s, len) ≡ 0) return (1);
    }
    return (0);
  }
```

900. Decode a key-binding definition.

```
⟨ Emacs mode source 890 ⟩ +≡
  static char *kb_decode(const char *s)
  {
    static char l[LINE + 1];
    unsigned int i, at ← 0;
    l[0] ← '\0';
    for (i ← 0; i < strlen(s); i++) {
      if (iscntrl((unsigned char) s[i])) {
        l[at++] ← '^';
        l[at++] ← UNCTRL(s[i]);
      }
      else l[at++] ← s[i];
      l[at] ← '\0';
    }
    return (l);
  }
```

901. For completeness, the encode is here.

```
⟨ Emacs mode source 890 ⟩ +≡
  static char *kb_encode(const char *s)
  {
    static char l[LINE + 1];
    int at ← 0;
    l[at] ← '\0';
    while (*s) {
      if (*s ≡ '^') {
        s++;
        if (*s ≥ '?') l[at++] ← CTRL(*s);
        else {
          l[at++] ← '^';
          s−−;
        }
      }
      else l[at++] ← *s;
      l[at] ← '\0';
      s++;
    }
    return (l);
  }
```

902. And printing it.

```
⟨ Emacs mode source 890 ⟩ +≡
  static void kb_print(struct kb_entry *k)
  {
    if (¬(k→ftab→xf_flags & XF_NOBIND)) shprintf("%s␣=␣%s\n", kb_decode(k→seq), k→ftab→xf_name);
    else if (k→args) {
      shprintf("%s␣=␣", kb_decode(k→seq));
      shprintf("'%s'\n", kb_decode(k→args));
    }
  }
```

903. Add a key binding.

```
⟨ Emacs mode source 890 ⟩ +≡
  static struct kb_entry *kb_add_string(kb_func func, void *args, char *str)
  {
    unsigned int ele, count;
    struct kb_entry *k;
    struct x_ftab *xf ← Λ;
    ⟨ Find func in x_ftab 904 ⟩
    if (kb_match(str)) {
      if (x_bind_quiet ≡ 0) bi_errorf("duplicate␣binding␣for␣%s", kb_decode(str));
      return (Λ);
    }
    count ← strlen(str);
    k ← alloc(sizeof *k + count + 1, AEDIT);
    k→seq ← (unsigned char *)(k + 1);
    k→len ← count;
    k→ftab ← xf;
    k→args ← args ? strdup(args) : Λ;
    strlcpy(k→seq, str, count + 1);
    TAILQ_INSERT_TAIL(&kblist, k, entry);
    return (k);
  }
```

904. ⟨ Find *func* in **x_ftab** 904 ⟩ ≡

```
  for (ele ← 0; ele < NELEM(x_ftab); ele++)
    if (x_ftab[ele].xf_func ≡ func) {
      xf ← (struct x_ftab *) &x_ftab[ele];
      break;
    }
  if (xf ≡ Λ) return (Λ);
```

This code is used in section 903.

905. ⟨ Emacs mode source 890 ⟩ +≡

```
  static struct kb_entry *kb_add(kb_func func, ...)
  {
    va_list ap;
    unsigned char ch;
    unsigned int i;
    char line[LINE + 1];
    va_start(ap, func);
    for (i ← 0; i < sizeof (line) − 1; i++) {
      ch ← va_arg(ap, unsigned int);
      if (ch ≡ 0) break;
      line[i] ← ch;
    }
    va_end(ap);
    line[i] ← '\0';
    return (kb_add_string(func, Λ, line));
  }
```

906. Remove a key binding.

```
⟨ Emacs mode source 890 ⟩ +≡
  static void kb_del(struct kb_entry *k)
  {
    TAILQ_REMOVE(&kblist, k, entry);
    free(k→args);
    afree(k, AEDIT);
  }
```

907. Should be in the next section?

```
⟨ Emacs mode source 890 ⟩ +≡
  int x_bind(const char *a1, const char *a2,
  int macro,      /* "bind -m" */
  int list)     /* "bind -l" */
  {
    unsigned int i;
    struct kb_entry *k, *kb;
    char in[LINE + 1];
    if (x_tty ≡ 0) {
      bi_errorf("cannot␣bind,␣not␣a␣tty");
      return (1);
    }
    if (list) {⟨ Show all Emacs mode function names and return 908 ⟩}
    if (a1 ≡ Λ) {⟨ Show all key bindings and return 909 ⟩}
    snprintf(in, sizeof in, "%s", kb_encode(a1));
    if (a2 ≡ Λ) {⟨ Print a key binding and return 910 ⟩}
    if (strlen(a2) ≡ 0) {⟨ Clear a key binding and return 911 ⟩}
    if (macro) {⟨ Set a macro key binding and return 912 ⟩}
    for (i ← 0; i < NELEM(x_ftab); i++) {
      if (x_ftab[i].xf_name ≡ Λ) continue;
      if (¬strcmp(x_ftab[i].xf_name, a2)) {⟨ Replace a non-macro key binding and return 913 ⟩}
    }
    bi_errorf("%s:␣no␣such␣function", a2);
    return (1);
  }
```

908. ⟨ Show all Emacs mode function names and return 908 ⟩ ≡

```
  for (i ← 0; i < NELEM(x_ftab); i++) {
    if (x_ftab[i].xf_name ≡ Λ) continue;
    if (x_ftab[i].xf_name ∧ ¬(x_ftab[i].xf_flags & XF_NOBIND)) shprintf("%s\n", x_ftab[i].xf_name);
  }
  return (0);
```

This code is used in section 907.

909. ⟨ Show all key bindings and return 909 ⟩ ≡

```
  TAILQ_FOREACH (k, &kblist, entry) kb_print(k);
  return (0);
```

This code is used in section 907.

910. ⟨ Print a key binding and **return** 910 ⟩ ≡

```
TAILQ_FOREACH (k, &kblist, entry)
  if (¬strcmp(k→seq, in)) {
    kb_print(k);
    return (0);
  }
shprintf("%s␣=␣%s\n", kb_decode(a1), "auto-insert");
return (0);
```

This code is used in section 907.

911. ⟨ Clear a key binding and **return** 911 ⟩ ≡

```
TAILQ_FOREACH_SAFE (k, &kblist, entry, kb)
  if (¬strcmp(k→seq, in)) {
    kb_del(k);
    break;
  }
return (0);
```

This code is used in section 907.

912. ⟨ Set a macro key binding and **return** 912 ⟩ ≡

```
TAILQ_FOREACH_SAFE (k, &kblist, entry, kb)
  if (¬strcmp(k→seq, in)) {
    kb_del(k);
    break;
  }
kb_add_string(x_ins_string, kb_encode(a2), in);
return (0);
```

This code is used in section 907.

913. ⟨ Replace a non-macro key binding and **return** 913 ⟩ ≡

```
TAILQ_FOREACH_SAFE (k, &kblist, entry, kb)
  if (¬strcmp(k→seq, in)) {
    kb_del(k);
    break;
  }
kb_add_string(x_ftab[i].xf_func, Λ, in);
return (0);
```

This code is used in section 907.

914. Emacs Mode Functions. There are over 800 of these which should be organised into some sort of coherency but for now here is a big grab-bag of everything. The first is the main run-time entry point *x_emacs* which is called by *x_read* to read the next line and from where, apart from initialisation, everything happens. Some vague attempt has been made to group related functions together.

```
⟨ Emacs mode source 890 ⟩ +≡
  int x_emacs(char *buf, size_t len)
  {
    struct kb_entry *k, *kmatch ← Λ;
    char line[LINE + 1];
    int at ← 0, ntries ← 0, submatch, ret;
    const char *p;
    ⟨ Prepare Emacs mode buffers 915 ⟩
    ⟨ Prepare Emacs mode tty 916 ⟩
    ⟨ Print the prompt in Emacs mode 917 ⟩
    ⟨ Do some kind of history thing in Emacs mode 918 ⟩
    x_literal_set ← 0;
    x_arg ← −1;
    x_last_command ← Λ;
    while (1) {
      ⟨ Wait for input in Emacs mode 919 ⟩
      ⟨ Handle input in Emacs mode 920 ⟩
    }
    ⟨ Reset Emacs mode meta sequence 924 ⟩
  }
```

915. ⟨ Prepare Emacs mode buffers 915 ⟩ ≡

```
  xbp ← xbuf ← buf;
  xend ← buf + len;
  xlp ← xcp ← xep ← buf;
  *xcp ← 0;
  xlp_valid ← true;
  xmp ← Λ;
  x_histp ← histptr + 1;
```

This code is used in section 914.

916. ⟨ Prepare Emacs mode tty 916 ⟩ ≡

```
  xx_cols ← x_cols;
  x_col ← promptlen(prompt, &p);
  prompt_skip ← p − prompt;
  x_adj_ok ← 1;
  prompt_redraw ← 1;
  if (x_col > xx_cols) x_col ← x_col − (x_col/xx_cols) * xx_cols;
  x_displen ← xx_cols − 2 − x_col;
  x_adj_done ← 0;
```

This code is used in section 914.

917. ⟨ Print the prompt in Emacs mode 917 ⟩ ≡

```
pprompt(prompt, 0);
if (x_displen < 1) {
  x_col ← 0;
  x_displen ← xx_cols − 2;
  x_e_putc('\n');
  prompt_redraw ← 0;
}
```

This code is used in section 914.

918. ⟨ Do some kind of history thing in Emacs mode 918 ⟩ ≡

```
if (x_nextcmd ≥ 0) {
  int off ← source¬line − x_nextcmd;
  if (histptr − history ≥ off) x_load_hist(histptr − off);
  x_nextcmd ← −1;
}
```

This code is used in section 914.

919. ⟨ Wait for input in Emacs mode 919 ⟩ ≡

```
x_flush();
if ((at ← x_e_getu8(line, at)) < 0) return 0;      /* get the next (utf-8-encoded) character */
ntries++;
if (x_arg ≡ −1) {      /* is set to -1 immediately prior to this loop */
  x_arg ← 1;
  x_arg_defaulted ← 1;
}
```

This code is used in section 914.

920. Handlier functions will return one of these values.

```
#define KSTD 0
#define KEOL 1 /* ^M/^J */
#define KINTR 2 /* ^G/^C */
```

⟨ Handle input in Emacs mode 920 ⟩ ≡

```
if (x_literal_set) { /* literal, so insert it */ /* starts off not; set by quote, bound to ^V */
  x_literal_set ← 0;
  submatch ← 0;
}
else {⟨ Find a key-binding definition 921 ⟩}
if (submatch ≡ 1 ∧ kmatch) {⟨ Process a matched key-binding in Emacs mode 922 ⟩}
else {⟨ Process an unmatched key-binding in Emacs mode 923 ⟩}
switch (ret) {
case KSTD:
  if (kmatch) x_last_command ← kmatch→ftab→xf_func;
  else x_last_command ← Λ;
  break;
case KEOL: ret ← xep − xbuf;
  return (ret);
  break;
case KINTR: trapsig(SIGINT);
  x_mode(false);
  unwind (LSHELL);
  x_arg ← −1;
  break;
default: bi_errorf("invalid␣return␣code"); /* can't happen */
}
```

This code is used in section 914.

921. ⟨ Find a key-binding definition 921 ⟩ ≡

```
submatch ← 0;
kmatch ← Λ;
TAILQ_FOREACH (k, &kblist, entry) {
  if (at > k→len) continue;
  if (memcmp(k→seq, line, at) ≡ 0) { /* sub match */
    submatch++;
    if (k→len ≡ at) kmatch ← k;
  }
  if (submatch > 1) break; /* see if we can abort search early */
}
```

This code is cited in section 899.

This code is used in section 920.

922. ⟨ Process a matched key-binding in Emacs mode 922 ⟩ ≡

```
if (kmatch→ftab→xf_func ≡ x_ins_string ∧ kmatch→args ∧ ¬macro_args) {
    /* treat macro string as input */
  macro_args ← kmatch→args;
  ret ← KSTD;
}
else ret ← kmatch→ftab→xf_func(line[at − 1]);
```

This code is used in section 920.

923. ⟨ Process an unmatched key-binding in Emacs mode 923 ⟩ ≡

```
if (submatch) continue;
if (ntries > 1) {
  ret ← x_error(0);      /* unmatched meta sequence */
}
else if (at > 1) {
  x_ins(line);
  ret ← KSTD;
}
else {
  ret ← x_insert(line[0]);
}
```

This code is used in section 920.

924. ⟨ Reset Emacs mode meta sequence 924 ⟩ ≡

```
at ← ntries ← 0;
if (x_arg_set) x_arg_set ← 0;      /* reset args next time around */
else x_arg ← −1;
```

This code is used in section 914.

925. Set a user argument value for the next function.

⟨ Emacs mode source 890 ⟩ +≡

```
static int x_set_arg(int c)
{
  int n ← 0;
  int first ← 1;
  for ( ; c ≥ 0 ∧ isdigit(c); c ← x_e_getc( ), first ← 0) n ← n * 10 + (c − '0');
  if (c < 0 ∨ first) {
    x_e_putc(BEL);
    x_arg ← 1;
    x_arg_defaulted ← 1;
  }
  else {
    x_e_ungetc(c);
    x_arg ← n;
    x_arg_defaulted ← 0;
    x_arg_set ← 1;
  }
  return KSTD;
}
```

926. Insertion & Deletion. A number of things insert stuff. *x_zots* may result in a call to *x_adjust*; we want *xcp* to reflect the new position.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_do_ins(const char *, size_t);
  static int x_ins(char *s)
  {
    char *cp ← xcp;
    int adj ← x_adj_done;
    if (x_do_ins(s, strlen(s)) < 0) return −1;
    xlp_valid ← false;
    x_lastcp( );
    x_adj_ok ← (xcp ≥ xlp);
    x_zots(cp);
    if (adj ≡ x_adj_done) {      /* x_adjust has not been called */
      for (cp ← xlp; cp > xcp; ) x_bs(*−−cp);
    }
    x_adj_ok ← 1;
    return 0;
  }
```

927.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_do_ins(const char *cp, size_t len)
  {
    if (xep + len ≥ xend) {
      x_e_putc(BEL);
      return −1;
    }
    memmove(xcp + len, xcp, xep − xcp + 1);
    memmove(xcp, cp, len);
    xcp += len;
    xep += len;
    return 0;
  }
```

928. Should allow tab and control characters.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_insert(int c)
  {
    char str[2];
    if (c ≡ 0) {
      x_e_putc(BEL);
      return KSTD;
    }
    str[0] ← c;
    str[1] ← '\0';
    while (x_arg−−) x_ins(str);
    return KSTD;
  }
```

929. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_ins_string(int c)
{
  return x_insert(c);
}
```

930. On the other hand this "inserts" whatever was already stored in the buffer.

⟨ Emacs mode source 890 ⟩ +≡

```
static int x_literal(int c)
{
  x_literal_set ← 1;
  return KSTD;
}
```

931. Other things delete stuff; *nc* bytes to the right of the cursor (including cursor position).

⟨ Emacs mode source 890 ⟩ +≡

```
static void x_delete(int nc, int push)
{
  int i, j;
  char *cp;
  if (nc ≡ 0) return;
  if (xmp ≠ Λ ∧ xmp > xcp) {
    if (xcp + nc > xmp) xmp ← xcp;
    else xmp -= nc;
  }
  if (push) x_push(nc);      /* This lets us yank a word we have deleted. */
  xep -= nc;
  cp ← xcp;
  j ← 0;
  i ← nc;
  while (i--) { j += x_size((unsigned char) *cp++); }
  memmove(xcp, xcp + nc, xep - xcp + 1);      /* Copies the null */
  x_adj_ok ← 0;      /* don't redraw */
  xlp_valid ← false;
  x_zots(xcp);
  if ((i ← xx_cols - 2 - x_col) > 0) {      /* If we are already filling the line there is no need to output
        "␣⟨backspace⟩" but if we must, make sure we do the minimum. */
    j ← (j < i) ? j : i;
    i ← j;
    while (i--) x_e_putc('␣');
    i ← j;
    while (i--) x_e_putc('\b');
  }      /* TODO Commented out here: x_goto(xcp) */
  x_adj_ok ← 1;
  xlp_valid ← false;
  for (cp ← x_lastcp(); cp > xcp; ) x_bs(*--cp);
  return;
}
```

932. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_del_back(int c)
{
  int col ← xcp − xbuf;
  if (col ≡ 0) {
    x_e_putc(BEL);
    return KSTD;
  }
  if (x_arg > col) x_arg ← col;
  while (x_arg < col ∧ isu8cont(xcp[−x_arg])) x_arg++;
  x_goto(xcp − x_arg);
  x_delete(x_arg, false);
  return KSTD;
}
```

933. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_del_char(int c)
{
  int nleft ← xep − xcp;
  if (¬nleft) {
    x_e_putc(BEL);
    return KSTD;
  }
  if (x_arg > nleft) x_arg ← nleft;
  while (x_arg < nleft ∧ isu8cont(xcp[x_arg])) x_arg++;
  x_delete(x_arg, false);
  return KSTD;
}
```

934. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_del_bword(int c)
{
  x_delete(x_bword( ), true);
  return KSTD;
}
```

935. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_del_fword(int c)
{
  x_delete(x_fword( ), true);
  return KSTD;
}
```

936. Navigation. Still other things move the cursor; back ...

```
#define is_cfs(c) (c ≡ '␣' ∨ c ≡ '\t' ∨ c ≡ '"' ∨ c ≡ '\'')  /* Separator for completion */
#define is_mfs(c) (¬(isalnum((unsigned char) c) ∨
        c ≡ '_' ∨ c ≡ '$' ∨ c & 0x80))  /* Separator for motion */
```

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_bword(void)
  {
    int nc ← 0;
    char *cp ← xcp;
    if (cp ≡ xbuf) {
      x_e_putc(BEL);
      return 0;
    }
    while (x_arg −−) {
      while (cp ≠ xbuf ∧ is_mfs(cp[−1])) {
        cp −−;
        nc ++;
      }
      while (cp ≠ xbuf ∧ ¬is_mfs(cp[−1])) {
        cp −−;
        nc ++;
      }
    }
    x_goto(cp);
    return nc;
  }
```

937. ... and forth, ...

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_fword(void)
  {
    int nc ← 0;
    char *cp ← xcp;
    if (cp ≡ xep) {
      x_e_putc(BEL);
      return 0;
    }
    while (x_arg −−) {
      while (cp ≠ xep ∧ is_mfs(*cp)) {
        cp ++;
        nc ++;
      }
      while (cp ≠ xep ∧ ¬is_mfs(*cp)) {
        cp ++;
        nc ++;
      }
    }
    return nc;
  }
```

938. ... wherever ...

```
⟨ Emacs mode source 890 ⟩ +≡
  static void x_goto(char *cp)
  {
    if (cp < xbp ∨ cp ≥ (xbp + x_displen)) {      /* we are heading off screen */
      xcp ← cp;
      x_adjust( );
    }
    else if (cp < xcp) {      /* move back */
      while (cp < xcp) x_bs((unsigned char) *--xcp);
    }
    else if (cp > xcp) {      /* move forward */
      while (cp > xcp) x_zotc((unsigned char) *xcp++);
    }
  }
```

939. ... and deleting en route.

```
⟨ Emacs mode source 890 ⟩ +≡
  static void x_bs(int c)
  {
    int i;
    i ← x_size(c);
    while (i--) x_e_putc('\b');
  }
```

940.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_mv_bword(int c)
  {
    (void) x_bword( );
    return KSTD;
  }
```

941.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_mv_fword(int c)
  {
    x_goto(xcp + x_fword( ));
    return KSTD;
  }
```

942. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_mv_back(int c)
{
  int col ← xcp − xbuf;
  if (col ≡ 0) {
    x_e_putc(BEL);
    return KSTD;
  }
  if (x_arg > col) x_arg ← col;
  while (x_arg < col ∧ isu8cont(xcp[−x_arg])) x_arg++;
  x_goto(xcp − x_arg);
  return KSTD;
}
```

943. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_mv_forw(int c)
{
  int nleft ← xep − xcp;
  if (¬nleft) {
    x_e_putc(BEL);
    return KSTD;
  }
  if (x_arg > nleft) x_arg ← nleft;
  while (x_arg < nleft ∧ isu8cont(xcp[x_arg])) x_arg++;
  x_goto(xcp + x_arg);
  return KSTD;
}
```

944. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_mv_begin(int c)
{
  x_goto(xbuf);
  return KSTD;
}
```

945. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_mv_end(int c)
{
  x_goto(xep);
  return KSTD;
}
```

946. Termination.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_newline(int c)
  {
    x_e_putc('\r');
    x_e_putc('\n');
    x_flush();
    *xep++ ← '\n';
    return KEOL;
  }
```

947.
```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_nl_next_com(int c)
  {
    x_nextcmd ← source¬line − (histptr − x_histp) + 1;
    return (x_newline(c));
  }
```

948.
```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_end_of_text(int c)
  {
    x_zotc(edchars.eof);
    x_putc('\r');
    x_putc('\n');
    x_flush();
    return KEOL;
  }
```

949.
```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_eot_del(int c)
  {
    if (xep ≡ xbuf ∧ x_arg_defaulted) return (x_end_of_text(c));
    else return (x_del_char(c));
  }
```

950.
```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_del_line(int c)
  {
    int i, j;
    *xep ← 0;
    i ← xep − xbuf;
    j ← x_size_str(xbuf);
    xcp ← xbuf;
    x_push(i);
    xlp ← xbp ← xep ← xbuf;
    xlp_valid ← true;
    *xcp ← 0;
    xmp ← Λ;
    x_redraw(j);
    return KSTD;
  }
```

951. Mutation. Comment or uncomment the current line.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_comment(int c)
  {
    int oldsize ← x_size_str(xbuf);
    int len ← xep − xbuf;
    int ret ← x_do_comment(xbuf, xend − xbuf, &len);
    if (ret < 0) x_e_putc(BEL);
    else {
      xep ← xbuf + len;
      ∗xep ← '\0';
      xcp ← xbp ← xbuf;
      x_redraw(oldsize);
      if (ret > 0) return x_newline('\n');
    }
    return KSTD;
  }
```

952. What transpose is meant to do seems to be up for debate. This is a general summary of the options; the text is "abcd" with the upper case character or underscore indicating the cursor position:

```
Before | After || Before | After
 abCd    abdC  ||  abcd_    ⟨bell⟩ —AT&T ksh in GNU Emacs mode.
 abCd    bcCd  ||  abcd_    abdc_ – AT&T ksh in Gosling Emacs mode.
 abCd    acbD  ||  abcd_    abdc_ – GNU Emacs.
```

This ksh currently goes with GNU behavior since its author believed this is the most common version of Emacs, unless in gmacs mode in which case it acts like AT&T ksh in Gosling Emacs mode. This "should really" be broken up into 3 functions so users can bind to the one they want.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_transpose(int c)
  {
    char tmp;
    if (xcp ≡ xbuf) {
      x_e_putc(BEL);
      return KSTD;
    }
    else if (xcp ≡ xep ∨ Flag(FGMACS)) {⟨ Transpose Gosling style 953 ⟩}
    else {⟨ Transpose GNU style 954 ⟩}
    return KSTD;
  }
```

953. Swap two characters before the cursor without changing the cursor position.

```
⟨ Transpose Gosling style 953 ⟩ ≡
  if (xcp − xbuf ≡ 1) {
    x_e_putc(BEL);
    return KSTD;
  }
  x_bs(xcp[−1]);
  x_bs(xcp[−2]);
  x_zotc(xcp[−1]);
  x_zotc(xcp[−2]);
  tmp ← xcp[−1];
  xcp[−1] ← xcp[−2];
  xcp[−2] ← tmp;
```

This code is used in section 952.

954. Swap the characters before and under the cursor and move cursor position along one.

```
⟨ Transpose GNU style 954 ⟩ ≡
  x_bs(xcp[−1]);
  x_zotc(xcp[0]);
  x_zotc(xcp[−1]);
  tmp ← xcp[−1];
  xcp[−1] ← xcp[0];
  xcp[0] ← tmp;
  x_bs(xcp[0]);
  x_goto(xcp + 1);
```

This code is used in section 952.

955.
```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_fold_upper(int c)
  { return x_fold_case('U'); }
```

956.
```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_fold_lower(int c)
  { return x_fold_case('L'); }
```

957.
```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_fold_capitalize(int c)
  { return x_fold_case('C'); }
```

958. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_fold_case(int c)
{
  char *cp ← xcp;
  if (cp ≡ xep) {
    x_e_putc(BEL);
    return KSTD;
  }
  while (x_arg --) {
    while (cp ≠ xep ∧ is_mfs(*cp)) cp++;      /* skip over any white-space */
    if (cp ≠ xep) { ⟨ Case-fold the first character 959 ⟩ cp++; }
    while (cp ≠ xep ∧ ¬is_mfs(*cp)) { ⟨ Case-fold the remaining characters 960 ⟩ cp++; }
  }
  x_goto(cp);
  return KSTD;
}
```

959. The first character is folded separately since it may require a different action than for the rest.

⟨ Case-fold the first character 959 ⟩ ≡

```
if (c ≡ 'L') {      /* lowercase */
  if (isupper((unsigned char) *cp)) *cp ← tolower((unsigned char) *cp);
}
else {      /* uppercase, capitalize */
  if (islower((unsigned char) *cp)) *cp ← toupper((unsigned char) *cp);
}
```

This code is used in section 958.

960. ⟨ Case-fold the remaining characters 960 ⟩ ≡

```
if (c ≡ 'U') {      /* uppercase */
  if (islower((unsigned char) *cp)) *cp ← toupper((unsigned char) *cp);
}
else {      /* lowercase, capitalize */
  if (isupper((unsigned char) *cp)) *cp ← tolower((unsigned char) *cp);
}
```

This code is used in section 958.

961. Drawing. Sets *xlp* (and **return**s it) to the last byte after *xbp* after up to *x_displen* characters. The sequence **for** (*cp* ← *x_lastcp*( ); *cp* > *xcp*; *cp*) *x_bs*(∗−−*cp*) will position the cursor currectly on the screen.

```
⟨ Emacs mode source 890 ⟩ +≡
  static char *x_lastcp(void)
  {
    char *rcp;
    int i;
    if (¬xlp_valid) {
      for (i ← 0, rcp ← xbp; rcp < xep ∧ i < x_displen; rcp++) i += x_size((unsigned char) *rcp);
      xlp ← rcp;
    }
    xlp_valid ← true;
    return (xlp);
  }
```

962. The width in tty cells of a character's "glyph".

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_size(int c)
  {
    if (c ≡ '\t') return 4;      /* Kludge: tabs are always four spaces. */
    if (iscntrl(c))      /* control char */
      return 2;
    if (isu8cont(c)) return 0;
    return 1;
  }
```

963. As above, cumulative over a string.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_size_str(char *cp)
  {
    int size ← 0;
    while (*cp) size += x_size(*cp++);
    return size;
  }
```

964.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_draw_line(int c)
  {
    x_redraw(−1);
    return KSTD;
  }
```

965.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_clear_screen(int c)
  {
    x_redraw(−2);
    return KSTD;
  }
```

966. Redraw (part of) the line. A non-negative limit is the screen column up to which needs redrawing. A limit of -1 redraws on a new line, while a limit of -2 (attempts to) clear the screen.

```
⟨ Emacs mode source 890 ⟩ +≡
  static void x_redraw(int limit)
  {
    int i, j, truncate ← 0;
    char *cp;
    x_adj_ok ← 0;
    if (limit ≡ −2) {⟨ Clear the screen 967 ⟩}
    else if (limit ≡ −1) x_e_putc('\n');
    else if (limit ≥ 0) x_e_putc('\r');
    x_flush( );
    if (xbp ≡ xbuf) {⟨ Redraw the last line of the prompt 968 ⟩}
    ⟨ Draw on the tty from the input buffer 969 ⟩
    if (xbp ≠ xbuf ∨ xep > xlp) limit ← xx_cols;      /* the line doesn't fit */
    if (limit ≥ 0) {⟨ Draw a continuation character at the end of the tty 970 ⟩}
    for (cp ← xlp; cp > xcp; ) x_bs(*−−cp);      /* move the cursor back */
    x_adj_ok ← 1;
#ifdef DEBUG
    x_flush( );
#endif
    return;
  }
```

967. ⟨ Clear the screen 967 ⟩ ≡

```
  int cleared ← 0;
#ifndef SMALL
  if (cur_term ≠ Λ ∧ clear_screen ≠ Λ) {
    if (tputs(clear_screen, 1, x_putc) ≠ ERR) cleared ← 1;
  }
#endif
  if (¬cleared) x_e_putc('\n');
```

This code is used in section 966.

968. ⟨ Redraw the last line of the prompt 968 ⟩ ≡

```
  x_col ← promptlen(prompt, Λ);
  if (x_col > xx_cols) truncate ← (x_col/xx_cols) * xx_cols;
  if (prompt_redraw) pprompt(prompt + prompt_skip, truncate);
```

This code is used in section 966.

969. ⟨ Draw on the tty from the input buffer 969 ⟩ ≡

```
  if (x_col > xx_cols) x_col ← x_col − (x_col/xx_cols) * xx_cols;
  x_displen ← xx_cols − 2 − x_col;
  if (x_displen < 1) {
    x_col ← 0;
    x_displen ← xx_cols − 2;
  }
  xlp_valid ← false;
  x_lastcp( );
  x_zots(xbp);
```

This code is used in section 966.

970. ⟨ Draw a continuation character at the end of the tty 970 ⟩ ≡

```
if (xep > xlp) i ← 0;      /* the space available is full */
else i ← limit − (xlp − xbp);      /* TODO: I think this line is "wrong" ... */
for (j ← 0; j < i ∧ x_col < (xx_cols − 2); j++) x_e_putc('␣');      /* but this condition saves it */
i ← '␣';
if (xep > xlp) {      /* more to the right */
  if (xbp > xbuf) i ← '*';      /* ... and the left */
  else i ← '>';
}
else if (xbp > xbuf) i ← '<';      /* more to the left */
x_e_putc(i);
j++;
while (j−−) x_e_putc('\b');
```

This code is used in section 966.

971. Move backwards on the tty as many characters are between *xbuf* and *str* then overwrite them with their replacement using *x_zotc*.

⟨ Emacs mode source 890 ⟩ +≡

```
static void x_zots(char *str)
{
  int adj ← x_adj_done;
  if (str > xbuf ∧ isu8cont(*str)) {
    while (str > xbuf ∧ isu8cont(*str)) str−−;
    x_e_putc('\b');
  }
  x_lastcp( );
  while (*str ∧ str < xlp ∧ adj ≡ x_adj_done) x_zotc(*str++);
}
```

972. Draws a character on the tty, encoding control characters and tabs specially.

⟨ Emacs mode source 890 ⟩ +≡

```
static void x_zotc(int c)
{
  if (c ≡ '\t') {      /* Kludge: tabs are always four spaces. */
    x_e_puts("␣␣␣␣");
  }
  else if (iscntrl(c)) {
    x_e_putc('^');
    x_e_putc(UNCTRL(c));
  }
  else x_e_putc(c);
}
```

973. Called when the bounds of the edit window have been exceeded. Redraw the line after adjusting the starting point etc.

It increments *x_adj_done* so that functions like *x_ins* and *x_delete* know that they can skip the *x_bs* stuff which has already been done here.

```
⟨ Emacs mode source 890 ⟩ +≡
  static void x_adjust(void)
  {
    x_adj_done++;      /* flag the fact that we were called. */
    if ((xbp ← xcp − (x_displen/2)) < xbuf)      /* we had a problem if the prompt length > xx_cols/2 */
      xbp ← xbuf;
    xlp_valid ← false;
    x_redraw(xx_cols);
    x_flush();
  }
```

974. Searching/History.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_search_char_back(int c)
  {
    char *cp ← xcp, *p;
    c ← x_e_getc( );
    for ( ; x_arg−−; cp ← p)
      for (p ← cp; ; ) {
        if (p−− ≡ xbuf) p ← xep;
        if (c < 0 ∨ p ≡ cp) {
          x_e_putc(BEL);
          return KSTD;
        }
        if (*p ≡ c) break;
      }
    x_goto(cp);
    return KSTD;
  }
```

975.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_search_char_forw(int c)
  {
    char *cp ← xcp;
    *xep ← '\0';
    c ← x_e_getc( );
    while (x_arg−−) {
      if (c < 0 ∨ ((cp ← (cp ≡ xep) ? Λ : strchr(cp + 1, c)) ≡ Λ ∧ (cp ← strchr(xbuf, c)) ≡ Λ)) {
        x_e_putc(BEL);
        return KSTD;
      }
    }
    x_goto(cp);
    return KSTD;
  }
```

976.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_beg_hist(int c)
  {
    x_load_hist(history);
    return KSTD;
  }
```

977.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_end_hist(int c)
  {
    x_load_hist(histptr);
    return KSTD;
  }
```

978. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_prev_com(int c)
{
  x_load_hist(x_histp - x_arg);
  return KSTD;
}
```

979. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_next_com(int c)
{
  x_load_hist(x_histp + x_arg);
  return KSTD;
}
```

980. Go to a particular history number obtained from the argument. If no argument is given history 1 is probably not what we want so we'll simply go to the oldest one.

⟨ Emacs mode source 890 ⟩ +≡
```
static int x_goto_hist(int c)
{
  if (x_arg_defaulted) x_load_hist(history);
  else x_load_hist(histptr + x_arg - source¬line);
  return KSTD;
}
```

981. ⟨ Emacs mode source 890 ⟩ +≡
```
static void x_load_hist(char **hp)
{
  int oldsize;
  if (hp < history ∨ hp > histptr) {
    x_e_putc(BEL);
    return;
  }
  x_histp ← hp;
  oldsize ← x_size_str(xbuf);
  strlcpy(xbuf, *hp, xend - xbuf);
  xbp ← xbuf;
  xep ← xcp ← xbuf + strlen(xbuf);
  xlp_valid ← false;
  if (xep ≤ x_lastcp()) x_redraw(oldsize);
  x_goto(xep);
}
```

982. Finds a function for a bare keystroke. Another variant of *kb_match*.

```
⟨ Emacs mode source 890 ⟩ +≡
  static kb_func kb_find_hist_func(char c)
  {
    struct kb_entry *k;
    char line[LINE + 1];
    line[0] ← c;
    line[1] ← '\0';
    TAILQ_FOREACH (k, &kblist, entry)
      if (¬strcmp(k→seq, line)) return (k→ftab→xf_func);
    return (x_insert);
  }
```

983. Reverse incremental history search.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_search_hist(int c)
  {
    int offset ← −1;      /* offset of match in xbuf, else -1 */
    char pat[256 + 1];      /* pattern buffer */
    char *p ← pat;
    int (*f)(int);      /* kb_func f */
    *p ← '\0';
    while (1) {
      if (offset < 0) {
        x_e_puts("\nI-search:␣");
        x_e_puts(pat);
      }
      x_flush();
      if ((c ← x_e_getc()) < 0) return KSTD;
      f ← kb_find_hist_func(c);
      if (c ≡ CTRL('[')) { x_e_ungetc(c); break; }
      else if (f ≡ x_search_hist) offset ← x_search(pat, 0, offset);      /* search for the next entry */
      else if (f ≡ x_del_back) {⟨ Remove a history search pattern character in Emacs mode 984 ⟩}
      else if (f ≡ x_insert) {⟨ Add a character to the Emacs mode history search 985 ⟩}
      else { x_e_ungetc(c); break; }      /* other command */
    }
    if (offset < 0) x_redraw(−1);
    return KSTD;
  }
```

984. ⟨ Remove a history search pattern character in Emacs mode 984 ⟩ ≡

```
  if (p ≡ pat) {
    offset ← −1;
    break;
  }
  if (p > pat) *−−p ← '\0';
  if (p ≡ pat) offset ← −1;
  else offset ← x_search(pat, 1, offset);
  continue;
```

This code is used in section 983.

985. ⟨ Add a character to the Emacs mode history search 985 ⟩ ≡

```
if (p ≥ &pat[sizeof (pat) − 1]) {      /* overflow check */
  x_e_putc(BEL);
  continue;
}
*p++ ← c, *p ← '\0';
if (offset ≥ 0) {      /* already have partial match */
  offset ← x_match(xbuf, pat);
  if (offset ≥ 0) {
    x_goto(xbuf + offset + (p − pat) − (*pat ≡ '^'));
    continue;
  }
}
offset ← x_search(pat, 0, offset);
```

This code is used in section 983.

986. Search backward from current line.

⟨ Emacs mode source 890 ⟩ +≡

```
static int x_search(char *pat, int sameline, int offset)
{
  char **hp;
  int i;
  for (hp ← x_histp − (sameline ? 0 : 1); hp ≥ history; −−hp) {
    i ← x_match(*hp, pat);
    if (i ≥ 0) {
      if (offset < 0) x_e_putc('\n');
      x_load_hist(hp);
      x_goto(xbuf + i + strlen(pat) − (*pat ≡ '^'));
      return i;
    }
  }
  x_e_putc(BEL);
  x_histp ← histptr;
  return −1;
}
```

987. Return the position of the first match of *pat* in *str*, or -1.

⟨ Emacs mode source 890 ⟩ +≡

```
static int x_match(char *str, char *pat)
{
  if (*pat ≡ '^') {
    return (strncmp(str, pat + 1, strlen(pat + 1)) ≡ 0) ? 0 : −1;
  }
  else {
    char *q ← strstr(str, pat);
    return (q ≡ Λ) ? −1 : q − str;
  }
}
```

988. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_prev_histword(int c)
{
  char *rcp;
  char *cp;
  cp ← *histptr;
  if (¬cp) x_e_putc(BEL);
  else if (x_arg_defaulted) {⟨ Copy the previous command's last argument 989 ⟩}
  else {⟨ Copy the previous command's x_arg^th argument 990 ⟩}
  return KSTD;
}
```

989. ⟨ Copy the previous command's last argument 989 ⟩ ≡

```
rcp ← &cp[strlen(cp) − 1];
while (rcp > cp ∧ is_cfs(*rcp)) rcp−−;      /* ignore white-space after the last word */
while (rcp > cp ∧ ¬is_cfs(*rcp)) rcp−−;
if (is_cfs(*rcp)) rcp++;
x_ins(rcp);
```

This code is used in section 988.

990. ⟨ Copy the previous command's x_arg^{th} argument 990 ⟩ ≡

```
rcp ← cp;
while (*rcp ∧ is_cfs(*rcp)) rcp++;      /* ignore white-space at the start of the line */
while (x_arg−− > 1) {
  while (*rcp ∧ ¬is_cfs(*rcp)) rcp++;
  while (*rcp ∧ is_cfs(*rcp)) rcp++;
}
cp ← rcp;
while (*rcp ∧ ¬is_cfs(*rcp)) rcp++;
c ← *rcp;
*rcp ← '\0';
x_ins(cp);
*rcp ← c;
```

This code is used in section 988.

991. Kill Ring & Region.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_kill(int c)
  {
    int col ← xcp − xbuf;
    int lastcol ← xep − xbuf;
    int ndel;
    if (x_arg_defaulted) x_arg ← lastcol;
    else if (x_arg > lastcol) x_arg ← lastcol;
    while (x_arg < lastcol ∧ isu8cont(xbuf[x_arg])) x_arg++;
    ndel ← x_arg − col;
    if (ndel < 0) {
      x_goto(xbuf + x_arg);
      ndel ← −ndel;
    }
    x_delete(ndel, true);
    return KSTD;
  }
```

992. This puts things into *killstack*—should probably bring its users closer.

```
⟨ Emacs mode source 890 ⟩ +≡
  static void x_push(int nchars)
  {
    char *cp ← str_nsave(xcp, nchars, AEDIT);
    afree(killstack[killsp], AEDIT);
    killstack[killsp] ← cp;
    killsp ← (killsp + 1) % KILLSIZE;
  }
```

993.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_yank(int c)
  {
    if (killsp ≡ 0) killtp ← KILLSIZE;
    else killtp ← killsp;
    killtp−−;
    if (killstack[killtp] ≡ 0) {
      x_e_puts("\nnothing␣to␣yank");
      x_redraw(−1);
      return KSTD;
    }
    xmp ← xcp;
    x_ins(killstack[killtp]);
    return KSTD;
  }
```

994. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_meta_yank(int c)
{
  int len;
  if ((x_last_command ≠ x_yank ∧ x_last_command ≠ x_meta_yank) ∨ killstack[killtp] ≡ 0) {
    killtp ← killsp;
    x_e_puts("\nyank␣something␣first");
    x_redraw(−1);
    return KSTD;
  }
  len ← strlen(killstack[killtp]);
  x_goto(xcp − len);
  x_delete(len, false);
  do {
    if (killtp ≡ 0) killtp ← KILLSIZE − 1;
    else killtp−−;
  } while (killstack[killtp] ≡ 0);
  x_ins(killstack[killtp]);
  return KSTD;
}
```

995. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_set_mark(int c)
{
  xmp ← xcp;
  return KSTD;
}
```

996. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_xchg_point_mark(int c)
{
  char *tmp;
  if (xmp ≡ Λ) {
    x_e_putc(BEL);
    return KSTD;
  }
  tmp ← xmp;
  xmp ← xcp;
  x_goto(tmp);
  return KSTD;
}
```

997. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_kill_region(int c)
{
  int rsize;
  char *xr;
  if (xmp ≡ Λ) {
    x_e_putc(BEL);
    return KSTD;
  }
  if (xmp > xcp) {
    rsize ← xmp − xcp;
    xr ← xcp;
  }
  else {
    rsize ← xcp − xmp;
    xr ← xmp;
  }
  x_goto(xr);
  x_delete(rsize, true);
  xmp ← xr;
  return KSTD;
}
```

998. File/command name completion routines.

⟨ Emacs mode source 890 ⟩ +≡
```
static int x_comp_comm(int c)
{ do_complete(XCF_COMMAND, CT_COMPLETE); return KSTD; }
```

999. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_list_comm(int c)
{ do_complete(XCF_COMMAND, CT_LIST); return KSTD; }
```

1000. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_complete(int c)
{ do_complete(XCF_COMMAND_FILE, CT_COMPLETE); return KSTD; }
```

1001. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_enumerate(int c)
{ do_complete(XCF_COMMAND_FILE, CT_LIST); return KSTD; }
```

1002. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_comp_file(int c)
{ do_complete(XCF_FILE, CT_COMPLETE); return KSTD; }
```

1003. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_list_file(int c)
{ do_complete(XCF_FILE, CT_LIST); return KSTD; }
```

1004. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_comp_list(int c)
{ do_complete(XCF_COMMAND_FILE, CT_COMPLIST); return KSTD; }
```

1005. Arguments for *do_complete*.

⟨ Type definitions 17 ⟩ +≡
```
typedef enum {
  CT_LIST,      /* (M-=) list the possible completions */
  CT_COMPLETE,    /* (M-⟨escape⟩) complete to longest prefix */
  CT_COMPLIST     /* (M-?) complete and then list (if non-exact) */
} Comp_type;
```

1006. ⟨ Emacs mode source 890 ⟩ +≡

```
static void do_complete(int flags,      /* XCF_{COMMAND,FILE,COMMAND_FILE} */
Comp_type type)
{
  char **words;
  int nwords;
  int start, end, nlen, olen;
  int is_command;
  int completed ← 0;
  nwords ← x_cf_glob(flags, xbuf, xep − xbuf, xcp − xbuf, &start, &end, &words, &is_command);
    /* no match */
  if (nwords ≡ 0) {
    x_e_putc(BEL);
    return;
  }
  if (type ≡ CT_LIST) {
    x_print_expansions(nwords, words, is_command);
    x_redraw(0);
    x_free_words(nwords, words);
    return;
  }
  olen ← end − start;
  nlen ← x_longest_prefix(nwords, words);      /* complete */
  if (nwords ≡ 1 ∨ nlen > olen) {
    x_goto(xbuf + start);
    x_delete(olen, false);
    x_escape(words[0], nlen, x_do_ins);
    x_adjust( );
    completed ← 1;
  }     /* add space if single non-dir match */
  if (nwords ≡ 1 ∧ words[0][nlen − 1] ≠ '/') {
    x_ins("␣");
    completed ← 1;
  }
  if (type ≡ CT_COMPLIST ∧ ¬completed) {
    x_print_expansions(nwords, words, is_command);
    completed ← 1;
  }
  if (completed) x_redraw(0);
  x_free_words(nwords, words);
}
```

1007. ⟨ Emacs mode source 890 ⟩ +≡

```
static int x_expand(int c)
{
  char **words;
  int nwords ← 0;
  int start, end;
  int is_command;
  int i;
  nwords ← x_cf_glob(XCF_FILE, xbuf, xep − xbuf, xcp − xbuf, &start, &end, &words, &is_command);
  if (nwords ≡ 0) {
    x_e_putc(BEL);
    return KSTD;
  }
  x_goto(xbuf + start);
  x_delete(end − start, false);
  for (i ← 0; i < nwords; ) {
    if (x_escape(words[i], strlen(words[i]), x_do_ins) < 0 ∨ (++i < nwords ∧ x_ins("␣") < 0)) {
      x_e_putc(BEL);
      return KSTD;
    }
  }
  x_adjust( );
  return KSTD;
}
```

1008. I/O.

⟨ Emacs mode static variables 886 ⟩ +≡
```
static int unget_char ← −1;
```

1009. ⟨ Emacs mode source 890 ⟩ +≡
```
static void x_e_ungetc(int c)
{
  unget_char ← c;
}
```

1010. ⟨ Emacs mode source 890 ⟩ +≡
```
static int x_e_getc(void)
{
  int c;
  if (unget_char ≥ 0) {
    c ← unget_char;
    unget_char ← −1;
  }
  else if (macro_args) {
    c ← ∗macro_args++;
    if (¬c) {
      macro_args ← Λ;
      c ← x_getc( );
    }
  }
  else c ← x_getc( );
  return c;
}
```

1011. Emacs and vi modes define this function with the same name differently.

⟨ Emacs mode source 890 ⟩ +≡
```
int isu8cont(unsigned char c)
{
  return (c & (0x80 | 0x40)) ≡ 0x80;
}
```

1012. In the following, comments refer to violations of the inequality tests at the ends of the lines. See the utf8(7) manual page for details.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_e_getu8(char *buf, int off)
  {
    int c, cc, len;
    c ← x_e_getc( );
    if (c ≡ −1) return −1;
    buf[off ++] ← c;
    if ((c & 0xf8) ≡ 0xf0 ∧ c < 0xf5) len ← 4;      /* beyond Unicode */
    else if ((c & 0xf0) ≡ 0xe0) len ← 3;
    else if ((c & 0xe0) ≡ 0xc0 ∧ c > 0xc1) len ← 2;      /* use single byte */
    else len ← 1;
    for ( ; len > 1; len −−) {
      cc ← x_e_getc( );
      if (cc ≡ −1) break;
      if (isu8cont(cc) ≡ 0 ∨
          (c ≡ 0xe0 ∧ len ≡ 3 ∧ cc < 0xa0) ∨      /* use 2 bytes */
          (c ≡ 0xed ∧ len ≡ 3 ∧ cc > 0x9f) ∨      /* surrogates */
          (c ≡ 0xf0 ∧ len ≡ 4 ∧ cc < 0x90) ∨      /* use 3 bytes */
          (c ≡ 0xf4 ∧ len ≡ 4 ∧ cc > 0x8f)) {      /* beyond Unicode */
        x_e_ungetc(cc);
        break;
      }
      buf[off ++] ← cc;
    }
    buf[off] ← '\0';
    return off;
  }
```

1013.

```
⟨ Emacs mode source 890 ⟩ +≡
  static void x_e_putc(int c)
  {
    if (c ≡ '\r' ∨ c ≡ '\n') x_col ← 0;
    if (x_col < xx_cols) {
      x_putc(c);
      switch (c) {
      case BEL: break;
      case '\r': case '\n': break;
      case '\b': x_col −−;
        break;
      default:
        if (¬isu8cont(c)) x_col ++;
        break;
      }
    }
    if (x_adj_ok ∧ (x_col < 0 ∨ x_col ≥ (xx_cols − 2))) x_adjust( );
  }
```

1014. ⟨ Emacs mode source 890 ⟩ +≡

```
static void x_e_puts(const char *s)
{
  int adj ← x_adj_done;
  while (*s ∧ adj ≡ x_adj_done) x_e_putc(*s++);
}
```

1015. ⟨ Emacs mode source 890 ⟩ +≡

```
#ifdef DEBUG
static int x_debug_info(int c)
{
  x_flush( );
  shellf("\nksh debug:\n");
  shellf("\tx_col == %d,\t\tx_cols == %d,\tx_displen == %d\n", x_col, xx_cols, x_displen);
  shellf("\txcp == 0x%lx,\txep == 0x%lx\n", (long) xcp, (long) xep);
  shellf("\txbp == 0x%lx,\txbuf == 0x%lx\n", (long) xbp, (long) xbuf);
  shellf("\txlp == 0x%lx\n", (long) xlp);
  shellf("\txlp == 0x%lx\n", (long) x_lastcp( ));
  shellf("\n");
  x_redraw(-1);
  return 0;
}
#endif
```

1016. Bits. Emacs mode is initialised simply by saving the default key bindings.

```
⟨ Emacs mode source 890 ⟩ +≡
  void x_init_emacs(void)
  {
    x_tty ← 1;
    ainit(AEDIT);
    x_nextcmd ← −1;
    TAILQ_INIT(&kblist);
    ⟨ Emacs mode keybindings 891 ⟩
  }
```

1017. Terminal-handling key bindings are always set when entering emacs mode.

```
⟨ Emacs mode source 890 ⟩ +≡
  void x_emacs_keys(X_chars *ec)
  {
    x_bind_quiet ← 1;
    if (ec→erase ≥ 0) {
      kb_add(x_del_back, ec→erase, 0);
      kb_add(x_del_bword, CTRL('['), ec→erase, 0);
    }
    if (ec→kill ≥ 0) kb_add(x_del_line, ec→kill, 0);
    if (ec→werase ≥ 0) kb_add(x_del_bword, ec→werase, 0);
    if (ec→intr ≥ 0) kb_add(x_abort, ec→intr, 0);
    if (ec→quit ≥ 0) kb_add(x_noop, ec→quit, 0);
    x_bind_quiet ← 0;
  }
```

1018.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_abort(int c)
  {     /* x_zotc(c); */
    xlp ← xep ← xcp ← xbp ← xbuf;
    xlp_valid ← true;
    *xcp ← 0;
    return KINTR;
  }
```

1019.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_error(int c)
  {
    x_e_putc(BEL);
    return KSTD;
  }
```

1020.

```
⟨ Emacs mode source 890 ⟩ +≡
  static int x_noop(int c)
  { return KSTD; }
```

1021. vi. John Rochester's implementation of vi command editing for nsh was bludgeoned to fit pdksh by Larry Bouzane, Jeff Sparkes and Eric Gisin.

```
⟨ vi.c 1021 ⟩ ≡
#include "config.h"
#ifdef VI
#include <sys/stat.h>      /* completion */
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#ifndef SMALL
#include <term.h>
#include <curses.h>
#endif
#include "sh.h"
#include "edit.h"
#undef CTRL
#define CTRL(x)  ((x) & 0x1F)      /* ASCII */
  ⟨ vi mode static variables 1022 ⟩
  ⟨ vi mode static functions 1023 ⟩
  ⟨ vi mode source 1028 ⟩
#endif      /* VI */
```

1022. **#define** MAXVICMD 3
#define SRCHLEN 40

⟨ vi mode static variables 1022 ⟩ ≡

```
static char undocbuf[LINE];
static struct edstate *save_edstate(struct edstate *old);
static void restore_edstate(struct edstate *old, struct edstate *new);
static void free_edstate(struct edstate *old);
static struct edstate ebuf;
static struct edstate undobuf ← {undocbuf, LINE, 0, 0, 0};
static struct edstate *es;      /* current editor state */
static struct edstate *undo;
static char ibuf[LINE];      /* input buffer */
static int first_insert;      /* set when starting in insert mode */
static int saved_inslen;      /* saved inslen for first insert */
static int inslen;      /* length of input buffer */
static int srchlen;      /* number of bytes in search pattern */
static char ybuf[LINE];      /* yank buffer */
static int yanklen;      /* length of yank buffer */
static int fsavecmd ← ' ';      /* last find command */
static int fsavech;      /* character to find */
static char lastcmd[MAXVICMD];      /* last non-move command */
static int lastac;      /* argcnt for lastcmd */
static int lastsearch ← ' ';      /* last search command */
static char srchpat[SRCHLEN];      /* last search pattern */
static int insert;      /* mode: INSERT, REPLACE, or 0 */
static int hnum;      /* position in history */
static int ohnum;      /* history line copied (after mod) */
static int hlast;      /* 1 past last position in history */
static int modified;      /* buffer has been "modified" */
static int state;
```

See also sections 1025, 1026, 1037, and 1112.

This code is used in section 1021.

1023. ⟨ vi mode static functions 1023 ⟩ ≡

```
static int vi_hook(int);
static void vi_reset(char *, size_t);
static int nextstate(int);
static int vi_insert(int);
static int vi_cmd(int, const char *);
static int domove(int, const char *, int);
static int redo_insert(int);
static void yank_range(int, int);
static int bracktype(int);
static void save_cbuf(void);
static void restore_cbuf(void);
static void edit_reset(char *, size_t);
static int putbuf(const char *, int, int);
static void del_range(int, int);
static int findch(int, int, int, int);
static int forwword(int);
static int backword(int);
static int endword(int);
static int Forwword(int);
static int Backword(int);
static int Endword(int);
static int grabhist(int, int);
static int grabsearch(int, int, int, char *);
static void do_clear_screen(void);
static void redraw_line(int, int);
static void refresh_line(int);
static int outofwin(void);
static void rewindow(void);
static int newcol(int, int);
static void display(char *, char *, int);
static void ed_mov_opt(int, char *);
static int expand_word(int);
static int complete_word(int, int);
static int print_expansions(struct edstate *);
static int char_len(int);
static void x_vi_zotc(int);
static void vi_pprompt(int);
static void vi_error(void);
static void vi_macro_reset(void);
static int x_vi_putbuf(const char *, size_t);
static int isu8cont(unsigned char);
```

This code is used in section 1021.

1024. vi mode macros. To be described (short: set `$_`⟨*char*⟩ to a vi command sequence).

⟨ Type definitions 17 ⟩ +≡

```
struct macro_state {
  unsigned char *p;      /* current position in buf */
  unsigned char *buf;    /* pointer to macro(s) being expanded */
  int len;     /* how much data in buffer */
};
```

1025. Information for keeping track of macros that are being expanded. The format of *buf* is the alias contents followed by `'\0'` followed by the name (letter) of the alias. The end of the buffer is marked by a double `'\0'`. The name of the alias is stored so recursive macros can be detected.

⟨ vi mode static variables 1022 ⟩ +≡
```
static struct macro_state macro;
```

1026. Key command flags. This was beautifully tabular in its original setting. I don't know how to reproduce it even half as nicely using CWEB.

```
#define C_ 0x1 /* a valid command that isn't a M_, E_ or U_ */
#define M_ 0x2 /* movement command (h, l, etc.) */
#define E_ 0x4 /* extended command (c, d, y) */
#define X_ 0x8 /* long command (@, f, F, t, T, etc.) */
#define U_ 0x10 /* an un-undoable command (that isn't a M_) */
#define B_ 0x20 /* bad command (^@) */
#define Z_ 0x40 /* repeat count defaults to 0 (not 1) */
#define S_ 0x80 /* search (/, ?) */
```

⟨ vi mode static variables 1022 ⟩ +≡
```
const unsigned char classify[128] ← {
  B_,      0,       0,       0,       0,       C_ | U_, C_ | Z_,  0,          /* °00_: ^@ .. ^G */
  M_,      C_ | Z_, 0,       0,       C_ | U_, 0,       C_,       0,          /* °01_: ^H .. ^O */
  C_,      0,       C_ | U_, 0,       0,       0,       C_,       0,          /* °02_: ^P .. ^W */
  C_,      0,       0,       C_ | Z_, 0,       0,       0,        0,          /* °03_: ^X .. ^_ */
  M_,      0,       0,       C_,      M_,      M_,      0,        0,          /* °04_: ␣ .. ' */
  0,       0,       C_,      C_,      M_,      C_,      0,        C_ | S_,    /* °05_: ( .. / */
  M_,      0,       0,       0,       0,       0,       0,        0,          /* °06_: 0 .. 7 */
  0,       0,       0,       M_,      0,       C_,      0,        C_ | S_,    /* °07_: 8 .. ? */
  C_ | X_, C_,      M_,      C_,      C_,      M_,      M_ | X_,  C_ | U_ | Z_, /* °10_: @ .. G */
  0,       C_,      0,       0,       0,       0,       C_ | U_,  0,          /* °11_: H .. O */
  C_,      0,       C_,      C_,      M_ | X_, C_,      0,        M_,         /* °12_: P .. W */
  C_,      C_ | U_, 0,       0,       C_ | Z_, 0,       M_,       C_ | Z_,    /* °13_: X .. _ */
  0,       C_,      M_,      E_,      E_,      M_,      M_ | X_,  C_ | Z_,    /* °14_: ` .. g */
  M_,      C_,      C_ | U_, C_ | U_, M_,      0,       C_ | U_,  0,          /* °15_: h .. o */
  C_,      0,       X_,      C_,      M_ | X_, C_ | U_, C_ | U_ | Z_, M_,    /* °16_: p .. w */
  C_,      E_ | U_, 0,       0,       M_ | Z_, 0,       C_,       0           /* °17_: x .. ^? */
};
```

1027. Interrogation.

```
#define is_bad(c) (classify[(c) & 0x7f] & B_)
#define is_cmd(c) (classify[(c) & 0x7f] & (M_ | E_ | C_ | U_))
#define is_move(c) (classify[(c) & 0x7f] & M_)
#define is_extend(c) (classify[(c) & 0x7f] & E_)
#define is_long(c) (classify[(c) & 0x7f] & X_)
#define is_undoable(c) (¬(classify[(c) & 0x7f] & U_))
#define is_srch(c) (classify[(c) & 0x7f] & S_)
#define is_zerocount(c) (classify[(c) & 0x7f] & Z_)
```

1028. ⟨ vi mode source 1028 ⟩ ≡

```
int x_vi(char *buf, size_t len)
{
  int c;
  vi_reset(buf, len > LINE ? LINE : len);
  vi_pprompt(1);
  x_flush( );
  while (1) {
    if (macro.p) {⟨ Look for the next macro character 1032 ⟩}
    else c ← x_getc( );
    if (c ≡ −1) break;
    if (state ≠ VLIT) {
      if (c ≡ edchars.intr ∨ c ≡ edchars.quit) {⟨ Pretend we got an interrupt 1033 ⟩}
      else if (c ≡ edchars.eof) {⟨ Receive a literal 'EOF' (^D) character 1034 ⟩}
    }
    if (vi_hook(c)) break;
    x_flush( );
  }
  x_putc('\r'); x_putc('\n'); x_flush( );
  if (c ≡ −1 ∨ len ≤ (size_t) es→linelen) return −1;
  if (es→cbuf ≠ buf) memmove(buf, es→cbuf, es→linelen);
  buf[es→linelen++] ← '\n';
  return es→linelen;
}
```

See also sections 1029, 1030, 1031, 1035, 1038, 1039, 1040, 1041, 1042, 1043, 1045, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1058, 1059, 1060, 1061, 1062, 1063, 1064, 1076, 1077, 1078, 1088, 1100, 1101, 1110, 1118, 1119, 1124, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1146, and 1147.

This code is used in section 1021.

1029. ⟨ vi mode source 1028 ⟩ +≡

```
static void vi_reset(char *buf, size_t len)
{
  state ← VNORMAL;
  ohnum ← hnum ← hlast ← histnum(−1) + 1;
  insert ← INSERT;
  saved_inslen ← inslen;
  first_insert ← 1;
  inslen ← 0;
  modified ← 1;
  vi_macro_reset( );
  edit_reset(buf, len);
}
```

1030. ⟨ vi mode source 1028 ⟩ +≡

```
static void vi_macro_reset(void)
{
  if (macro.p) {
    afree(macro.buf, APERM);
    memset((char *) &macro, 0, sizeof (macro));
  }
}
```

1031. ⟨ vi mode source 1028 ⟩ +≡

```
static void edit_reset(char *buf, size_t len)
{
  const char *p;
  es ← &ebuf;
  es→cbuf ← buf;
  es→cbufsize ← len;
  undo ← &undobuf;
  undo→cbufsize ← len;
  es→linelen ← undo→linelen ← 0;
  es→cursor ← undo→cursor ← 0;
  es→winleft ← undo→winleft ← 0;
  cur_col ← pwidth ← promptlen(prompt, &p);
  prompt_skip ← p − prompt;
  if (pwidth > x_cols − 3 − MIN_EDIT_SPACE) {
    cur_col ← x_cols − 3 − MIN_EDIT_SPACE;
    prompt_trunc ← pwidth − cur_col;
    pwidth −= prompt_trunc;
  }
  else prompt_trunc ← 0;
  if (¬wbuf_len ∨ wbuf_len ≠ x_cols − 3) {
    wbuf_len ← x_cols − 3;
    wbuf[0] ← aresize(wbuf[0], wbuf_len, APERM);
    wbuf[1] ← aresize(wbuf[1], wbuf_len, APERM);
  }
  (void) memset(wbuf[0], '␣', wbuf_len);
  (void) memset(wbuf[1], '␣', wbuf_len);
  winwidth ← x_cols − pwidth − 3;
  win ← 0;
  morec ← '␣';
  holdlen ← 0;
}
```

1032. ⟨ Look for the next macro character 1032 ⟩ ≡

```
c ← (unsigned char) *macro.p++;
if (¬c) {      /* end of current macro */
  if (*macro.p++) continue;      /* more macros left to finish */
  vi_macro_reset();
  c ← x_getc();
}
```

This code is used in section 1028.

1033. ⟨ Pretend we got an interrupt 1033 ⟩ ≡

```
x_vi_zotc(c);
x_flush();
trapsig(c ≡ edchars.intr ? SIGINT : SIGQUIT);
x_mode(false);
unwind (LSHELL);
```

This code is used in section 1028.

1034. ⟨ Receive a literal 'EOF' (^D) character 1034 ⟩ ≡

```
if (es¬linelen ≡ 0) {
  x_vi_zotc(edchars.eof);
  c ← −1;
  break;
}
continue;
```

This code is used in section 1028.

1035. ⟨ vi mode source 1028 ⟩ +≡

```
static void vi_error(void)
{     /* Beem out of any macros as soon as an error occurs */
  vi_macro_reset( );
  x_putc(BEL);
  x_flush( );
}
```

1036. vi Edit Buffer.

```
⟨ Type definitions 17 ⟩ +≡
  struct edstate {
    char *cbuf;      /* main buffer to build the command line */
    int cbufsize;    /* number of bytes allocated for cbuf */
    int linelen;     /* current number of bytes in cbuf */
    int winleft;     /* first byte # in cbuf to be displayed */
    int cursor;      /* byte # in cbuf having the cursor */
  };
```

1037.

```
⟨ vi mode static variables 1022 ⟩ +≡
  static int cur_col;      /* current display column */
  static int pwidth;       /* display columns needed for the prompt */
  static int prompt_trunc;     /* how much of the prompt to truncate */
  static int prompt_skip;      /* how much of the prompt to skip */
  static int winwidth;     /* available column positions */
  static char *wbuf[2];      /* current & previous window buffer */
  static int wbuf_len;     /* length of window buffers (x_cols − 3) */
  static int win;      /* number of window buffer in use */
  static char morec;       /* more character(s) right of the window */
  static char holdbuf[LINE];       /* place to hold the last edit buffer */
  static int holdlen;      /* length of holdbuf */
```

1038.

```
⟨ vi mode source 1028 ⟩ +≡
  static void save_cbuf(void)
  {
    memmove(holdbuf, es→cbuf, es→linelen);
    holdlen ← es→linelen;
    holdbuf[holdlen] ← '\0';
  }
```

1039.

```
⟨ vi mode source 1028 ⟩ +≡
  static void restore_cbuf(void)
  {
    es→cursor ← 0;
    es→linelen ← holdlen;
    memmove(es→cbuf, holdbuf, holdlen);
  }
```

1040. Return a new edstate. Used by expansion/completion.

```
⟨ vi mode source 1028 ⟩ +≡
  static struct edstate *save_edstate(struct edstate *old)
  {
    struct edstate *new;
    new ← alloc(sizeof(struct edstate), APERM);
    new→cbuf ← alloc(old→cbufsize, APERM);
    memcpy(new→cbuf, old→cbuf, old→linelen);
    new→cbufsize ← old→cbufsize;
    new→linelen ← old→linelen;
    new→cursor ← old→cursor;
    new→winleft ← old→winleft;
    return new;
  }
```

1041.

```
⟨ vi mode source 1028 ⟩ +≡
  static void restore_edstate(struct edstate *new, struct edstate *old)
  {
    memcpy(new→cbuf, old→cbuf, old→linelen);
    new→linelen ← old→linelen;
    new→cursor ← old→cursor;
    new→winleft ← old→winleft;
    free_edstate(old);
  }
```

1042.

```
⟨ vi mode source 1028 ⟩ +≡
  static void free_edstate(struct edstate *old)
  {
    afree(old→cbuf, APERM);
    afree(old, APERM);
  }
```

1043.

```
⟨ vi mode source 1028 ⟩ +≡
  static void del_range(int a, int b)
  {
    if (es→linelen ≠ b) memmove(&es→cbuf[a], &es→cbuf[b], es→linelen − b);
    es→linelen −= b − a;
  }
```

1044. vi Display. Ctrl-L: redraw the screen.

```
⟨ Perform a vi mode command 1044 ⟩ ≡
case CTRL('l'): do_clear_screen();
  break;
```

See also sections 1046, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1102, 1104, 1105, 1106, 1107, 1108, 1109, 1113, 1114, 1115, 1116, 1117, 1140, 1141, 1142, 1143, 1144, 1149, 1153, 1154, 1155, 1156, and 1157.

This code is used in section 1077.

1045.

```
⟨ vi mode source 1028 ⟩ +≡
  static void do_clear_screen(void)
  {
    int neednl ← 1;
#ifndef SMALL
    if (cur_term ≠ Λ ∧ clear_screen ≠ Λ) {
      if (tputs(clear_screen, 1, x_putc) ≠ ERR) neednl ← 0;
    }
#endif
    redraw_line(neednl, ¬neednl);      /* Only print the full prompt if we cleared the screen. */
  }
```

1046. Ctrl-R: redraw the current line.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case CTRL('r'): redraw_line(1, 0);
  break;
```

1047.

```
⟨ vi mode source 1028 ⟩ +≡
  static void redraw_line(int neednl, int full)
  {
    (void) memset(wbuf[win], '␣', wbuf_len);
    if (neednl) {
      x_putc('\r');
      x_putc('\n');
    }
    vi_pprompt(full);
    cur_col ← pwidth;
    morec ← '␣';
  }
```

1048.

```
⟨ vi mode source 1028 ⟩ +≡
  static void refresh_line(int leftside)
  {
    if (outofwin()) rewindow();
    display(wbuf[1 − win], wbuf[win], leftside);
    win ← 1 − win;
  }
```

1049. ⟨ vi mode source 1028 ⟩ +≡

```
static int outofwin(void)
{
  int cur, col;
  if (es→cursor < es→winleft) return 1;
  col ← 0;
  cur ← es→winleft;
  while (cur < es→cursor) col ← newcol((unsigned char) es→cbuf[cur++], col);
  if (col ≥ winwidth) return 1;
  return 0;
}
```

1050. ⟨ vi mode source 1028 ⟩ +≡

```
static void rewindow(void)
{
  int tcur, tcol;
  int holdcur1, holdcol1;
  int holdcur2, holdcol2;
  holdcur1 ← holdcur2 ← tcur ← 0;
  holdcol1 ← holdcol2 ← tcol ← 0;
  while (tcur < es→cursor) {
    if (tcol − holdcol2 > winwidth/2) {
      holdcur1 ← holdcur2;
      holdcol1 ← holdcol2;
      holdcur2 ← tcur;
      holdcol2 ← tcol;
    }
    tcol ← newcol((unsigned char) es→cbuf[tcur++], tcol);
  }
  while (tcol − holdcol1 > winwidth/2)
    holdcol1 ← newcol((unsigned char) es→cbuf[holdcur1++], holdcol1);
  es→winleft ← holdcur1;
}
```

1051. Printing the byte *ch* at display column *col* moves to which column?

⟨ vi mode source 1028 ⟩ +≡

```
static int newcol(int ch, int col)
{
  if (ch ≡ '\t') return (col | 7) + 1;
  if (isu8cont(ch)) return col;
  return col + char_len(ch);
}
```

1052. The number of bytes needed to encode byte *c*. Control bytes get "M-" or "^" prepended. This function does not handle tabs.

```
⟨ vi mode source 1028 ⟩ +≡
  static int char_len(int c)
  {
    int len ← 1;
    if ((c & 0x80) ∧ Flag(FVISHOW8)) {
      len += 2;
      c &= 0x7f;
    }
    if (c < ' ' ∨ c ≡ 0x7f) len++;
    return len;
  }
```

1053. Display *wb1* assuming that *wb2* is currently displayed.

```
⟨ vi mode source 1028 ⟩ +≡
  static void display(char *wb1, char *wb2, int leftside)
  {
    char *twb1;      /* pointer into the buffer to display */
    char *twb2;      /* pointer into the previous display buffer */
    static int lastb ← −1;      /* last byte # written from wb1, if UTF-8 */
    int cur;      /* byte # in the main command line buffer */
    int col;      /* display column loop variable */
    int ncol;      /* display column of the cursor */
    int cnt;      /* remaining display columns to fill */
    int moreright;
    char mc;      /* new "more character" at the right of window */
    unsigned char ch;
    ⟨ Fill the current display buffer 1054 ⟩
    ⟨ Pad the current display buffer to the right margin 1055 ⟩
    ⟨ Update the terminal from wb1 1056 ⟩
    ⟨ Update the "more character(s)" 1057 ⟩
    if (cur_col ≠ ncol) {      /* Move the cursor to its new position. */
      ed_mov_opt(ncol, wb1);
      lastb ← −1;
    }
  }
```

1054. Fill the current display buffer with data from *cbuf*. In this first loop *col* does not include the prompt.

```
⟨ Fill the current display buffer 1054 ⟩ ≡
  ncol ← col ← 0;
  cur ← es→winleft;
  moreright ← 0;
  twb1 ← wb1;
  while (col < winwidth ∧ cur < es→linelen) {
    if (cur ≡ es→cursor ∧ leftside) ncol ← col + pwidth;
    if ((ch ← es→cbuf[cur]) ≡ '\t') {
      do {
        *twb1++ ← '␣';
      } while (++col < winwidth ∧ (col & 7) ≠ 0);
    }
    else {
      if ((ch & 0x80) ∧ Flag(FVISHOW8)) {
        *twb1++ ← 'M';
        if (++col < winwidth) {
          *twb1++ ← '-';
          col++;
        }
        ch &= 0x7f;
      }
      if (col < winwidth) {
        if (ch < '␣' ∨ ch ≡ 0x7f) {
          *twb1++ ← '^';
          if (++col < winwidth) {
            *twb1++ ← ch ⊕ '@';
            col++;
          }
        }
        else {
          *twb1++ ← ch;
          if (¬isu8cont(ch)) col++;
        }
      }
    }
    if (cur ≡ es→cursor ∧ ¬leftside) ncol ← col + pwidth − 1;
    cur++;
  }
  if (cur ≡ es→cursor) ncol ← col + pwidth;
```

This code is used in section 1053.

1055. ⟨ Pad the current display buffer to the right margin 1055 ⟩ ≡

```
if (col < winwidth) {
  while (col < winwidth) {
    *twb1++ ← '␣';
    col++;
  }
}
else moreright++;
*twb1 ← '␣';
```

This code is used in section 1053.

1056. Update the terminal display with data from *wb1*. In this final loop *col* includes the prompt.

⟨ Update the terminal from *wb1* 1056 ⟩ ≡

```
col ← pwidth;
cnt ← winwidth;
for (twb1 ← wb1, twb2 ← wb2; cnt; twb1++, twb2++) {
  if (*twb1 ≠ *twb2) {      /* When a byte changes in the middle of a UTF-8 character, back up to the
        start byte, unless the previous byte was the last one written. */
    if (col > 0 ∧ isu8cont(*twb1)) {
      col--;
      if (lastb ≥ 0 ∧ twb1 ≡ wb1 + lastb + 1) cur_col ← col;
      else
        while (twb1 > wb1 ∧ isu8cont(*twb1)) {
          twb1--;
          twb2--;
        }
    }
    if (cur_col ≠ col) ed_mov_opt(col, wb1);
        /* Always write complete characters, and advance all pointers accordingly. */
    x_putc(*twb1);
    while (isu8cont(twb1[1])) {
      x_putc(*++twb1);
      twb2++;
    }
    lastb ← *twb1 & 0x80 ? twb1 − wb1 : −1;
    cur_col++;
  }
  else if (isu8cont(*twb1)) continue;      /* For changed continuation bytes, we backed up. For
        unchanged ones, we jumped to the next byte. So, getting here, we had a real column. */
  col++;
  cnt--;
}
```

This code is used in section 1053.

1057. ⟨ Update the "more character(s)" 1057 ⟩ ≡

```
if (es→winleft > 0 ∧ moreright) mc ← '+';      /* POSIX says to use "*" for this but that is a globbing
        character and may confuse people; "+" is more innocuous */
else if (es→winleft > 0) mc ← '<';
else if (moreright) mc ← '>';
else mc ← '␣';
if (mc ≠ morec) {
  ed_mov_opt(pwidth + winwidth + 1, wb1);
  x_putc(mc);
  cur_col++;
  morec ← mc;
  lastb ← −1;
}
```

This code is used in section 1053.

1058. Move the display cursor to display column number *col*.

⟨ vi mode source 1028 ⟩ +≡

```
static void ed_mov_opt(int col, char *wb)
{
  int ci;
  if (cur_col ≡ col) return;      /* The cursor is already at the right place. */
  if (cur_col > col) {      /* The cursor is too far right. */
    if (cur_col > 2 * col + 1) {      /* Much too far right, redraw from scratch. */
      x_putc('\r');
      vi_pprompt(0);
      cur_col ← pwidth;
    }
    else {      /* Slightly too far right, back up. */
      do {
        x_putc('\b');
      } while (−−cur_col > col);
      return;
    }
  }
  for (ci ← pwidth; ci < col ∨ isu8cont(*wb); ci ← newcol((unsigned char) *wb++, ci))
    if (ci > cur_col ∨ (ci ≡ cur_col ∧ ¬isu8cont(*wb))) x_putc(*wb);      /* Advance the cursor. */
  cur_col ← ci;
}
```

1059. ⟨ vi mode source 1028 ⟩ +≡

```
static int print_expansions(struct edstate *e)
{
  int nwords;
  int start, end;
  char **words;
  int is_command;
  nwords ← x_cf_glob(XCF_COMMAND_FILE | XCF_FULLPATH, e→cbuf, e→linelen, e→cursor, &start, &end,
      &words, &is_command);
  if (nwords ≡ 0) {
    vi_error( );
    return −1;
  }
  x_print_expansions(nwords, words, is_command);
  x_free_words(nwords, words);
  redraw_line(0, 0);
  return 0;
}
```

1060. ⟨ vi mode source 1028 ⟩ +≡

```
static void vi_pprompt(int full)
{
  pprompt(prompt + (full ? 0 : prompt_skip), prompt_trunc);
}
```

1061. Emacs and vi modes define this function with the same name differently.

⟨ vi mode source 1028 ⟩ +≡

```
static int isu8cont(unsigned char c)
{
  return ¬Flag(FVISHOW8) ∧ (c & (0x80 | 0x40)) ≡ 0x80;
}
```

1062. Similar to *x_zotc* but no tab weirdness.

⟨ vi mode source 1028 ⟩ +≡

```
static void x_vi_zotc(int c)
{
  if (Flag(FVISHOW8) ∧ (c & 0x80)) {
    x_puts("M-");
    c &= 0x7f;
  }
  if (c < ' ' ∨ c ≡ 0x7f) {
    x_putc('^');
    c ⊕= '@';
  }
  x_putc(c);
}
```

1063. vi Modes.

```
#define VNORMAL 0 /* command, insert or replace mode */
#define VARG1 1 /* digit prefix (first, eg. 5l) */
#define VEXTCMD 2 /* cmd + movement (eg. cl) */
#define VARG2 3 /* digit prefix (second, eg. 2c3l) */
#define VXCH 4 /* f, F, t, T, @ */
#define VFAIL 5 /* bad command */
#define VCMD 6 /* single char command (eg. X) */
#define VREDO 7 /* . */
#define VLIT 8 /* ^V */
#define VSEARCH 9 /* /, ? */
#define INSERT 1 /* insert: sub-type of VNORMAL */
#define REPLACE 2 /* insert: sub-type of VNORMAL */
```

⟨ vi mode source 1028 ⟩ +≡

```
static int vi_hook(int ch)
{
  static char curcmd[MAXVICMD], locpat[SRCHLEN];
  static int cmdlen, argc1, argc2;
  switch (state) {
  case VNORMAL:
    if (insert ≠ 0) {⟨ vi VNORMAL insert or replace state 1065 ⟩}
    else {⟨ vi VNORMAL command state 1066 ⟩}
    break;
  case VLIT: ⟨ vi mode VLIT state 1067 ⟩
  case VARG1: ⟨ vi mode VARG1 state 1068 ⟩
  case VEXTCMD: ⟨ vi mode VEXTCMD state 1071 ⟩
  case VARG2: ⟨ vi mode VARG2 state 1069 ⟩
  case VXCH: ⟨ vi mode VXCH state 1072 ⟩
  case VSEARCH: ⟨ vi mode VSEARCH state 1145 ⟩
  }
  switch (state) {
  case VCMD: ⟨ vi mode VCMD state 1074 ⟩
  case VREDO: ⟨ vi mode VREDO state 1075 ⟩
  case VFAIL: ⟨ vi mode VFAIL state 1073 ⟩
  }
  return 0;
}
```

1064. Primarily from VNORMAL state but also others.

⟨ vi mode source 1028 ⟩ +≡

```
static int nextstate(int ch)
{
  if (is_extend(ch)) return VEXTCMD;
  else if (is_srch(ch)) return VSEARCH;
  else if (is_long(ch)) return VXCH;
  else if (ch ≡ '.') return VREDO;
  else if (is_cmd(ch)) return VCMD;
  else return VFAIL;
}
```

1065. If "^V" is pressed the next keypress is inserted literally; draw a "^" glyph but do not advance the cursor position.

```
⟨ vi VNORMAL insert or replace state 1065 ⟩ ≡
  if (ch ≡ CTRL('v')) {
    state ← VLIT;
    ch ← '^';
  }
  switch (vi_insert(ch)) {      /* vi_insert: attempt to insert the character */
  case −1: vi_error( );
    state ← VNORMAL;
    break;
  case 0:
    if (state ≡ VLIT) { es→cursor −−; refresh_line(0); }
    else refresh_line(insert ≠ 0);
    break;
  case 1: return 1;
  }
```

This code is used in section 1063.

1066.

```
⟨ vi VNORMAL command state 1066 ⟩ ≡
  if (ch ≡ '\r' ∨ ch ≡ '\n') return 1;
  cmdlen ← 0;
  argc1 ← 0;
  if (ch ≥ '1' ∧ ch ≤ '9') {      /* begin gathering a repeat count */
    argc1 ← ch − '0';
    state ← VARG1;
  }
  else {
    curcmd[cmdlen++] ← ch;
    state ← nextstate(ch);
    if (state ≡ VSEARCH) {      /* begin searching command history */
      save_cbuf( );
      es→cursor ← 0;
      es→linelen ← 0;
      if (ch ≡ '/') {
        if (putbuf("/", 1, 0) ≠ 0) return −1;
      }
      else if (putbuf("?", 1, 0) ≠ 0) return −1;
      refresh_line(0);
    }
  }
```

This code is used in section 1063.

1067. The only bad character is `'\0'`, everything else is appended literally and we return to `VNORMAL`.

```
⟨ vi mode VLIT state 1067 ⟩ ≡
  if (is_bad(ch)) {      /* ie. ^@ */
    del_range(es→cursor, es→cursor + 1);
    vi_error( );
  }
  else es→cbuf[es→cursor++] ← ch;
  refresh_line(1);
  state ← VNORMAL;
  break;
```

This code is used in section 1063.

1068. Continue collecting digits for a repeat count or switch state to expect the command.

```
⟨ vi mode VARG1 state 1068 ⟩ ≡
  if (isdigit(ch)) argc1 ← argc1 * 10 + ch − '0';
  else {
    curcmd[cmdlen++] ← ch;
    state ← nextstate(ch);
  }
  break;
```

This code is used in section 1063.

1069. Commands which require an option may also have a repeat count, saved in *argc2*. The subsequent set of states is more limited after the second argument.

```
⟨ vi mode VARG2 state 1069 ⟩ ≡
  if (isdigit(ch)) argc2 ← argc2 * 10 + ch − '0';
  else {
    if (argc1 ≡ 0) argc1 ← argc2;
    else argc1 *= argc2;
    curcmd[cmdlen++] ← ch;
    ⟨ Expect a command in vi mode 1070 ⟩
  }
  break;
```

This code is used in section 1063.

1070.

```
⟨ Expect a command in vi mode 1070 ⟩ ≡
  if (ch ≡ curcmd[0]) state ← VCMD;
  else if (is_move(ch)) state ← nextstate(ch);
  else state ← VFAIL;
```

This code is used in sections 1069 and 1071.

1071. The command needs an option, possibly repeated. The state change is of course identical to that in the VARG2 state.

```
⟨ vi mode VEXTCMD state 1071 ⟩ ≡
  argc2 ← 0;
  if (ch ≥ '1' ∧ ch ≤ '9') {
    argc2 ← ch − '0';
    state ← VARG2;
    return 0;
  }
  else {
    curcmd[cmdlen++] ← ch;
    ⟨ Expect a command in vi mode 1070 ⟩
  }
  break;
```

This code is used in section 1063.

1072. The command needs a character option, such as fF/tT.

```
⟨ vi mode VXCH state 1072 ⟩ ≡
  if (ch ≡ CTRL('[')) state ← VNORMAL;
  else {
    curcmd[cmdlen++] ← ch;
    state ← VCMD;
  }
  break;
```

This code is used in section 1063.

1073.

```
⟨ vi mode VFAIL state 1073 ⟩ ≡
  state ← VNORMAL;
  vi_error( );
  break;
```

This code is used in section 1063.

1074. When a complete command sequence is received the second **switch** in *vi_hook* performs it. The performance routine is *vi_cmd*; this is checking its result.

```
⟨ vi mode VCMD state 1074 ⟩ ≡
  state ← VNORMAL;
  switch (vi_cmd(argc1, curcmd)) {
  case −1: vi_error( );
    refresh_line(0);
    break;
  case 0:
    if (insert ≠ 0) inslen ← 0;
    refresh_line(insert ≠ 0);
    break;
  case 1: refresh_line(0); return 1;
  case 2: return 1;      /* back from a 'v' command—don't redraw the screen */
  }
  break;
```

This code is used in section 1063.

1075. Alternatively repeat the previous command, possibly a different number of times.

```
⟨ vi mode VREDO state 1075 ⟩ ≡
  state ← VNORMAL;
  if (argc1 ≠ 0) lastac ← argc1;
  switch (vi_cmd(lastac, lastcmd)) {
  case −1: vi_error( );
    refresh_line(0);
    break;
  case 0:
    if (insert ≠ 0) {
      if (lastcmd[0] ≡ 's' ∨ lastcmd[0] ≡ 'c' ∨ lastcmd[0] ≡ 'C') {
        if (redo_insert(1) ≠ 0) vi_error( );
      }
      else {
        if (redo_insert(lastac) ≠ 0) vi_error( );
      }
    }
    refresh_line(0);
    break;
  case 1: refresh_line(0);
    return 1;
  case 2: return 1;      /* back from a 'v' command—don't redraw the screen */
  }
  break;
```

This code is used in section 1063.

1076.

```
⟨ vi mode source 1028 ⟩ +≡
  static int redo_insert(int count)
  {
    while (count−− > 0)
      if (putbuf(ibuf, inslen, insert ≡ REPLACE) ≠ 0) return −1;
    if (es→cursor > 0)
      while (isu8cont(es→cbuf[−−es→cursor])) continue;
    insert ← 0;
    return 0;
  }
```

1077. vi Commands. Not much more complicated than a single switch statement with a case per character.

```
⟨ vi mode source 1028 ⟩ +≡
  static int vi_cmd(int argcnt, const char *cmd)
  {
    int ncursor;
    int cur, c1, c2, c3 ← 0;
    int any;
    struct edstate *t;
    if (argcnt ≡ 0 ∧ ¬is_zerocount(*cmd)) argcnt ← 1;
    if (is_move(*cmd)) {
      if ((cur ← domove(argcnt, cmd, 0)) ≥ 0) {
        if (cur ≡ es¬linelen ∧ cur ≠ 0)
          while (isu8cont(es¬cbuf[--cur])) continue;
        es¬cursor ← cur;
      }
      else return −1;
    }
    else {      /* Don't save state in the middle of a macro. */
      if (is_undoable(*cmd) ∧ ¬macro.p) {
        undo¬winleft ← es¬winleft;
        memmove(undo¬cbuf, es¬cbuf, es¬linelen);
        undo¬linelen ← es¬linelen;
        undo¬cursor ← es¬cursor;
        lastac ← argcnt;
        memmove(lastcmd, cmd, MAXVICMD);
      }
      switch (*cmd) {⟨ Perform a vi mode command 1044 ⟩}
      if (insert ≡ 0 ∧ es¬cursor ≥ es¬linelen)
        while (es¬cursor > 0)
          if (¬isu8cont(es¬cbuf[--es¬cursor])) break;
    }
    return 0;
  }
```

1078. Moving, either directly or as part of another command, is performed by *domove*.

```
⟨ vi mode source 1028 ⟩ +≡
  static int domove(int argcnt, const char *cmd, int sub)
  {
    int bcount, i ← 0, t;
    int ncursor ← 0;
    switch (*cmd) {
      ⟨ Perform a vi movement command 1121 ⟩
    default: return −1;
    }
    return ncursor;
  }
```

1079. vi Insert Mode. a: enter insert mode with the cursor after the current character.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'a': modified ← 1;
  hnum ← hlast;
  if (es¬linelen ≠ 0)
    while (isu8cont(es¬cbuf[++es¬cursor])) continue;
  insert ← INSERT;
  break;
```

1080. A: enter insert mode the cursor after the last character in the current line.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'A': modified ← 1;
  hnum ← hlast;
  del_range(0, 0);
  es¬cursor ← es¬linelen;
  insert ← INSERT;
  break;
```

1081. i (eye): enter insert mode at the current cursor position.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'i': modified ← 1;
  hnum ← hlast;
  insert ← INSERT;
  break;
```

1082. I (eye): enter insert mode at the beginning of the current line.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'I': modified ← 1;
  hnum ← hlast;
  es¬cursor ← domove(1, "^", 1);
  insert ← INSERT;
  break;
```

1083. r: replace the character under the cursor but do not enter insert mode (like `s` does).

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'r':
  if (es¬linelen ≡ 0) return −1;
  modified ← 1;
  hnum ← hlast;
  if (cmd[1] ≡ 0) vi_error( );
  else {
    c1 ← 0;
    for (cur ← es¬cursor; cur < es¬linelen; cur ++) {
      if (¬isu8cont(es¬cbuf[cur])) c1 ++;
      if (c1 > argcnt) break;
    }
    if (argcnt > c1) return −1;
    del_range(es¬cursor, cur);
    while (argcnt −− > 0) putbuf(&cmd[1], 1, 0);
    while (es¬cursor > 0)
      if (¬isu8cont(es¬cbuf[−− es¬cursor])) break;
    es¬cbuf[es¬linelen] ← '\0';
  }
  break;
```

1084. R: replace the rest of the line. Unlike changing the rest of the line (`C`/`c$`) `R` enters "replace" mode which is like insert mode except that if characters already exist under the cursor they are replaced rather than being shifted right, and deleting inserted characters restores the overwritten ones. The replaced text is also not copied to the yank buffer.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'R': modified ← 1;
  hnum ← hlast;
  insert ← REPLACE;
  break;
```

1085. s: delete one or more characters and enter insert mode.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 's':
  if (es¬linelen ≡ 0) return −1;
  modified ← 1;
  hnum ← hlast;
  for (cur ← es¬cursor; cur < es¬linelen; cur ++)
    if (¬isu8cont(es¬cbuf[cur]))
      if (argcnt −− ≡ 0) break;
  del_range(es¬cursor, cur);
  insert ← INSERT;
  break;
```

1086. S: enter insert mode replacing the whole current line.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'S': es→cursor ← domove(1, "^", 1);
  del_range(es→cursor, es→linelen);
  modified ← 1;
  hnum ← hlast;
  insert ← INSERT;
  break;
```

1087. /* If any chars are entered before escape, trash the saved insert * buffer (if user inserts & deletes char, *ibuf* gets trashed and * we don't want to use it) */

1088.

```
⟨ vi mode source 1028 ⟩ +≡
  static int vi_insert(int ch)
  {
    int tcursor;
    if (ch ≡ edchars.erase ∨ ch ≡ CTRL('h')) {⟨ "Insert" a backspace character and return 1089 ⟩}
    if (ch ≡ edchars.kill) {⟨ Cancel the current line and return 1090 ⟩}
    if (ch ≡ edchars.werase) {⟨ Erase the current word and return 1091 ⟩}
    if (first_insert ∧ ch ≠ CTRL('[')) saved_inslen ← 0;
    switch (ch) {
    case '\0': return −1;
    case '\r': case '\n': return 1;
    case CTRL('['): ⟨ Leave vi insert mode and return 1092 ⟩
      ⟨ vi insert mode non-standard commands 1093 ⟩ /* case CTRL('i') will FALLTHROUGH */
    default: ⟨ Insert a character in vi mode 1099 ⟩
    }
    return 0;
  }
```

1089.

```
⟨ "Insert" a backspace character and return 1089 ⟩ ≡
  if (insert ≡ REPLACE) {
    if (es→cursor ≡ undo→cursor) {      /* cannot delete past where replacing began */
      vi_error( );
      return 0;
    }
  }
  else {
    if (es→cursor ≡ 0) { return 0; }      /* no annoying "x_putc(BEL)" bell here */
  }
  tcursor ← es→cursor − 1;
  while (tcursor > 0 ∧ isu8cont(es→cbuf[tcursor])) tcursor−−;
  if (insert ≡ INSERT) memmove(es→cbuf + tcursor, es→cbuf + es→cursor, es→linelen − es→cursor);
  if (insert ≡ REPLACE ∧ es→cursor < undo→linelen)
    memcpy(es→cbuf + tcursor, undo→cbuf + tcursor, es→cursor − tcursor);
  else es→linelen −= es→cursor − tcursor;
  if (inslen < es→cursor − tcursor) inslen ← 0;
  else inslen −= es→cursor − tcursor;
  es→cursor ← tcursor;
  expanded ← NONE;
  return 0;
```

This code is used in section 1088.

1090. ⟨ Cancel the current line and **return** 1090 ⟩ ≡

```
if (es→cursor ≠ 0) {
  inslen ← 0;
  memmove(es→cbuf, &es→cbuf[es→cursor], es→linelen − es→cursor);
  es→linelen −= es→cursor;
  es→cursor ← 0;
}
expanded ← NONE;
return 0;
```

This code is used in section 1088.

1091. ⟨ Erase the current word and **return** 1091 ⟩ ≡

```
if (es→cursor ≠ 0) {
  tcursor ← backword(1);
  memmove(&es→cbuf[tcursor], &es→cbuf[es→cursor], es→linelen − es→cursor);
  es→linelen −= es→cursor − tcursor;
  if (inslen < es→cursor − tcursor) inslen ← 0;
  else inslen −= es→cursor − tcursor;
  es→cursor ← tcursor;
}
expanded ← NONE;
return 0;
```

This code is used in section 1088.

1092. ⟨ Leave vi insert mode and **return** 1092 ⟩ ≡

```
expanded ← NONE;
if (first_insert) {
  first_insert ← 0;
  if (inslen ≡ 0) {
    inslen ← saved_inslen;
    return redo_insert(0);
  }
  lastcmd[0] ← 's';
  lastac ← 1;
}
if (lastcmd[0] ≡ 's' ∨ lastcmd[0] ≡ 'c' ∨ lastcmd[0] ≡ 'C') return redo_insert(0);
else return redo_insert(lastac − 1);
```

This code is used in section 1088.

1093. ⟨ vi insert mode non-standard commands 1093 ⟩ ≡

```
case CTRL('x'): expand_word(0);
  break;
```

See also sections 1094, 1095, 1096, 1097, and 1098.

This code is used in section 1088.

1094. ⟨ vi insert mode non-standard commands 1093 ⟩ +≡

```
case CTRL('f'): complete_word(0, 0);
  break;
```

1095. ⟨ vi insert mode non-standard commands 1093 ⟩ +≡

```
case CTRL('e'): print_expansions(es);
  break;
```

1096. ⟨ vi insert mode non-standard commands 1093 ⟩ +≡
```
case CTRL('l'): do_clear_screen();
  break;
```

1097. ⟨ vi insert mode non-standard commands 1093 ⟩ +≡
```
case CTRL('r'): redraw_line(1,0);
  break;
```

1098. This must be the last non-standard insert case so that it can fall through to the default insertion routine if FVITABCOMPLETE is disabled.

⟨ vi insert mode non-standard commands 1093 ⟩ +≡
```
case CTRL('i'):
  if (Flag(FVITABCOMPLETE)) {
    complete_word(0,0);
    break;
  }
```

1099. ⟨ Insert a character in vi mode 1099 ⟩ ≡
```
  if (es→linelen ≥ es→cbufsize − 1) return −1;
  ibuf[inslen++] ← ch;
  if (insert ≡ INSERT) {
    memmove(&es→cbuf[es→cursor + 1], &es→cbuf[es→cursor], es→linelen − es→cursor);
    es→linelen++;
  }
  es→cbuf[es→cursor++] ← ch;
  if (insert ≡ REPLACE ∧ es→cursor > es→linelen) es→linelen++;
  expanded ← NONE;
```
This code is used in section 1088.

1100. This is similar to the routine to insert a buffer.

⟨ vi mode source 1028 ⟩ +≡
```
  static int putbuf(const char *buf, int len, int repl)
  {
    if (len ≡ 0) return 0;
    if (repl) {
      if (es→cursor + len ≥ es→cbufsize) return −1;
      if (es→cursor + len > es→linelen) es→linelen ← es→cursor + len;
    }
    else {
      if (es→linelen + len ≥ es→cbufsize) return −1;
      memmove(&es→cbuf[es→cursor + len], &es→cbuf[es→cursor], es→linelen − es→cursor);
      es→linelen += len;
    }
    memmove(&es→cbuf[es→cursor], buf, len);
    es→cursor += len;
    return 0;
  }
```

1101. This is used for calling *x_escape* in *complete_word*.

```
⟨ vi mode source 1028 ⟩ +≡
  static int x_vi_putbuf(const char *s, size_t len)
  {
    return putbuf(s, len, 0);
  }
```

1102. vi Yank Buffer. Change (c), delete (d) or copy (y/Y)[1].

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'Y': cmd ← "y$";      /* FALLTHROUGH */
case 'c': case 'd': case 'y':
  if (*cmd ≡ cmd[1]) {      /* (ie. cmd[0] ≡ cmd[1]) copy the whole line */
    c1 ← *cmd ≡ 'c' ? domove(1, "^", 1) : 0;
    c2 ← es→linelen;
  }
  else if (¬is_move(cmd[1])) return −1;
  else {⟨ Calculate the range to copy from the buffer 1103 ⟩}
  if (*cmd ≠ 'c' ∧ c1 ≠ c2) yank_range(c1, c2);
  if (*cmd ≠ 'y') {
    del_range(c1, c2);
    es→cursor ← c1;
  }
  if (*cmd ≡ 'c') {
    modified ← 1;
    hnum ← hlast;
    insert ← INSERT;
  }
  break;
```

1103. ⟨ Calculate the range to copy from the buffer 1103 ⟩ ≡

```
if ((ncursor ← domove(argcnt, &cmd[1], 1)) < 0) return −1;
if (*cmd ≡ 'c' ∧ (cmd[1] ≡ 'w' ∨ cmd[1] ≡ 'W') ∧ ¬isspace((unsigned char) es→cbuf[es→cursor])) {
  while (isspace((unsigned char) es→cbuf[−−ncursor])) ;
  ncursor++;
}
if (ncursor > es→cursor) {
  c1 ← es→cursor;
  c2 ← ncursor;
}
else {
  c1 ← ncursor;
  c2 ← es→cursor;
  if (cmd[1] ≡ '%') c2++;
}
```

This code is used in section 1102.

1104. C: like "c$", change (delete and enter insert mode) from the cursor position to the end of the current line.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'C': modified ← 1;
  hnum ← hlast;
  del_range(es→cursor, es→linelen);
  insert ← INSERT;
  break;
```

[1] Yank.

1105. D: similarly delete from the cursor position to the end of the current line but does not enter insert mode.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'D': yank_range(es→cursor, es→linelen);
  del_range(es→cursor, es→linelen);
  if (es→cursor ≠ 0) es→cursor−−;
  break;
```

1106. x: delete the character under the cursor.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'x':
  if (es→linelen ≡ 0) return −1;
  modified ← 1;
  hnum ← hlast;
  for (cur ← es→cursor; cur < es→linelen; cur++)
    if (¬isu8cont(es→cbuf[cur]))
      if (argcnt−− ≡ 0) break;
  yank_range(es→cursor, cur);
  del_range(es→cursor, cur);
  break;
```

1107. X: delete the character before the cursor.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'X':
  if (es→cursor ≡ 0) return −1;
  modified ← 1;
  hnum ← hlast;
  for (cur ← es→cursor; cur > 0; cur−−)
    if (¬isu8cont(es→cbuf[cur]))
      if (argcnt−− ≡ 0) break;
  yank_range(cur, es→cursor);
  del_range(cur, es→cursor);
  es→cursor ← cur;
  break;
```

1108. p: paste the previously cut/copied text before the current cursor position.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'p': modified ← 1;
  hnum ← hlast;
  if (es→linelen ≠ 0) es→cursor++;
  while (putbuf(ybuf, yanklen, 0) ≡ 0 ∧ −−argcnt > 0) ;
  if (es→cursor ≠ 0) es→cursor−−;
  if (argcnt ≠ 0) return −1;
  break;
```

1109. P: paste the previously cut/copied text after the current cursor position.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'P': modified ← 1;
  hnum ← hlast;
  any ← 0;
  while (putbuf(ybuf, yanklen, 0) ≡ 0 ∧ −−argcnt > 0) any ← 1;
  if (any ∧ es→cursor ≠ 0) es→cursor −−;
  if (argcnt ≠ 0) return −1;
  break;
```

1110.

```
⟨ vi mode source 1028 ⟩ +≡
  static void yank_range(int a, int b)
  {
    yanklen ← b − a;
    if (yanklen ≠ 0) memmove(ybuf, &es→cbuf[a], yanklen);
  }
```

1111. vi Word Expansion.

```
⟨ Type definitions 17 ⟩ +≡
  enum expand_mode {
    NONE, EXPAND, COMPLETE, PRINT
  };
```

1112.

```
⟨ vi mode static variables 1022 ⟩ +≡
  static enum expand_mode expanded ← NONE;      /* pretend "last" input was expanded */
```

1113. =/Ctrl-E: display all possible expansions of the word under the cursor.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case '=':      /* AT&T ksh */
case CTRL('e'):      /* Nonstandard vi/ksh */
  print_expansions(es);
  break;
```

1114. Not noted in the original sources, this command case falls through to the next, **case** `'\\'` ∨ `CTRL('f')`.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case CTRL('['):      /* some annoying AT&T ksh's */
  if (¬Flag(FVIESCCOMPLETE)) return −1;
```

1115. \/Ctrl-F: complete the word under the cursor (if possible and unique) ending in insert mode with the cursor after the last character appended.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case '\\':      /* AT&T ksh */
case CTRL('f'):      /* Nonstandard vi/ksh */
  complete_word(1, argcnt);
  break;
```

1116. Ctrl-I (Ctrl-eye—⟨*tab*⟩): complete the current mode if the `FVITABCOMPLETE` flag allows.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case CTRL('i'):      /* Nonstandard vi/ksh */
  if (¬Flag(FVITABCOMPLETE)) return −1;
  complete_word(1, argcnt);
  break;
```

1117. */Ctrl-X: replace the current word with *all* its possible expansions.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case '*':      /* AT&T ksh */
case CTRL('x'):      /* Nonstandard vi/ksh */
  expand_word(1);
  break;
```

1118. Replace word under the cursor with all of its expansions.

```
⟨ vi mode source 1028 ⟩ +≡
  static int expand_word(int command)
  {
    static struct edstate *buf;
    int rval ← 0;
    int nwords;
    int start, end;
    char **words;
    int i;
    if (command ≡ 0 ∧ expanded ≡ EXPAND ∧ buf) {      /* Undo previous expansion */
      restore_edstate(es, buf);
      buf ← Λ;
      expanded ← NONE;
      return 0;
    }
    if (buf) {
      free_edstate(buf);
      buf ← Λ;
    }
    nwords ← x_cf_glob(XCF_COMMAND_FILE | XCF_FULLPATH, es→cbuf, es→linelen, es→cursor, &start, &end,
        &words, Λ);
    if (nwords ≡ 0) {
      vi_error( );
      return −1;
    }
    buf ← save_edstate(es);
    expanded ← EXPAND;
    del_range(start, end);
    es→cursor ← start;
    for (i ← 0; i < nwords; ) {
      if (x_escape(words[i], strlen(words[i]), x_vi_putbuf) ≠ 0) {
        rval ← −1;
        break;
      }
      if (++i < nwords ∧ putbuf("␣", 1, 0) ≠ 0) {
        rval ← −1;
        break;
      }
    }
    i ← buf→cursor − end;
    if (rval ≡ 0 ∧ i > 0) es→cursor += i;
    modified ← 1;
    hnum ← hlast;
    insert ← INSERT;
    lastac ← 0;
    refresh_line(0);
    return rval;
  }
```

1119. ⟨ vi mode source 1028 ⟩ +≡

```
static int complete_word(int command, int count)
{
  static struct edstate *buf;
  int rval ← 0;
  int nwords;
  int start, end;
  char **words;
  char *match;
  int match_len;
  int is_unique;
  int is_command;
  if (command ≡ 0 ∧ expanded ≡ COMPLETE ∧ buf) {      /* Undo previous completion */
    print_expansions(buf);
    expanded ← PRINT;
    return 0;
  }
  if (command ≡ 0 ∧ expanded ≡ PRINT ∧ buf) {
    restore_edstate(es, buf);
    buf ← Λ;
    expanded ← NONE;
    return 0;
  }
  if (buf) {
    free_edstate(buf);
    buf ← Λ;
  }
  nwords ← x_cf_glob(XCF_COMMAND_FILE | (count ? XCF_FULLPATH : 0), es¬cbuf, es¬linelen, es¬cursor,
      &start, &end, &words, &is_command);
    /* XCF_FULLPATH for count because the menu printed by print_expansions was done this way. */
  if (nwords ≡ 0) {
    vi_error( );
    return −1;
  }
  if (count) {
    int i;
    count−−;
    if (count ≥ nwords) {
      vi_error( );
      x_print_expansions(nwords, words, is_command);
      x_free_words(nwords, words);
      redraw_line(0, 0);
      return −1;
    }
    ⟨ Expand the nth word 1120 ⟩
    match_len ← strlen(match);
    is_unique ← 1;      /* expanded ← PRINT; next call undo */
  }
  else {
    match ← words[0];
    match_len ← x_longest_prefix(nwords, words);
    expanded ← COMPLETE;      /* next call will list completions */
```

```
    is_unique ← nwords ≡ 1;
  }
  buf ← save_edstate(es);
  del_range(start, end);
  es¬cursor ← start;
  rval ← x_escape(match, match_len, x_vi_putbuf);
    /* escape all shell-sensitive characters and put the result into the command buffer */
  if (rval ≡ 0 ∧ is_unique) {      /* If the match is exact don't undo—allows directory completions to be
        used (ie. complete the next portion of the path). */
    expanded ← NONE;
    if (match_len > 0 ∧ match[match_len − 1] ≠ '/') rval ← putbuf("␣", 1, 0);
      /* If not a directory add a space to the end. */
  }
  x_free_words(nwords, words);
  modified ← 1;
  hnum ← hlast;
  insert ← INSERT;
  lastac ← 0;      /* prevent this from being redone */
  refresh_line(0);
  return rval;
}
```

1120. ⟨ Expand the nth word 1120 ⟩ ≡

```
if (is_command) {
  match ← words[count] + x_basename(words[count], Λ);
  for (i ← 0; i < nwords; i++)
    if (i ≠ count ∧ strcmp(words[i] + x_basename(words[i], Λ), match) ≡ 0) {
      /* If more than one match is possible, use the full path */
      match ← words[count];
      break;
    }
}
else match ← words[count];
```

This code is used in section 1119.

1121. vi Cursor Navigation. b/B: move backward to the beginning of the current word.

```
⟨ Perform a vi movement command 1121 ⟩ ≡
case 'b': case 'B':
  if (¬sub ∧ es→cursor ≡ 0) return −1;
  ncursor ← (∗cmd ≡ 'b' ? backword : Backword)(argcnt);
  break;
```

See also sections 1122, 1123, 1125, 1126, 1127, 1128, 1129, 1130, 1131, and 1132.

This code is used in section 1078.

1122. e/E: move to the end of the current word.

```
⟨ Perform a vi movement command 1121 ⟩ +≡
case 'e': case 'E':
  if (¬sub ∧ es→cursor + 1 ≥ es→linelen) return −1;
  ncursor ← (∗cmd ≡ 'e' ? endword : Endword)(argcnt);
  if (¬sub)
    while (isu8cont((unsigned char) es→cbuf[−−ncursor])) continue;
  break;
```

1123. Move to (f/F) or before (t/T) the named character, or the next one (,/;—, in the opposite direction). The capital-letter commands move backwards.

```
⟨ Perform a vi movement command 1121 ⟩ +≡
case 'f': case 'F': case 't': case 'T': fsavecmd ← ∗cmd;
  fsavech ← cmd[1];
case ',': case ';':
  if (fsavecmd ≡ '␣') return −1;
  i ← fsavecmd ≡ 'f' ∨ fsavecmd ≡ 'F';
  t ← fsavecmd > 'a';
  if (∗cmd ≡ ',') t ← ¬t;
  if ((ncursor ← findch(fsavech, argcnt, t, i)) < 0) return −1;
  if (sub ∧ t) ncursor ++;
  break;
```

1124. ⟨ vi mode source 1028 ⟩ +≡

```
static int findch(int ch, int cnt, int forw, int incl)
{
  int ncursor;
  if (es→linelen ≡ 0) return −1;
  ncursor ← es→cursor;
  while (cnt−−) {
    do {
      if (forw) {
        if (++ncursor ≡ es→linelen) return −1;
      }
      else {
        if (−−ncursor < 0) return −1;
      }
    } while (es→cbuf[ncursor] ≠ ch);
  }
  if (¬incl) {
    if (forw) ncursor−−;
    else ncursor++;
  }
  return ncursor;
}
```

1125. h/Ctrl-H: move backwards one character.

⟨ Perform a vi movement command 1121 ⟩ +≡

```
case 'h': case CTRL('h'):
  if (¬sub ∧ es→cursor ≡ 0) return −1;
  for (ncursor ← es→cursor; ncursor > 0; ncursor−−)
    if (¬isu8cont(es→cbuf[ncursor]))
      if (argcnt−− ≡ 0) break;
  break;
```

1126. ␣ (⟨*space*⟩)/l (ell): move forwards one character.

⟨ Perform a vi movement command 1121 ⟩ +≡

```
case '␣': case 'l':
  if (¬sub ∧ es→cursor + 1 ≥ es→linelen) return −1;
  for (ncursor ← es→cursor; ncursor < es→linelen; ncursor++)
    if (¬isu8cont(es→cbuf[ncursor]))
      if (argcnt−− ≡ 0) break;
  break;
```

1127. w/W: move to the beginning of the next word.

⟨ Perform a vi movement command 1121 ⟩ +≡

```
case 'w': case 'W':
  if (¬sub ∧ es→cursor + 1 ≥ es→linelen) return −1;
  ncursor ← (*cmd ≡ 'w' ? forwword : Forwword)(argcnt);
  break;
```

1128. 0: move to the beginning of the line.

```
⟨ Perform a vi movement command 1121 ⟩ +≡
case '0': ncursor ← 0;
  break;
```

1129. ^: move to first non-whitespace character on the line.

```
⟨ Perform a vi movement command 1121 ⟩ +≡
case '^': ncursor ← 0;
  while (ncursor < es¬linelen − 1 ∧ isspace((unsigned char) es¬cbuf[ncursor])) ncursor++;
  break;
```

1130. |: move to the first or the nth character on the line.

```
⟨ Perform a vi movement command 1121 ⟩ +≡
case '|': ncursor ← argcnt;
  if (ncursor > es¬linelen) ncursor ← es¬linelen;
  if (ncursor) ncursor−−;
  while (isu8cont(es¬cbuf[ncursor])) ncursor−−;
  break;
```

1131. $: move to the end of the line.

```
⟨ Perform a vi movement command 1121 ⟩ +≡
case '$': ncursor ← es¬linelen;
  break;
```

1132. %: move to the opposite of the next bracket after or at the cursor position.

```
⟨ Perform a vi movement command 1121 ⟩ +≡
case '%': ncursor ← es¬cursor;
  while (ncursor < es¬linelen ∧ (i ← bracktype(es¬cbuf[ncursor])) ≡ 0) ncursor++;
  if (ncursor ≡ es¬linelen) return −1;
  bcount ← 1;
  do {
    if (i > 0) {
      if (++ncursor ≥ es¬linelen) return −1;
    }
    else {
      if (−−ncursor < 0) return −1;
    }
    t ← bracktype(es¬cbuf[ncursor]);
    if (t ≡ i) bcount++;
    else if (t ≡ −i) bcount−−;
  } while (bcount ≠ 0);
  if (sub ∧ i > 0) ncursor++;
  break;
```

1133. ⟨ vi mode source 1028 ⟩ +≡

```
static int bracktype(int ch)
{
  switch (ch) {
  case '(': return 1;
  case '[': return 2;
  case '{': return 3;
  case ')': return -1;
  case ']': return -2;
  case '}': return -3;
  default: return 0;
  }
}
```

1134. Move right one character, and then to the beginning of the next word.

⟨ vi mode source 1028 ⟩ +≡

```
static int forwword(int argcnt)
{
  int ncursor, skip_space, want_letnum;
  unsigned char uc;
  ncursor ← es→cursor;
  while (ncursor < es→linelen ∧ argcnt--) {
    skip_space ← 0;
    want_letnum ← -1;
    ncursor--;
    while (++ncursor < es→linelen) {
      uc ← es→cbuf[ncursor];
      if (isspace(uc)) {
        skip_space ← 1;
        continue;
      }
      else if (skip_space) break;
      if (uc & 0x80) continue;
      if (want_letnum ≡ -1) want_letnum ← letnum(uc);
      else if (want_letnum ≠ letnum(uc)) break;
    }
  }
  return ncursor;
}
```

1135. Move left one character, and then to the beginning of the word.

```
⟨ vi mode source 1028 ⟩ +≡
  static int backword(int argcnt)
  {
    int ncursor, skip_space, want_letnum;
    unsigned char uc;
    ncursor ← es→cursor;
    while (ncursor > 0 ∧ argcnt−−) {
      skip_space ← 1;
      want_letnum ← −1;
      while (ncursor−− > 0) {
        uc ← es→cbuf[ncursor];
        if (isspace(uc)) {
          if (skip_space) continue;
          else break;
        }
        skip_space ← 0;
        if (uc & 0x80) continue;
        if (want_letnum ≡ −1) want_letnum ← letnum(uc);
        else if (want_letnum ≠ letnum(uc)) break;
      }
      ncursor++;
    }
    return ncursor;
  }
```

1136. Move right one character, and then to the byte after the word.

```
⟨ vi mode source 1028 ⟩ +≡
  static int endword(int argcnt)
  {
    int ncursor, skip_space, want_letnum;
    unsigned char uc;
    ncursor ← es→cursor;
    while (ncursor < es→linelen ∧ argcnt−−) {
      skip_space ← 1;
      want_letnum ← −1;
      while (++ncursor < es→linelen) {
        uc ← es→cbuf[ncursor];
        if (isspace(uc)) {
          if (skip_space) continue;
          else break;
        }
        skip_space ← 0;
        if (uc & 0x80) continue;
        if (want_letnum ≡ −1) want_letnum ← letnum(uc);
        else if (want_letnum ≠ letnum(uc)) break;
      }
    }
    return ncursor;
  }
```

1137. Move right one character, and then to the beginning of the next big word.

```
⟨ vi mode source 1028 ⟩ +≡
  static int Forwword(int argcnt)
  {
    int ncursor;
    ncursor ← es→cursor;
    while (ncursor < es→linelen ∧ argcnt−−) {
      while (¬isspace((unsigned char) es→cbuf[ncursor]) ∧ ncursor < es→linelen) ncursor++;
      while (isspace((unsigned char) es→cbuf[ncursor]) ∧ ncursor < es→linelen) ncursor++;
    }
    return ncursor;
  }
```

1138. Move left one character, and then to the beginning of the big word.

```
⟨ vi mode source 1028 ⟩ +≡
  static int Backword(int argcnt)
  {
    int ncursor;
    ncursor ← es→cursor;
    while (ncursor > 0 ∧ argcnt−−) {
      while (−−ncursor ≥ 0 ∧ isspace((unsigned char) es→cbuf[ncursor])) ;
      while (ncursor ≥ 0 ∧ ¬isspace((unsigned char) es→cbuf[ncursor])) ncursor−−;
      ncursor++;
    }
    return ncursor;
  }
```

1139. Move right one character, and then to the byte after the big word.

```
⟨ vi mode source 1028 ⟩ +≡
  static int Endword(int argcnt)
  {
    int ncursor;
    ncursor ← es→cursor;
    while (ncursor < es→linelen ∧ argcnt−−) {
      while (++ncursor < es→linelen ∧ isspace((unsigned char) es→cbuf[ncursor])) ;
      while (ncursor < es→linelen ∧ ¬isspace((unsigned char) es→cbuf[ncursor])) ncursor++;
    }
    return ncursor;
  }
```

1140. vi History Navigation. Go to a specific (g) or the first (G) command in the history.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'g': if (¬argcnt) argcnt ← hlast;      /* FALLTHROUGH */
case 'G':
  if (¬argcnt) argcnt ← 1;
  else argcnt ← hlast − (source→line − argcnt);
  if (grabhist(modified, argcnt − 1) < 0) return −1;
  else {
    modified ← 0;
    hnum ← argcnt − 1;
  }
  break;
```

1141. j, + or Ctrl-N: Navigate forward through the command history.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'j': case '+': case CTRL('n'):
  if (grabhist(modified, hnum + argcnt) < 0) return −1;
  else {
    modified ← 0;
    hnum += argcnt;
  }
  break;
```

1142. k, - or Ctrl-P: Navigate backwards through the command history.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'k': case '-': case CTRL('p'):
  if (grabhist(modified, hnum − argcnt) < 0) return −1;
  else {
    modified ← 0;
    hnum −= argcnt;
  }
  break;
```

1143. Search forwards (/), backwards (?) or again (n/N).

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case '?': if (hnum ≡ hlast) hnum ← −1;      /* FALLTHROUGH */
case '/': c3 ← 1; srchlen ← 0; lastsearch ← *cmd;      /* FALLTHROUGH (again) */
case 'n': case 'N':
  if (lastsearch ≡ '␣') return −1;
  if (lastsearch ≡ '?') c1 ← 1;
  else c1 ← 0;
  if (*cmd ≡ 'N') c1 ← ¬c1;
  if ((c2 ← grabsearch(modified, hnum, c1, srchpat)) < 0) {
    if (c3) {
      restore_cbuf();
      refresh_line(0);
    }
    return −1;
  }
  else {
    modified ← 0;
    hnum ← c2;
    ohnum ← hnum;
  }
  break;
```

1144. _: append the last or nth word from the previous command.

```
#define issp(c) (isspace((unsigned char)(c)) ∨ (c) ≡ '\n')
⟨ Perform a vi mode command 1044 ⟩ +≡
case '_':
  {
    int inspace;
    char *p, *sp;
    if (histnum(-1) < 0) return -1;
    p ← *histpos( );
    if (argcnt) {
      while (*p ∧ issp(*p)) p++;
      while (*p ∧ --argcnt) {
        while (*p ∧ ¬issp(*p)) p++;
        while (*p ∧ issp(*p)) p++;
      }
      if (¬*p) return -1;
      sp ← p;
    }
    else {
      sp ← p;
      inspace ← 0;
      while (*p) {
        if (issp(*p)) inspace ← 1;
        else if (inspace) {
          inspace ← 0;
          sp ← p;
        }
        p++;
      }
      p ← sp;
    }
    modified ← 1;
    hnum ← hlast;
    if (es→cursor ≠ es→linelen) es→cursor++;
    while (*p ∧ ¬issp(*p)) {
      argcnt++;
      p++;
    }
    if (putbuf("␣", 1, 0) ≠ 0) argcnt ← -1;
    else if (putbuf(sp, argcnt, 0) ≠ 0) argcnt ← -1;
    if (argcnt < 0) {
      if (es→cursor ≠ 0) es→cursor--;
      return -1;
    }
    insert ← INSERT;
  }
  break;
```

1145. ⟨ vi mode VSEARCH state 1145 ⟩ ≡

```
if (ch ≡ '\r' ∨ ch ≡ '\n') {      /* ∨ch ≡ CTRL('[') */
  restore_cbuf();      /* Repeat last search? */
  if (srchlen ≡ 0) {
    if (¬srchpat[0]) {
      vi_error();
      state ← VNORMAL;
      refresh_line(0);
      return 0;
    }
  }
  else {
    locpat[srchlen] ← '\0';
    (void) strlcpy(srchpat, locpat, sizeof srchpat);
  }
  state ← VCMD;
}
else if (ch ≡ edchars.erase ∨ ch ≡ CTRL('h')) {
  if (srchlen ≠ 0) {
    do {
      srchlen−−;
      es¬linelen −= char_len((unsigned char) locpat[srchlen]);
    } while (srchlen > 0 ∧ isu8cont(locpat[srchlen]));
    es¬cursor ← es¬linelen;
    refresh_line(0);
    return 0;
  }
  restore_cbuf();
  state ← VNORMAL;
  refresh_line(0);
}
else if (ch ≡ edchars.kill) {
  srchlen ← 0;
  es¬linelen ← 1;
  es¬cursor ← 1;
  refresh_line(0);
  return 0;
}
else if (ch ≡ edchars.werase) {
  struct edstate new_es, *save_es;
  int i;
  int n ← srchlen;
  new_es.cursor ← n;
  new_es.cbuf ← locpat;
  save_es ← es;
  es ← &new_es;
  n ← backword(1);
  es ← save_es;
  for (i ← srchlen; −−i ≥ n; ) es¬linelen −= char_len((unsigned char) locpat[i]);
  srchlen ← n;
  es¬cursor ← es¬linelen;
  refresh_line(0);
```

```
    return 0;
  }
  else {
    if (srchlen ≡ SRCHLEN − 1) vi_error( );
    else {
      locpat[srchlen++] ← ch;
      if ((ch & 0x80) ∧ Flag(FVISHOW8)) {
        if (es→linelen + 2 > es→cbufsize) vi_error( );
        es→cbuf[es→linelen++] ← '^';
        es→cbuf[es→linelen++] ← '-';
        ch &= 0x7f;
      }
      if (ch < ' ' ∨ ch ≡ 0x7f) {
        if (es→linelen + 2 > es→cbufsize) vi_error( );
        es→cbuf[es→linelen++] ← '^';
        es→cbuf[es→linelen++] ← ch ⊕ '@';
      }
      else {
        if (es→linelen ≥ es→cbufsize) vi_error( );
        es→cbuf[es→linelen++] ← ch;
      }
      es→cursor ← es→linelen;
      refresh_line(0);
    }
    return 0;
  }
  break;
```

This code is used in section 1063.

1146. ⟨ vi mode source 1028 ⟩ +≡

```
static int grabhist(int save, int n)
{
  char *hptr;
  if (n < 0 ∨ n > hlast) return −1;
  if (n ≡ hlast) {
    restore_cbuf( );
    ohnum ← n;
    return 0;
  }
  (void) histnum(n);
  if ((hptr ← *histpos( )) ≡ Λ) {
    internal_warningf("%s: bad history array", __func__);
    return −1;
  }
  if (save) save_cbuf( );
  if ((es→linelen ← strlen(hptr)) ≥ es→cbufsize) es→linelen ← es→cbufsize − 1;
  memmove(es→cbuf, hptr, es→linelen);
  es→cursor ← 0;
  ohnum ← n;
  return 0;
}
```

1147. ⟨ vi mode source 1028 ⟩ +≡

```
static int grabsearch(int save, int start, int fwd, char *pat)
{
  char *hptr;
  int hist;
  int anchored;
  if ((start ≡ 0 ∧ fwd ≡ 0) ∨ (start ≥ hlast − 1 ∧ fwd ≡ 1)) return −1;
  if (fwd) start++;
  else start−−;
  anchored ← *pat ≡ '^' ? (++pat, 1) : 0;
  if ((hist ← findhist(start, fwd, pat, anchored)) < 0) {
    if (start ≠ 0 ∧ fwd ∧ strcmp(holdbuf, pat) ≥ 0) {
      /* TODO: should strcmp be strncmp (was match)? */
      restore_cbuf();
      return 0;
    }
    else return −1;
  }
  if (save) save_cbuf();
  histnum(hist);
  hptr ← *histpos();
  if ((es→linelen ← strlen(hptr)) ≥ es→cbufsize) es→linelen ← es→cbufsize − 1;
  memmove(es→cbuf, hptr, es→linelen);
  es→cursor ← 0;
  return hist;
}
```

1148. Other vi Commands.

1149. @ (at): perform a macro, stored as an alias named "_" followed by the argument character exactly (so macros are case sensitive and not limited to the 26 lower-case (52) English letters).

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case '@':
  {
    static char alias[] ← "_\0";
    struct tbl *ap;
    int olen, nlen;
    char *p, *nbuf;
    ⟨ Look up macro letter in alias list 1150 ⟩
    ⟨ Ensure a vi macro is not called recursively 1151 ⟩
    ⟨ Copy the alias to the macro buffer 1152 ⟩
  }
  break;
```

1150. ⟨ Look up macro letter in alias list 1150 ⟩ ≡

```
alias[1] ← cmd[1];
ap ← ktsearch(&aliases, alias, hash(alias));
if (¬cmd[1] ∨ ¬ap ∨ ¬(ap→flag & ISSET)) return −1;
```

This code is used in section 1149.

1151. ⟨ Ensure a vi macro is not called recursively 1151 ⟩ ≡

```
if ((p ← (char *) macro.p))
  while ((p ← strchr(p, '\0')) ∧ p[1])
    if (*++p ≡ cmd[1]) return −1;
```

This code is used in section 1149.

1152. ⟨ Copy the alias to the macro buffer 1152 ⟩ ≡

```
nlen ← strlen(ap→val.s) + 1;
olen ← ¬macro.p ? 2 : macro.len − (macro.p − macro.buf);
nbuf ← alloc(nlen + 1 + olen, APERM);
memcpy(nbuf, ap→val.s, nlen);
nbuf[nlen++] ← cmd[1];
if (macro.p) {
  memcpy(nbuf + nlen, macro.p, olen);
  afree(macro.buf, APERM);
  nlen += olen;
}
else {
  nbuf[nlen++] ← '\0';
  nbuf[nlen++] ← '\0';
}
macro.p ← macro.buf ← (unsigned char *) nbuf;
macro.len ← nlen;
```

This code is used in section 1149.

1153. v: edit the current line in an external editor—does not work if the current line is not the first/only.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'v':
  if (es¬linelen ≡ 0 ∧ argcnt ≡ 0) return −1;
  if (¬argcnt) {
    if (modified) {
      es¬cbuf[es¬linelen] ← '\0';
      source¬line++;
      histsave(source¬line, es¬cbuf, 1);
    }
    else argcnt ← source¬line + 1 − (hlast − hnum);
  }
  shf_snprintf(es¬cbuf, es¬cbufsize, argcnt ? "%s␣%d" : "%s", "fc␣-e␣${VISUAL:-${EDITOR:-vi}}␣--",
      argcnt);
  es¬linelen ← strlen(es¬cbuf);
  return 2;
```

1154. u: undo the previous command.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'u': t ← es;
  es ← undo;
  undo ← t;
  break;
```

1155. U: undo all (possible) changes to the current line.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case 'U':
  if (¬modified) return −1;
  if (grabhist(modified, ohnum) < 0) return −1;
  modified ← 0;
  hnum ← ohnum;
  break;
```

1156. ~: invert the case of the character under the cursor and move forward.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case '~':
  {
    char *p;
    unsigned char c;
    int i;
    if (es→linelen ≡ 0) return −1;
    for (i ← 0; i < argcnt; i++) {
      p ← &es→cbuf[es→cursor];
      c ← (unsigned char) *p;
      if (islower(c)) {
        modified ← 1;
        hnum ← hlast;
        *p ← toupper(c);
      }
      else if (isupper(c)) {
        modified ← 1;
        hnum ← hlast;
        *p ← tolower(c);
      }
      if (es→cursor < es→linelen − 1) es→cursor++;
    }
    break;
  }
```

1157. #: comment or un-comment the current line. Like `v` this also does not work with multi-line commands.

```
⟨ Perform a vi mode command 1044 ⟩ +≡
case '#':
  {
    int ret ← x_do_comment(es→cbuf, es→cbufsize, &es→linelen);
    if (ret ≥ 0) es→cursor ← 0;
    return ret;
  }
```

1158. Mathematical Expressions.

⟨ expr.c 1158 ⟩ ≡
```
#include <ctype.h>
#include <limits.h>
#include <string.h>
#include "sh.h"
  static void evalerr(Expr_state *, enum error_type, const char *)__attribute__((__noreturn__));
  static struct tbl *evalexpr(Expr_state *, enum Prec);
  static void Mtoken(Expr_state *);
  static struct tbl *do_ppmm(Expr_state *, enum token, struct tbl *, bool);
  static void assign_check(Expr_state *, enum token, struct tbl *);
  static struct tbl *tempvar(void);
  static struct tbl *intvar(Expr_state *, struct tbl *);
  ⟨ Mathematical Operators 1167 ⟩
```
See also sections 1161, 1168, 1170, 1172, 1176, 1178, 1186, 1187, and 1188.

1159. ⟨ Shared function declarations 4 ⟩ +≡
```
  int evaluate(const char *, int64_t *, int, bool);
  int v_evaluate(struct tbl *, const char *, volatile int, bool);
```

1160. ⟨ Type definitions 17 ⟩ +≡
```
  enum error_type {
    ET_UNEXPECTED, ET_BADLIT, ET_RECURSIVE, ET_LVALUE, ET_RDONLY, ET_STR
  };
```

1161. ⟨ expr.c 1158 ⟩ +≡
```
  static void evalerr(Expr_state *es, enum error_type type, const char *str)
  {
    char tbuf[2];
    const char *s;
    es→arith ← false;
    switch (type) {
    case ET_UNEXPECTED: ⟨ Unexpected evaluation; break 1162 ⟩
    case ET_BADLIT: warningf(true, "%s: bad number '%s'", es→expression, str); break;
    case ET_RECURSIVE:
      warningf(true, "%s: expression recurses on parameter '%s'", es→expression, str); break;
    case ET_LVALUE: warningf(true, "%s: %s requires lvalue", es→expression, str); break;
    case ET_RDONLY: warningf(true, "%s: %s applied to read only variable", es→expression, str);
      break;
    default: case ET_STR:     /* keep gcc happy */
      warningf(true, "%s: %s", es→expression, str); break;
    }
    unwind (LAEXPR);
  }
```

1162. ⟨ Unexpected evaluation; **break** 1162 ⟩ ≡

```
switch (es→tok) {
case VAR: s ← es→val→name; break;
case LIT: s ← str_val(es→val); break;
case END: s ← "end␣of␣expression"; break;
case BAD: tbuf[0] ← *es→tokp;
  tbuf[1] ← '\0';
  s ← tbuf;
  break;
default: s ← opinfo[(int) es→tok].name;
}
warningf(true, "%s:␣unexpected␣`%s'", es→expression, s);
break;
```

This code is used in section 1161.

1163. These symbols are arranged in groups so that binary and/or assignment operators can be distinguished. The order must match that of *opinfo* below. Except where noted these are binary operators.

```
#define IS_BINOP(op) (((int) op) ≥ (int) O_EQ ∧ ((int) op) ≤ (int) O_COMMA)
#define IS_ASSIGNOP(op) ((int)(op) ≥ (int) O_ASN ∧ (int)(op) ≤ (int) O_BORASN)
```

⟨ Type definitions 17 ⟩ +≡

```
enum token {
  O_PLUSPLUS ← 0, O_MINUSMINUS,      /* some (long) unary operators */
  O_EQ, O_NE,      /* also binary operators, assignments are in range O_ASN–O_BORASN: */
   O_ASN, O_TIMESASN, O_DIVASN, O_MODASN, O_PLUSASN, O_MINUSASN,
   O_LSHIFTASN, O_RSHIFTASN, O_BANDASN, O_BXORASN, O_BORASN,
  O_LSHIFT, O_RSHIFT, O_LE, O_GE, O_LT, O_GT, O_LAND, O_LOR,
  O_TIMES, O_DIV, O_MOD, O_PLUS, O_MINUS, O_BAND, O_BXOR, O_BOR,
  O_TERN, O_COMMA,
  O_BNOT, O_LNOT,      /* remaining unary operators */
  OPEN_PAREN, CLOSE_PAREN, CTERN,      /* miscellaneous */
  VAR, LIT, END, BAD      /* things that don't appear in the opinfo table */
};
```

1164. `#define MAX_PREC P_COMMA`

⟨ Type definitions 17 ⟩ +≡

```
enum Prec {
  P_PRIMARY ← 0,      /* VAR LIT (...) ~ ! - + */
  P_MULT,      /* * / % */
  P_ADD,      /* + - */
  P_SHIFT,      /* << >> */
  P_RELATION,      /* < <= > >= */
  P_EQUALITY,      /* == != */
  P_BAND,      /* & */
  P_BXOR,      /* ^ */
  P_BOR,      /* | */
  P_LAND,      /* && */
  P_LOR,      /* || */
  P_TERN,      /* ?: */
  P_ASSIGN,      /* = *= /= %= += -= <<= >>= &= ^= |= */
  P_COMMA      /* , */
};
```

1165. ⟨ Type definitions 17 ⟩ +≡

```
struct Opinfo {
  char name[4];
  int len;      /* name length */
  enum Prec prec;      /* precedence: lower is higher */
};
```

1166. Tokens in this table must be ordered so the longest prefixes are first (eg. `++` before `+=` before `+`). They must also match the order of **enum token** above.

1167. ⟨ Mathematical Operators 1167 ⟩ ≡

```
static const struct Opinfo opinfo[] ← {
    {"++", 2, P_PRIMARY},
    {"--", 2, P_PRIMARY},
    {"==", 2, P_EQUALITY},
    {"!=", 2, P_EQUALITY},
    {"=", 1, P_ASSIGN},
    {"*=", 2, P_ASSIGN},
    {"/=", 2, P_ASSIGN},
    {"%=", 2, P_ASSIGN},
    {"+=", 2, P_ASSIGN},
    {"-=", 2, P_ASSIGN},
    {"<<=", 3, P_ASSIGN},
    {">>=", 3, P_ASSIGN},
    {"&=", 2, P_ASSIGN},
    {"^=", 2, P_ASSIGN},
    {"|=", 2, P_ASSIGN},
    {"<<", 2, P_SHIFT},
    {">>", 2, P_SHIFT},
    {"<=", 2, P_RELATION},
    {">=", 2, P_RELATION},
    {"<", 1, P_RELATION},
    {">", 1, P_RELATION},
    {"&&", 2, P_LAND},
    {"||", 2, P_LOR},
    {"*", 1, P_MULT},
    {"/", 1, P_MULT},
    {"%", 1, P_MULT},
    {"+", 1, P_ADD},
    {"-", 1, P_ADD},
    {"&", 1, P_BAND},
    {"^", 1, P_BXOR},
    {"|", 1, P_BOR},
    {"?", 1, P_TERN},
    {",", 1, P_COMMA},
    {"~", 1, P_PRIMARY},
    {"!", 1, P_PRIMARY},
    {"(", 1, P_PRIMARY},
    {")", 1, P_PRIMARY},
    {":", 1, P_PRIMARY},
    {"", 0, P_PRIMARY}      /* end of table */
};
```

This code is used in section 1158.

1168. Parse and evaluate expression.

```
⟨ expr.c 1158 ⟩ +≡
  int evaluate(const char *expr, int64_t *rval, int error_ok, bool arith)
  {
    struct tbl v;
    int ret;
    v.flag ← DEFINED | INTEGER;
    v.type ← 0;
    ret ← v_evaluate(&v, expr, error_ok, arith);
    *rval ← v.val.i;
    return ret;
  }
```

1169. Parse and evaluate expression, storing result in *vp*.

```
⟨ Type definitions 17 ⟩ +≡
  typedef struct expr_state Expr_state;
  struct expr_state {
    const char *expression;      /* expression being evaluated */
    const char *tokp;      /* lexical position */
    enum token tok;      /* token from Mtoken */
    int noassign;      /* don't do assigns (for ?:, &&, ||) */
    bool arith;      /* true if evaluating an $(( ... )) expression */
    struct tbl *val;      /* value from Mtoken */
    struct tbl *evaling;      /* variable that is being recursively expanded (EXPRINEVAL flag set) */
  };
```

1170. *v_evaluate*, used by *setstr*, may fail. If this might be acceptable the caller indicates by setting flags in *error_ok*.

```
#define KSH_UNWIND_ERROR 0x0    /* unwind the stack (longjmp) */
#define KSH_RETURN_ERROR 0x1    /* the return value indicates success (1) or failure (0) */
#define KSH_IGNORE_RDONLY 0x4    /* ignore the read-only flag */
```

⟨ expr.c 1158 ⟩ +≡

```
int v_evaluate(struct tbl *vp, const char *expr, volatile int error_ok, bool arith)
{
  struct tbl *v;
  Expr_state curstate;
  Expr_state *const es ← &curstate;    /* why? */
  int save_disable_subst;
  int i;

  curstate.expression ← curstate.tokp ← expr;
  curstate.noassign ← 0;
  curstate.arith ← arith;
  curstate.evaling ← Λ;
  curstate.val ← Λ;

  newenv(E_ERRH);
  save_disable_subst ← disable_subst;
  i ← sigsetjmp(genv→jbuf, 0);
  if (i) {⟨ unwind after an evaluation error 1171 ⟩}
  Mtoken(es);
#if 1    /* disable, and empty expressions are treated as 0 */
  if (es→tok ≡ END) { es→tok ← LIT; es→val ← tempvar(); }
#endif    /* 0 */
  v ← intvar(es, evalexpr(es, MAX_PREC));
  if (es→tok ≠ END) evalerr(es, ET_UNEXPECTED, Λ);
  if (vp→flag & INTEGER) setint_v(vp, v, es→arith);
  else setstr(vp, str_val(v), error_ok);    /* can fail if vp is readonly */
  quitenv(Λ);
  return 1;
}
```

1171. ⟨ **unwind** after an evaluation error 1171 ⟩ ≡

```
disable_subst ← save_disable_subst;
if (curstate.evaling) curstate.evaling→flag &= ∼EXPRINEVAL;
    /* Clear EXPRINEVAL of any variables we were playing with */
quitenv(Λ);
if (i ≡ LAEXPR) {
  if (error_ok ≡ KSH_RETURN_ERROR) return 0;
  errorf(Λ);
}
unwind (i);
```

This code is used in section 1170.

1172. The expression to evaluate is now in *es→expression*, this fills in the rest of the **Expr_state** object.

```
⟨ expr.c 1158 ⟩ +≡
  static void Mtoken(Expr_state *es)
  {
    const char *cp;
    int c;
    char *tvar;
    for (cp ← es→tokp; (c ← *cp), isspace((unsigned char) c); cp++) ;    /* skip white space */
    es→tokp ← cp;
    if (c ≡ '\0') es→tok ← END;
    else if (letter(c)) {⟨ Parse an expression beginning with a letter 1173 ⟩}
    else if (digit(c)) {⟨ Parse an expression beginning with a number 1174 ⟩}
    else {⟨ Parse an expression beginning with a symbol 1175 ⟩}
    es→tokp ← cp;
  }
```

1173.
```
⟨ Parse an expression beginning with a letter 1173 ⟩ ≡
  for ( ; letnum(c); c ← *cp) cp++;    /* find the next non-alphanumeric character */
  if (c ≡ '[') {    /* found an array, extract the index */
    int len;
    len ← array_ref_len(cp);
    if (len ≡ 0) evalerr(es, ET_STR, "missing␣]");
    cp += len;
  }
  else if (c ≡ '(') {
    ;    /* TODO: add math functions (all take single argument): abs acos asin atan cos cosh exp int
        log sin sinh sqrt tan tanh */
  }
  if (es→noassign) {
    es→val ← tempvar( );
    es→val→flag |= EXPRLVALUE;
  }
  else {
    tvar ← str_nsave(es→tokp, cp − es→tokp, ATEMP);
    es→val ← global(tvar);
    afree(tvar, ATEMP);
  }
  es→tok ← VAR;
```

This code is used in section 1172.

1174.
```
⟨ Parse an expression beginning with a number 1174 ⟩ ≡
  for ( ; c ≠ '_' ∧ (letnum(c) ∨ c ≡ '#'); c ← *cp++) ;
  tvar ← str_nsave(es→tokp, −−cp − es→tokp, ATEMP);
  es→val ← tempvar( );
  es→val→flag &= ∼INTEGER;
  es→val→type ← 0;
  es→val→val.s ← tvar;
  if (setint_v(es→val, es→val, es→arith) ≡ Λ) evalerr(es, ET_BADLIT, tvar);
  afree(tvar, ATEMP);
  es→tok ← LIT;
```

This code is used in section 1172.

1175. ⟨ Parse an expression beginning with a symbol 1175 ⟩ ≡

```
int i, n0;
for (i ← 0; (n0 ← opinfo[i].name[0]); i++)
  if (c ≡ n0 ∧ strncmp(cp, opinfo[i].name, opinfo[i].len) ≡ 0) {
    es→tok ← (enum token) i;
    cp += opinfo[i].len;
    break;
  }
if (¬n0) es→tok ← BAD;
```

This code is used in section 1172.

1176. Proceed to evaluate an expression. Recursively calls itself at a successively higher precedence until `P_PRIMARY` then descends precedence by unwinding the stack.

Because the contents of *es* may be changed when evaluating a single sub-expression the effect of the for loop is to ensure that, at each level of precedence, all sub-expressions are evaluated in left-to-right order.

⟨ `expr.c` 1158 ⟩ +≡

```
static struct tbl *evalexpr(Expr_state *es, enum Prec prec)
{
  struct tbl *vl, *vr ← Λ, *vasn;
  enum token op;
  int64_t res ← 0;
  if (prec ≡ P_PRIMARY) {⟨ Evaluate high-precedence expressions and return it 1177 ⟩}
  vl ← evalexpr(es, ((int) prec) − 1);      /* evaluate higher-precedence expressions first */
  for (op ← es→tok;
       IS_BINOP(op) ∧ opinfo[(int) op].prec ≡ prec;
       op ← es→tok) {
    ⟨ Perform the lower-precision operations 1179 ⟩
  }
  return vl;
}
```

1177. ⟨ Evaluate high-precedence expressions and **return** it 1177 ⟩ ≡

```
op ← es→tok;
if (op ≡ O_BNOT ∨ op ≡ O_LNOT ∨ op ≡ O_MINUS ∨ op ≡ O_PLUS) {
  Mtoken(es);
  vl ← intvar(es, evalexpr(es, P_PRIMARY));
  if (op ≡ O_BNOT) vl→val.i ← ∼vl→val.i;
  else if (op ≡ O_LNOT) vl→val.i ← ¬vl→val.i;
  else if (op ≡ O_MINUS) vl→val.i ← −vl→val.i;      /* op ≡ O_PLUS is a no-op */
}
else if (op ≡ OPEN_PAREN) {
  Mtoken(es);
  vl ← evalexpr(es, MAX_PREC);
  if (es→tok ≠ CLOSE_PAREN) evalerr(es, ET_STR, "missing␣)");
  Mtoken(es);
}
else if (op ≡ O_PLUSPLUS ∨ op ≡ O_MINUSMINUS) {
  Mtoken(es);
  vl ← do_ppmm(es, op, es→val, true);
  Mtoken(es);
}
else if (op ≡ VAR ∨ op ≡ LIT) {
  vl ← es→val;
  Mtoken(es);
}
else {
  evalerr(es, ET_UNEXPECTED, Λ);      /* NOTREACHED */
}
if (es→tok ≡ O_PLUSPLUS ∨ es→tok ≡ O_MINUSMINUS) {
  vl ← do_ppmm(es, es→tok, vl, false);
  Mtoken(es);
}
return vl;
```

This code is used in section 1176.

1178. Do a "++" or "--" operation.

⟨ expr.c 1158 ⟩ +≡

```
static struct tbl *do_ppmm(Expr_state *es, enum token op, struct tbl *vasn, bool is_prefix)
{
  struct tbl *vl;
  int oval;
  assign_check(es, op, vasn);
  vl ← intvar(es, vasn);
  oval ← op ≡ O_PLUSPLUS ? vl→val.i++ : vl→val.i--;
  if (vasn→flag & INTEGER) setint_v(vasn, vl, es→arith);
  else setint(vasn, vl→val.i);
  if (¬is_prefix)      /* undo the increment/decrement */
    vl→val.i ← oval;
  return vl;
}
```

1179. ⟨ Perform the lower-precision operations 1179 ⟩ ≡

```
Mtoken(es);
vasn ← vl;
if (op ≠ O_ASN) vl ← intvar(es, vl);      /* vl may not have a value yet */
if (IS_ASSIGNOP(op)) {
  assign_check(es, op, vasn);
  vr ← intvar(es, evalexpr(es, P_ASSIGN));
}
else if (op ≠ O_TERN ∧ op ≠ O_LAND ∧ op ≠ O_LOR) vr ← intvar(es, evalexpr(es, ((int) prec) − 1));
if ((op ≡ O_DIV ∨ op ≡ O_MOD ∨ op ≡ O_DIVASN ∨ op ≡ O_MODASN) ∧ vr→val.i ≡ 0) {      /* x/0 ? */
  if (es→noassign) vr→val.i ← 1;
  else evalerr(es, ET_STR, "zero␣divisor");
}
switch ((int) op) {⟨ Perform a mathematical operation 1180 ⟩}
if (IS_ASSIGNOP(op)) {
  vr→val.i ← res;
  if (vasn→flag & INTEGER) setint_v(vasn, vr, es→arith);
  else setint(vasn, res);
  vl ← vr;
}
else if (op ≠ O_TERN) vl→val.i ← res;      /* The ternary handler fills in vl itself */
```

This code is used in section 1176.

1180. Elementry mathematics.

⟨ Perform a mathematical operation 1180 ⟩ ≡

```
case O_TIMES: case O_TIMESASN: res ← vl→val.i * vr→val.i; break;
case O_DIV: case O_DIVASN:
  if (vl→val.i ≡ LONG_MIN ∧ vr→val.i ≡ −1) res ← LONG_MIN;
  else res ← vl→val.i/vr→val.i;
  break;
case O_MOD: case O_MODASN:
  if (vl→val.i ≡ LONG_MIN ∧ vr→val.i ≡ −1) res ← 0;
  else res ← vl→val.i % vr→val.i;
  break;
case O_PLUS: case O_PLUSASN: res ← vl→val.i + vr→val.i; break;
case O_MINUS: case O_MINUSASN: res ← vl→val.i − vr→val.i; break;
```

See also sections 1181, 1182, 1183, 1184, and 1185.

This code is used in section 1179.

1181. Bit shifting.

⟨ Perform a mathematical operation 1180 ⟩ +≡

```
case O_LSHIFT: case O_LSHIFTASN: res ← vl→val.i ≪ vr→val.i; break;
case O_RSHIFT: case O_RSHIFTASN: res ← vl→val.i ≫ vr→val.i; break;
```

1182. Predicates.

```
⟨ Perform a mathematical operation 1180 ⟩ +≡
case O_LT: res ← vl→val.i < vr→val.i; break;
case O_LE: res ← vl→val.i ≤ vr→val.i; break;
case O_GT: res ← vl→val.i > vr→val.i; break;
case O_GE: res ← vl→val.i ≥ vr→val.i; break;
case O_EQ: res ← vl→val.i ≡ vr→val.i; break;
case O_NE: res ← vl→val.i ≠ vr→val.i; break;
```

1183. Boolean & predicate logic.

```
⟨ Perform a mathematical operation 1180 ⟩ +≡
case O_BAND: case O_BANDASN: res ← vl→val.i & vr→val.i; break;
case O_BXOR: case O_BXORASN: res ← vl→val.i ⊕ vr→val.i; break;
case O_BOR: case O_BORASN: res ← vl→val.i | vr→val.i; break;
case O_LAND:
  if (¬vl→val.i) es→noassign++;
  vr ← intvar(es, evalexpr(es, ((int) prec) − 1));
  res ← vl→val.i ∧ vr→val.i;
  if (¬vl→val.i) es→noassign−−;
  break;
case O_LOR:
  if (vl→val.i) es→noassign++;
  vr ← intvar(es, evalexpr(es, ((int) prec) − 1));
  res ← vl→val.i ∨ vr→val.i;
  if (vl→val.i) es→noassign−−;
  break;
```

1184. Ternary (`... ? ... : ...`)

```
⟨ Perform a mathematical operation 1180 ⟩ +≡
case O_TERN:
  {
    int e ← vl→val.i ≠ 0;
    if (¬e) es→noassign++;
    vl ← evalexpr(es, MAX_PREC);
    if (¬e) es→noassign−−;
    if (es→tok ≠ CTERN) evalerr(es, ET_STR, "missing␣:");
    Mtoken(es);
    if (e) es→noassign++;
    vr ← evalexpr(es, P_TERN);
    if (e) es→noassign−−;
    vl ← e ? vl : vr;
  }
  break;
```

1185. Assignment and comma. Apparently the same.

```
⟨ Perform a mathematical operation 1180 ⟩ +≡
case O_ASN: res ← vr→val.i; break;
case O_COMMA: res ← vr→val.i; break;
```

1186. ⟨ expr.c 1158 ⟩ +≡

```
static struct tbl *intvar(Expr_state *es, struct tbl *vp)
{
  struct tbl *vq;
  if (vp→name[0] ≡ '\0' ∧ (vp→flag & (ISSET | INTEGER | EXPRLVALUE)) ≡ (ISSET | INTEGER))
      /* try to avoid replacing a temp variable with another temp variable */
    return vp;
  vq ← tempvar( );
  if (setint_v(vq, vp, es→arith) ≡ Λ) {
    if (vp→flag & EXPRINEVAL) evalerr(es, ET_RECURSIVE, vp→name);
    es→evaling ← vp;
    vp→flag |= EXPRINEVAL;
    disable_subst++;
    v_evaluate(vq, str_val(vp), KSH_UNWIND_ERROR, es→arith);
    disable_subst--;
    vp→flag &= ∼EXPRINEVAL;
    es→evaling ← Λ;
  }
  return vq;
}
```

1187. ⟨ expr.c 1158 ⟩ +≡

```
static struct tbl *tempvar(void)
{
  struct tbl *vp;
  vp ← alloc(sizeof(struct tbl), ATEMP);
  vp→flag ← ISSET | INTEGER;
  vp→type ← 0;
  vp→areap ← ATEMP;
  vp→val.i ← 0;
  vp→name[0] ← '\0';
  return vp;
}
```

1188. ⟨ expr.c 1158 ⟩ +≡

```
static void assign_check(Expr_state *es, enum token op, struct tbl *vasn)
{
  if (es→tok ≡ END ∨ vasn ≡ Λ ∨ (vasn→name[0] ≡ '\0' ∧ ¬(vasn→flag & EXPRLVALUE)))
    evalerr(es, ET_LVALUE, opinfo[(int) op].name);
  else if (vasn→flag & RDONLY) evalerr(es, ET_RDONLY, opinfo[(int) op].name);
}
```

1189. Mail. Originally by Robert J. Gibson, adapted for PD ksh by John R. MacMillan

⟨ mail.c 1189 ⟩ ≡

```
#include <sys/stat.h>
#include <sys/time.h>
#include <string.h>
#include <time.h>
#include "config.h"
#include "sh.h"
  static mbox_t *mplist;
  static mbox_t mbox;
  static struct timespec mlastchkd;      /* when mail was last checked */
  static time_t mailcheck_interval;
  static void munset(mbox_t *);      /* free mlist and mval */
  static mbox_t *mballoc(char *, char *);      /* allocate a new mbox */
  static void mprintit(mbox_t *);
```

See also sections 1192, 1193, 1194, 1195, 1196, 1197, and 1198.

1190. ⟨ Shared function declarations 4 ⟩ +≡

```
  void mcheck(void);
  void mcset(int64_t);
  void mbset(char *);
  void mpset(char *);
```

1191. ⟨ Type definitions 17 ⟩ +≡

```
  typedef struct Mbox {
    struct Mbox *mb_next;      /* next mbox in list */
    char *mb_path;      /* path to mail file */
    char *mb_msg;      /* to announce arrival of new mail */
    time_t mb_mtime;      /* mtime of mail file */
  } mbox_t;
```

1192. `$MAILPATH` is a linked list of mboxes. `$MAIL` is a treated as a special case of `$MAILPATH`, where the list has only one node. The same list is used for both since they are exclusive.

```
⟨ mail.c 1189 ⟩ +≡
  void mcheck(void)
  {
    mbox_t *mbp;
    struct timespec elapsed, now;
    struct tbl *vp;
    struct stat stbuf;
    static int first ← 1;
    if (mplist) mbp ← mplist;
    else if ((vp ← global("MAIL")) ∧ (vp→flag & ISSET)) mbp ← &mbox;
    else mbp ← Λ;
    if (mbp ≡ Λ) return;
    clock_gettime(CLOCK_MONOTONIC, &now);
    if (first) {
      mlastchkd ← now;
      first ← 0;
    }
    timespecsub(&now, &mlastchkd, &elapsed);
    if (elapsed.tv_sec ≥ mailcheck_interval) {
      mlastchkd ← now;
      while (mbp) {
        if (mbp→mb_path ∧ stat(mbp→mb_path, &stbuf) ≡ 0 ∧ S_ISREG(stbuf.st_mode)) {
          if (stbuf.st_size ∧ mbp→mb_mtime ≠ stbuf.st_mtime ∧ stbuf.st_atime ≤ stbuf.st_mtime)
            mprintit(mbp);
          mbp→mb_mtime ← stbuf.st_mtime;
        }
        else {      /* Some mail readers remove the mail file if all mail is read—if the file does not exist
                       assume this is the case and set mtime to zero. */
          mbp→mb_mtime ← 0;
        }
        mbp ← mbp→mb_next;
      }
    }
  }
```

1193.

```
⟨ mail.c 1189 ⟩ +≡
  void mcset(int64_t interval)
  {
    mailcheck_interval ← interval;
  }
```

1194. ⟨ mail.c 1189 ⟩ +≡

```
void mbset(char *p)
{
  struct stat stbuf;
  afree(mbox.mb_msg, APERM);
  afree(mbox.mb_path, APERM);
  mbox.mb_path ← str_save(p, APERM);      /* Save a copy to protect from export */
  mbox.mb_msg ← Λ;
  if (p ∧ stat(p, &stbuf) ≡ 0 ∧ S_ISREG(stbuf.st_mode)) mbox.mb_mtime ← stbuf.st_mtime;
  else mbox.mb_mtime ← 0;
}
```

1195. ⟨ mail.c 1189 ⟩ +≡

```
void mpset(char *mptoparse)
{
  mbox_t *mbp;
  char *mpath, *mmsg, *mval;
  char *p;
  munset(mplist);
  mplist ← Λ;
  mval ← str_save(mptoparse, APERM);
  while (mval) {
    mpath ← mval;
    if ((mval ← strchr(mval, ':')) ≠ Λ) {
      *mval ← '\0';
      mval++;
    }
    for (p ← mpath; (mmsg ← strchr(p, '%')); ) {      /* POSIX/bourne-shell say "file%message" */
      if (mmsg > mpath ∧ mmsg[−1] ≡ '\\') {      /* a literal percent? (POSIXism) */
        memmove(mmsg − 1, mmsg, strlen(mmsg) + 1);
        p ← mmsg;
        continue;
      }
      break;
    }
    if (¬mmsg ∧ ¬Flag(FPOSIX))      /* AT&T ksh says "file?message" */
      mmsg ← strchr(mpath, '?');
    if (mmsg) {
      *mmsg ← '\0';
      mmsg++;
      if (*mmsg ≡ '\0') mmsg ← Λ;
    }
    if (*mpath ≡ '\0') continue;
    mbp ← mballoc(mpath, mmsg);
    mbp→mb_next ← mplist;
    mplist ← mbp;
  }
}
```

1196. ⟨ mail.c 1189 ⟩ +≡

```
static void munset(mbox_t *mlist)
{
  mbox_t *mbp;
  while (mlist ≠ Λ) {
    mbp ← mlist;
    mlist ← mbp→mb_next;
    if (¬mlist) afree(mbp→mb_path, APERM);
    afree(mbp, APERM);
  }
}
```

1197. ⟨ mail.c 1189 ⟩ +≡

```
static mbox_t *mballoc(char *p, char *m)
{
  struct stat stbuf;
  mbox_t *mbp;
  mbp ← alloc(sizeof(mbox_t), APERM);
  mbp→mb_next ← Λ;
  mbp→mb_path ← p;
  mbp→mb_msg ← m;
  if (stat(mbp→mb_path, &stbuf) ≡ 0 ∧ S_ISREG(stbuf.st_mode)) mbp→mb_mtime ← stbuf.st_mtime;
  else mbp→mb_mtime ← 0;
  return (mbp);
}
```

1198. `#define MBMESSAGE "you␣have␣mail␣in␣$_"`

⟨ mail.c 1189 ⟩ +≡

```
static void mprintit(mbox_t *mbp)
{
  struct tbl *vp;
#if 0
    /* * I doubt this $_ overloading is bad in /bin/sh mode. Anyhow, we * crash as the code looks
       now if we do not set vp. Now, this is * easy to fix too, but I'd like to see what POSIX says before
       doing * a change like that. */
  if (¬Flag(FSH))
#endif
    setstr((vp ← local("_", false)), mbp→mb_path, KSH_RETURN_ERROR);
      /* Ignore setstr errors here (arbitrary) */
  shellf("%s\n", substitute(mbp→mb_msg ? mbp→mb_msg : MBMESSAGE, 0));
  unset(vp, 0);
}
```

1199. Built-in Commands. Divided between Bourne commands (`c_sh.c`), KSH additions (`c_ksh.c`) and test and ulimit in separate files (`c_test.c` & `c_ulimit.c`).

In the command declaration a leading "`=`" means assignments before command are kept, "`*`" means its a POSIX special builtin and "`+`" means a POSIX regular builtin (`*` and `+` should not be combined).

⟨ Type definitions 17 ⟩ +≡

```
struct builtin {
  const char *name;
  int(*func)(char **);
};
```

1200. ⟨ c_sh.c 1200 ⟩ ≡

```
#include <sys/resource.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
  static void p_tv(struct Shf *,int,struct timeval *,int,char *,char *);
  static void p_ts(struct Shf *,int,struct timespec *,int,char *,char *);
  extern int c_test(char **wp);      /* in c_test.c */
  extern int c_ulimit(char **wp);    /* in c_ulimit.c */
  ⟨ Bourne commands 311 ⟩
  const struct builtin shbuiltins[] ← {
      {"*=.", c_dot},
      {"*=:", c_label},
      {"[", c_test},
      {"*=break", c_brkcont},
      {"=builtin", c_builtin},
      {"*=continue", c_brkcont},
      {"*=eval", c_eval},
      {"*=exec", c_exec},
      {"*=exit", c_exitreturn},
      {"+false", c_label},
      {"*=return", c_exitreturn},
      {"*=set", c_set},
      {"*=shift", c_shift},
      {"*=times", c_times},
      {"*=trap", c_trap},
      {"+=wait", c_wait},
      {"+read", c_read},
      {"test", c_test},
      {"+true", c_label},
      {"ulimit", c_ulimit},
      {"+umask", c_umask},
      {"*=unset", c_unset},
      {"suspend", c_suspend},
      {Λ, Λ}
  };
```

1201. ⟨ c_ksh.c 1201 ⟩ ≡

```
#include <sys/stat.h>
#include <ctype.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
  static char *kill_fmt_entry(void *arg, int i, char *buf, int buflen);
  ⟨ KSH commands 36 ⟩
  const struct builtin kshbuiltins[] ← {
      {"+alias", c_alias},      /* no "=": AT&T manual wrong */
      {"+cd", c_cd},
      {"+command", c_command},
      {"echo", c_print},
      {"*=export", c_typeset},
      {"+fc", c_fc},
      {"+getopts", c_getopts},
      {"+jobs", c_jobs},
      {"+kill", c_kill},
      {"let", c_let},
      {"print", c_print},
      {"pwd", c_pwd},
      {"*=readonly", c_typeset},
      {"type", c_type},
      {"=typeset", c_typeset},
      {"+unalias", c_unalias},
      {"whence", c_whence},
      {"+bg", c_fgbg},
      {"+fg", c_fgbg},
#ifdef EMACS
      {"bind", c_bind},
#endif
      {Λ, Λ}
  };
```

1202. ⟨ Externally-linked variables 6 ⟩ +≡

```
  extern const struct builtin shbuiltins[], kshbuiltins[];
```

1203. `c_sh.c`.

⟨ Shared function declarations 4 ⟩ +≡

```
int c_label(char **);
int c_shift(char **);
int c_umask(char **);
int c_dot(char **);
int c_wait(char **);
int c_read(char **);
int c_eval(char **);
int c_trap(char **);
int c_brkcont(char **);
int c_exitreturn(char **);
int c_set(char **);
int c_unset(char **);
int c_ulimit(char **);
int c_times(char **);
int timex(struct Op *, int, volatile int *);
void timex_hook(struct Op *, char **volatile *);
int c_exec(char **);
int c_builtin(char **);
```

1204. `c_ksh.c`.

⟨ Shared function declarations 4 ⟩ +≡

```
int c_cd(char **);
int c_pwd(char **);
int c_print(char **);
int c_whence(char **);
int c_command(char **);
int c_type(char **);
int c_typeset(char **);
int c_alias(char **);
int c_unalias(char **);
int c_let(char **);
int c_jobs(char **);
int c_fgbg(char **);
int c_kill(char **);
void getopts_reset(int);
int c_getopts(char **);
int c_bind(char **);
```

1205. "`:`", "`false`" & "`true`" (arguments are ignored).

⟨ Bourne commands 311 ⟩ +≡

```
int c_label(char **wp)
{
  return wp[0][0] == 'f' ? 1 : 0;
}
```

1206. "`shift` [*number*]".

```
⟨ Bourne commands 311 ⟩ +≡
  int c_shift(char **wp)
  {
    struct Block *l ← genv¬loc;
    int n;
    int64_t val;
    char *arg;
    if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
    arg ← wp[builtin_opt.optind];
    if (arg) {
      evaluate(arg, &val, KSH_UNWIND_ERROR, false);
      n ← val;
    }
    else n ← 1;
    if (n < 0) {
      bi_errorf("%s: bad number", arg);
      return (1);
    }
    if (l¬argc < n) {
      bi_errorf("nothing to shift");
      return (1);
    }
    l¬argv[n] ← l¬argv[0];
    l¬argv += n;
    l¬argc -= n;
    return 0;
  }
```

1207. "`umask [-S] [mask]`".

```
⟨ Bourne commands 311 ⟩ +≡
  int c_umask(char **wp)
  {
    int i;
    char *cp;
    int symbolic ← 0;
    mode_t old_umask;
    int optc;
    while ((optc ← ksh_getopt(wp, &builtin_opt, "S")) ≠ −1)
      switch (optc) {
      case 'S': symbolic ← 1; break;
      case '?': return 1;
      }
    cp ← wp[builtin_opt.optind];
    if (cp ≡ Λ) {⟨ Display the current umask 1208 ⟩}
    else {
      mode_t new_umask;
      if (digit(*cp)) {⟨ Parse a numeric umask 1209 ⟩}
      else {⟨ Parse a symbolic umask 1210 ⟩}
      umask(new_umask);
    }
    return 0;
  }
```

1208. ⟨ Display the current *umask* 1208 ⟩ ≡

```
  old_umask ← umask(0);
  umask(old_umask);
  if (symbolic) {
    char buf[18];
    int j;
    old_umask ← ∼old_umask;
    cp ← buf;
    for (i ← 0; i < 3; i++) {
      *cp++ ← "ugo"[i];
      *cp++ ← '=';
      for (j ← 0; j < 3; j++)
        if (old_umask & (1 ≪ (8 − (3 * i + j)))) *cp++ ← "rwx"[j];
      *cp++ ← ',';
    }
    cp[−1] ← '\0';
    shprintf("%s\n", buf);
  }
  else shprintf("%#3.3o\n", old_umask);
```

This code is used in section 1207.

1209. ⟨ Parse a numeric *umask* 1209 ⟩ ≡

```
for (new_umask ← 0; *cp ≥ '0' ∧ *cp ≤ '7'; cp++) new_umask ← new_umask * 8 + (*cp − '0');
if (*cp) {
  bi_errorf("bad number");
  return 1;
}
```

This code is used in section 1207.

1210. ⟨ Parse a symbolic *umask* 1210 ⟩ ≡ /* symbolic format */

```
int positions, new_val;
char op;
old_umask ← umask(0);
umask(old_umask); /* in case of error */
old_umask ← ∼old_umask;
new_umask ← old_umask;
positions ← 0;
while (*cp) {⟨ Parse symbol umask characters 1211 ⟩}
if (*cp) {
  bi_errorf("bad mask");
  return 1;
}
new_umask ← ∼new_umask;
```

This code is used in section 1207.

1211. ⟨ Parse symbol umask characters 1211 ⟩ ≡

```
while (*cp ∧ strchr("augo", *cp))
  switch (*cp++) {
  case 'a': positions |= °111; break;
  case 'u': positions |= °100; break;
  case 'g': positions |= °010; break;
  case 'o': positions |= °001; break;
  }
if (¬positions) positions ← °111;      /* default is "a" */
if (¬strchr("=+-", op ← *cp)) break;
cp++;
new_val ← 0;
while (*cp ∧ strchr("rwxugoXs", *cp))
  switch (*cp++) {
  case 'r': new_val |= °4; break;
  case 'w': new_val |= °2; break;
  case 'x': new_val |= °1; break;
  case 'u': new_val |= old_umask ≫ 6; break;
  case 'g': new_val |= old_umask ≫ 3; break;
  case 'o': new_val |= old_umask ≫ 0; break;
  case 'X':
    if (old_umask & °111) new_val |= °1;
    break;
  case 's': break;      /* ignored */
  }
new_val ← (new_val & °7) * positions;
switch (op) {
case '-': new_umask &= ∼new_val; break;
case '=': new_umask ← new_val | (new_umask & ∼(positions * °7)); break;
case '+': new_umask |= new_val;
}
if (*cp ≡ ',') {
  positions ← 0;
  cp++;
}
else if (¬strchr("=+-", *cp)) break;
```

This code is used in section 1210.

1212. ". [*file* [*arg* ...]]" (differs trivially from the manpage).

```
⟨ Bourne commands 311 ⟩ +≡
  int c_dot(char **wp)
  {
    char *file, *cp;
    char **argv;
    int argc;
    int i;
    int err;
    if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
    if ((cp ← wp[builtin_opt.optind]) ≡ Λ) return 0;
    file ← search(cp, search_path, R_OK, &err);
    if (file ≡ Λ) {
      bi_errorf("%s: %s", cp, err ? strerror(err) : "not found");
      return 1;
    }
    if (wp[builtin_opt.optind + 1]) {      /* Set positional parameters? */
      argv ← wp + builtin_opt.optind;
      argv[0] ← genv→loc→argv[0];      /* preserve $0 */
      for (argc ← 0; argv[argc + 1]; argc++) ;
    }
    else {
      argc ← 0;
      argv ← Λ;
    }
    i ← Include(file, argc, argv, 0);
    if (i < 0) {      /* should not happen */
      bi_errorf("%s: %s", cp, strerror(errno));
      return 1;
    }
    return i;
  }
```

1213. "`wait` [*job* ...]".

```
⟨ Bourne commands 311 ⟩ +≡
  int c_wait(char **wp)
  {
    int rv ← 0;
    int sig;
    if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
    wp += builtin_opt.optind;
    if (*wp ≡ Λ) {
      while (waitfor(Λ, &sig) ≥ 0) ;
      rv ← sig;
    }
    else {
      for ( ; *wp; wp++) rv ← waitfor(*wp, &sig);
      if (rv < 0) rv ← sig ? sig : 127;      /* magic exit code: bad job-id */
    }
    return rv;
  }
```

1214. "`read` [`-prsu`[*n*]] [*parameter ...*]". The file is re-opened without buffering since we can't necessarily seek backwards on non-regular files.

AT&T ksh says it prints the prompt on the file descriptor if it's open for writing and is a tty, but it doesn't do it (it also doesn't check the interactive flag, as is indicated in the Kornshell book).

```
⟨ Bourne commands 311 ⟩ +≡
  int c_read(char **wp)
  {
    int c ← 0;
    int expand ← 1, savehist ← 0;
    int expanding;
    int ecode ← 0;
    char *cp;
    int fd ← 0;
    struct Shf *shf;
    int optc;
    const char *emsg;
    XString cs, xs;
    struct tbl *vp;
    char *xp ← Λ;
    while ((optc ← ksh_getopt(wp, &builtin_opt, "prsu,")) ≠ −1)
      switch (optc) {
      case 'p':
        if ((fd ← coproc_getfd(R_OK, &emsg)) < 0) {
          bi_errorf("-p: %s", emsg);
          return 1;
        }
        break;
      case 'r': expand ← 0;
        break;
      case 's': savehist ← 1;
        break;
      case 'u':
        if (¬*(cp ← builtin_opt.optarg)) fd ← 0;
        else if ((fd ← check_fd(cp, R_OK, &emsg)) < 0) {
          bi_errorf("-u: %s: %s", cp, emsg);
          return 1;
        }
        break;
      case '?': return 1;
      }
    wp += builtin_opt.optind;
    if (*wp ≡ Λ) *−−wp ← "REPLY";
    shf ← shf_reopen(fd, SHF_RD | SHF_INTERRUPT | can_seek(fd), shl_spare);
    if ((cp ← strchr(*wp, '?')) ≠ Λ) {
      *cp ← 0;
      if (isatty(fd)) {
        shellf("%s", cp + 1);
      }
    }
    ⟨ Read variables from the file descriptor 1215 ⟩
  }
```

1215. This part began with a comment before the disabled call to "*coproc_readw_close*(*fd*)": If we are reading from the co-process for the first time, make sure the other side of the pipe is closed first. This allows the detection of EOF.

This is not compatible with AT&T ksh—the file descriptor is kept so another co-process can be started with the same output, however this means EOF can't be detected. This is why it is closed here. If this call is removed, remove the EOF check below, too.

The EOF check was not removed, instead it has this comment: if this is the co-process file descriptor, close the file descriptor (we can get EOF if and only if all processes have died, ie. *coproc.njobs* is 0 and the pipe is closed).

```
⟨ Read variables from the file descriptor 1215 ⟩ ≡
  if (savehist) Xinit(xs, xp, 128, ATEMP);
  expanding ← 0;
  Xinit(cs, cp, 128, ATEMP);
  for ( ; *wp ≠ Λ; wp++) {
    for (cp ← Xstring(cs, cp); ; ) {⟨ Read a string into the next variable 1216 ⟩}
    if (¬wp[1])      /* strip trailing $IFS white space from the last variable */
      while (Xlength(cs, cp) ∧ ctype(cp[−1], C_IFS) ∧ ctype(cp[−1], C_IFSWS)) cp−−;
    Xput(cs, cp, '\0');
    vp ← global(*wp);
    if (vp→flag & RDONLY) {      /* Must be done before setting export. */
      shf_flush(shf);
      bi_errorf("%s is read only", *wp);
      return 1;
    }
    if (Flag(FEXPORT)) typeset(*wp, EXPORT, 0, 0, 0);
    if (¬setstr(vp, Xstring(cs, cp), KSH_RETURN_ERROR)) {
      shf_flush(shf);
      return 1;
    }
  }
  shf_flush(shf);
  if (savehist) {
    Xput(xs, xp, '\0');
    source→line++;
    histsave(source→line, Xstring(xs, xp), 1);
    Xfree(xs, xp);
  }
  if (c ≡ EOF ∧ ¬ecode) coproc_read_close(fd);
  return ecode ? ecode : c ≡ EOF;
```

This code is used in section 1214.

1216. ⟨ Read a string into the next variable 1216 ⟩ ≡
```
if (c ≡ '\n' ∨ c ≡ EOF) break;
while (1) {⟨ Read the next character 1217 ⟩}
if (savehist) {
  Xcheck(xs, xp);
  Xput(xs, xp, c);
}
Xcheck(cs, cp);
if (expanding) {⟨ Read another line and continue 1218 ⟩}
if (expand ∧ c ≡ '\\') {⟨ Prepare to read another line and continue 1219 ⟩}
if (c ≡ '\n' ∨ c ≡ EOF) break;
if (ctype(c, C_IFS)) {
  if (Xlength(cs, cp) ≡ 0 ∧ ctype(c, C_IFSWS)) continue;
  if (wp[1]) break;
}
Xput(cs, cp, c);
```
This code is used in section 1215.

1217. If `read` was interrupted and the offending signal was one that would normally kill a process, pretend `read` was killed.

⟨ Read the next character 1217 ⟩ ≡
```
c ← shf_getc(shf);
if (c ≡ '\0') continue;
if (c ≡ EOF ∧ shf_error(shf) ∧ shf¬errno_ ≡ EINTR) {
  ecode ← fatal_trap_check( );
  if (¬ecode) {      /* non fatal (eg. SIGCHLD), carry on */
    shf_clearerr(shf);
    continue;
  }
}
break;
```
This code is used in section 1216.

1218. ⟨ Read another line and **continue** 1218 ⟩ ≡
```
expanding ← 0;
if (c ≡ '\n') {
  c ← 0;
  if (Flag(FTALKING_I) ∧ isatty(fd)) {      /* in case this is called from .profile or $ENV */
    set_prompt(PS2);
    pprompt(prompt, 0);
  }
}
else if (c ≠ EOF) Xput(cs, cp, c);
continue;
```
This code is used in section 1216.

1219. ⟨ Prepare to read another line and **continue** 1219 ⟩ ≡
```
expanding ← 1;
continue;
```
This code is used in section 1216.

1220. "`eval` *command ...*". If `FPOSIX`: Handle the case where the command is empty due to a failed command substitution, eg. `eval "$(false)"`. In this case *shell* will not set/change *exstat* (because the compiled tree is empty), so will use this value. *subst_exstat* is cleared in *execute* so should be 0 if there were no substitutions. A strict reading of POSIX says we don't do this although it is traditionally done[1].

```
⟨ Bourne commands 311 ⟩ +≡
  int c_eval(char **wp)
  {
    struct Source *s;
    struct Source *saves ← source;
    int savef;
    int rv;
    if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
    s ← pushs(SWORDS, ATEMP);
    s¬u.strv ← wp + builtin_opt.optind;
    if (¬Flag(FPOSIX)) { exstat ← subst_exstat; }
    savef ← Flag(FERREXIT);
    Flag(FERREXIT) ← 0;
    rv ← shell(s, false);
    Flag(FERREXIT) ← savef;
    source ← saves;
    afree(s, ATEMP);
    return (rv);
  }
```

[1] *from 1003.2-1992*:

3.9.1: Simple Commands ... If there is a command name, execution shall continue as described in 3.9.1.1. If there is no command name, but the command contained a command substitution, the command shall complete with the exit status of the last command substitution

3.9.1.1: Command Search and Execution ...(1)...(a) If the command name matches the name of a special built-in utility, that special built-in utility shall be invoked.

3.14.5: Eval ... If there are no arguments, or only null arguments, eval shall return an exit status of zero.

1221. "`trap` [*handler signal ...*]". Use case sensitive lookup for the first argument (in the call to *gettrap*) so the command "`exit`" isn't confused with the pseudo-signal "`EXIT`".

```
⟨ Bourne commands 311 ⟩ +≡
  int c_trap(char **wp)
  {
    int i;
    char *s;
    Trap *p;
    if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
    wp += builtin_opt.optind;
    if (*wp ≡ Λ) {
      for (p ← sigtraps, i ← NSIG + 1; --i ≥ 0; p++) {
        if (p→trap ≠ Λ) {
          shprintf("trap␣--␣");
          print_value_quoted(p→trap);
          shprintf("␣%s\n", p→name);
        }
      }
      return 0;
    }
    s ← (gettrap(*wp, false) ≡ Λ) ? *wp++ : Λ;      /* get command */
    if (s ≠ Λ ∧ s[0] ≡ '-' ∧ s[1] ≡ '\0') s ← Λ;
    while (*wp ≠ Λ) {      /* set/clear traps */
      p ← gettrap(*wp++, true);
      if (p ≡ Λ) {
        bi_errorf("bad␣signal␣%s", wp[−1]);
        return 1;
      }
      settrap(p, s);
    }
    return 0;
  }
```

1222. "`exit|return` [*status*]". These are the same unless `return` has a previous scope to return to, and only then is *how* overridden with `LRETURN`.

```
#define STOP_RETURN(t) ((t) ≡ E_FUNC ∨ (t) ≡ E_INCL)  /* Do returns stop at environment t? */
⟨ Bourne commands 311 ⟩ +≡
  int c_exitreturn(char **wp)
  {
    int how ← LEXIT;
    int n;
    char *arg;
    if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
    arg ← wp[builtin_opt.optind];
    if (arg) {
      if (¬getn(arg, &n)) {
        exstat ← 1;
        warningf(true, "%s: bad number", arg);
      }
      else exstat ← n;
    }
    if (wp[0][0] ≡ 'r') {  /* return */
      struct Env *ep;
      for (ep ← genv; ep; ep ← ep→oenv)
        if (STOP_RETURN(ep→type)) {
          how ← LRETURN;
          break;
        }
    }
    if (how ≡ LEXIT ∧ ¬really_exit ∧ j_stopped_running()) {
      really_exit ← 1;
      how ← LSHELL;
    }
    quitenv(Λ);  /* get rid of any I/O redirections */
    unwind (how);
    return 0;  /* NOTREACHED */
  }
```

1223. "`break|continue` [*level*]".

```
#define STOP_BRKCONT(t) ((t) ≡ E_NONE ∨ (t) ≡ E_PARSE ∨ (t) ≡ E_FUNC ∨ (t) ≡ E_INCL)
⟨ Bourne commands 311 ⟩ +≡
  int c_brkcont(char **wp)
  {
    int n, quit;
    struct Env *ep, *last_ep ← Λ;
    char *arg;
    if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
    arg ← wp[builtin_opt.optind];
    if (¬arg) n ← 1;
    else if (¬bi_getn(arg, &n)) return 1;
    quit ← n;
    if (quit ≤ 0) {      /* AT&T ksh does this for non-interactive shells only - weird */
      bi_errorf("%s: bad value", arg);
      return 1;
    }
    for (ep ← genv; ep ∧ ¬STOP_BRKCONT(ep→type); ep ← ep→oenv)
        /* Stop at E_NONE, E_PARSE, E_FUNC or E_INCL */
      if (ep→type ≡ E_LOOP) {
        if (--quit ≡ 0) break;
        ep→flags |= EF_BRKCONT_PASS;
        last_ep ← ep;
      }
    if (quit) {
      if (n ≡ quit) {      /* AT&T ksh doesn't print a message, just does what it can. We print a message
            because it helps in debugging scripts, but don't generate an error (ie. keep going). */
        warningf(true, "%s: cannot %s", wp[0], wp[0]);
        return 0;
      }
      if (last_ep)      /* POSIX says if n is too big, the last enclosing loop shall be used. */
        last_ep→flags &= ∼EF_BRKCONT_PASS;
      warningf(true, "%s: can only %s %d level(s)", wp[0], wp[0], n − quit);
        /* It doesn't say to print an error but we do anyway because the user messed up. */
    }
    unwind (*wp[0] ≡ 'b' ? LBREAK : LCONTIN);      /* NOTREACHED */
  }
```

1224. "`set` [...]". POSIX says `set`'s exit status is 0 but old scripts that use getopt(1) use the construct "`set -- `getopt ab:c "$@"``", which assumes the exit value set will be that of the `` ` ... ` `` (*subst_exstat* is cleared in *execute* so that it will be 0 if there are no command substitutions).

```
⟨ Bourne commands 311 ⟩ +≡
  int c_set(char **wp)
  {
    int argi, setargs;
    struct Block *l ← genv→loc;
    char **owp ← wp;
    if (wp[1] ≡ Λ) {
      static const char *const args[] ← {"set", "-", Λ};
      return c_typeset((char **) args);
    }
    argi ← parse_args(wp, OF_SET, &setargs);
    if (argi < 0) return 1;
    if (setargs) {      /* set $# and $*/$@ */
      owp ← wp += argi − 1;
      wp[0] ← l→argv[0];      /* save $0 */
      while (*++wp ≠ Λ) *wp ← str_save(*wp, &l→area);
      l→argc ← wp − owp − 1;
      l→argv ← areallocarray(Λ, l→argc + 2, sizeof(char *), &l→area);
      for (wp ← l→argv; (*wp++ ← *owp++) ≠ Λ; ) ;
    }
    return Flag(FPOSIX) ? 0 : subst_exstat;
  }
```

1225. "`unset` [`-fv`] [*paramater ...*]".

```
⟨ Bourne commands 311 ⟩ +≡
  int c_unset(char **wp)
  {
    char *id;
    int optc, unset_var ← 1;
    while ((optc ← ksh_getopt(wp, &builtin_opt, "fv")) ≠ −1)
      switch (optc) {
      case 'f': unset_var ← 0; break;
      case 'v': unset_var ← 1; break;
      case '?': return 1;
      }
    wp += builtin_opt.optind;
    for ( ; (id ← *wp) ≠ Λ; wp++)
      if (unset_var) {      /* unset variable */
        struct tbl *vp ← global(id);
        if ((vp→flag & RDONLY)) {
          bi_errorf("%s is read only", vp→name);
          return 1;
        }
        unset(vp, strchr(id, '[') ? 1 : 0);
      }
      else {      /* unset function */
        Define(id, Λ);
      }
    return 0;
  }
```

1226. "`times`".

```
⟨ Bourne commands 311 ⟩ +≡
  int c_times(char **wp)
  {
    struct rusage usage;
    (void) getrusage(RUSAGE_SELF, &usage);
    p_tv(shl_stdout, 0, &usage.ru_utime, 0, Λ, " ");
    p_tv(shl_stdout, 0, &usage.ru_stime, 0, Λ, "\n");
    (void) getrusage(RUSAGE_CHILDREN, &usage);
    p_tv(shl_stdout, 0, &usage.ru_utime, 0, Λ, " ");
    p_tv(shl_stdout, 0, &usage.ru_stime, 0, Λ, "\n");
    return 0;
  }
```

1227. "exec" without arguments. The case with arguments is handled in *comexec*. For ksh keep any file descriptors ¿ 2 private, for sh let them be (POSIX says what happens is unspecified and the bourne shell keeps them open).

```
⟨ Bourne commands 311 ⟩ +≡
  int c_exec(char **wp)
  {
    int i;
    if (genv→savefd ≠ Λ) {      /* make sure redirects stay in place */
      for (i ← 0; i < NUFILE; i++) {
        if (genv→savefd[i] > 0) close(genv→savefd[i]);
        if (¬Flag(FSH) ∧ i > 2 ∧ genv→savefd[i]) fcntl(i, F_SETFD, FD_CLOEXEC);
      }
      genv→savefd ← Λ;
    }
    return 0;
  }
```

1228. "suspend".

```
⟨ Bourne commands 311 ⟩ +≡
  static int c_suspend(char **wp)
  {
    if (wp[1] ≠ Λ) {
      bi_errorf("too␣many␣arguments");
      return 1;
    }
    if (Flag(FLOGIN)) {      /* Can't suspend an orphaned process group. */
      pid_t parent ← getppid();
      if (getpgid(parent) ≡ getpgid(0) ∨ getsid(parent) ≠ getsid(0)) {
        bi_errorf("can't␣suspend␣a␣login␣shell");
        return 1;
      }
    }
    j_suspend();
    return 0;
  }
```

1229. "builtin [*command* [*arg* ...]]". This is a dummy function—builtin is a special case in *comexec*.

```
⟨ Bourne commands 311 ⟩ +≡
  int c_builtin(char **wp)
  {
    return 0;
  }
```

1230. "cd [-LP] [*directory*]" or "cd [-LP] [*old new*]".

```
⟨ KSH commands 36 ⟩ +≡
  int c_cd(char **wp)
  {
    int optc;
    int physical ← Flag(FPHYSICAL);
    int cdnode;      /* was a node from $CDPATH added in? */
    int printpath ← 0;      /* print where we changed to? */
    int rval;
    struct tbl *pwd_s, *oldpwd_s;
    XString xs;
    char *xp;
    char *dir, *try, *pwd;
    int phys_path;
    char *cdpath;
    char *fdir ← Λ;
    while ((optc ← ksh_getopt(wp, &builtin_opt, "LP")) ≠ −1)
      switch (optc) {
      case 'L': physical ← 0; break;
      case 'P': physical ← 1; break;
      case '?': return 1;
      }
    wp += builtin_opt.optind;
    if (Flag(FRESTRICTED)) {
      bi_errorf("restricted␣shell␣-␣can't␣cd");
      return 1;
    }
    pwd_s ← global("PWD");
    oldpwd_s ← global("OLDPWD");
    if (¬wp[0]) {⟨ cd with zero arguments—go $HOME 1231 ⟩}
    else if (¬wp[1]) {⟨ cd with one argument—go there 1232 ⟩}
    else if (¬wp[2]) {⟨ cd with two arguments—substitute in $PWD 1233 ⟩}
    else {
      bi_errorf("too␣many␣arguments");
      return 1;
    }
    ⟨ Change to the new directory 1234 ⟩
    ⟨ Save the new directory 1235 ⟩
    afree(fdir, ATEMP);
    return 0;
  }
```

1231.

```
⟨ cd with zero arguments—go $HOME 1231 ⟩ ≡
  if ((dir ← str_val(global("HOME"))) ≡ null) {
    bi_errorf("no␣home␣directory␣(HOME␣not␣set)");
    return 1;
  }
```

This code is used in section 1230.

1232. One argument: "-" or a directory.

```
⟨ cd with one argument—go there 1232 ⟩ ≡
  dir ← wp[0];
  if (strcmp(dir, "-") ≡ 0) {
    dir ← str_val(oldpwd_s);
    if (dir ≡ null) {
      bi_errorf("no␣OLDPWD");
      return 1;
    }
    printpath++;
  }
```

This code is used in section 1230.

1233. Substitute argument one in $PWD for argument two. If the first substitution fails because the cd fails we could try to find another substitution. For now we don't.

```
⟨ cd with two arguments—substitute in $PWD 1233 ⟩ ≡
  int ilen, olen, nlen, elen;
  char *cp;
  if (¬current_wd[0]) {
    bi_errorf("don't␣know␣current␣directory");
    return 1;
  }
  if ((cp ← strstr(current_wd, wp[0])) ≡ Λ) {
    bi_errorf("bad␣substitution");
    return 1;
  }
  ilen ← cp − current_wd;
  olen ← strlen(wp[0]);
  nlen ← strlen(wp[1]);
  elen ← strlen(current_wd + ilen + olen) + 1;
  fdir ← dir ← alloc(ilen + nlen + elen, ATEMP);
  memcpy(dir, current_wd, ilen);
  memcpy(dir + ilen, wp[1], nlen);
  memcpy(dir + ilen + nlen, current_wd + ilen + olen, elen);
  printpath++;
```

This code is used in section 1230.

1234. ⟨ Change to the new directory 1234 ⟩ ≡

```
Xinit(xs, xp, PATH_MAX, ATEMP);
xp ← Λ;  /* xp will have a bogus value after make_path—set it to Λ to crash if it's used (it is not) */
cdpath ← str_val(global("CDPATH"));
do {
  cdnode ← make_path(current_wd, dir, &cdpath, &xs, &phys_path);
  if (physical) rval ← chdir(try ← Xstring(xs, xp) + phys_path);
  else {
    simplify_path(Xstring(xs, xp));
    rval ← chdir(try ← Xstring(xs, xp));
  }
} while (rval ≡ −1 ∧ cdpath ≠ Λ);
if (rval ≡ −1) {
  if (cdnode) bi_errorf("%s: bad directory", dir);
  else bi_errorf("%s - %s", try, strerror(errno));
  afree(fdir, ATEMP);
  return 1;
}
```

This code is used in section 1230.

1235. ⟨ Save the new directory 1235 ⟩ ≡

```
flushcom(0);  /* Clear out tracked aliases with relative paths */
if (current_wd[0])  /* Set $OLDPWD (unsetting $OLDPWD doesn't disable this in AT&T ksh) */
  setstr(oldpwd_s, current_wd, KSH_RETURN_ERROR);  /* Ignore failure (ie. if readonly or integer) */
if (Xstring(xs, xp)[0] ≠ '/') { pwd ← Λ; }
else if (¬physical ∨ ¬(pwd ← get_phys_path(Xstring(xs, xp)))) pwd ← Xstring(xs, xp);
if (pwd) {  /* Set $PWD */
  char *ptmp ← pwd;
  set_current_wd(ptmp);
  setstr(pwd_s, ptmp, KSH_RETURN_ERROR);  /* Ignore failure (ie. if readonly or integer) */
}
else {  /* XXX unset $PWD? */
  set_current_wd(null);
  pwd ← Xstring(xs, xp);
}
if (printpath ∨ cdnode) shprintf("%s\n", pwd);
```

This code is used in section 1230.

1236. "pwd [-LP]".

```
⟨ KSH commands 36 ⟩ +≡
  int c_pwd(char **wp)
  {
    int optc;
    int physical ← Flag(FPHYSICAL);
    char *p, *freep ← Λ;
    while ((optc ← ksh_getopt(wp, &builtin_opt, "LP")) ≠ −1)
      switch (optc) {
      case 'L': physical ← 0; break;
      case 'P': physical ← 1; break;
      case '?': return 1;
      }
    wp += builtin_opt.optind;
    if (wp[0]) {
      bi_errorf("too␣many␣arguments");
      return 1;
    }
    p ← current_wd[0] ? (physical ? get_phys_path(current_wd) : current_wd) : Λ;
    if (p ∧ access(p, R_OK) ≡ −1) p ← Λ;
    if (¬p) {
      freep ← p ← ksh_get_wd(Λ, 0);
      if (¬p) {
        bi_errorf("can't␣get␣current␣directory␣-␣%s", strerror(errno));
        return 1;
      }
    }
    shprintf("%s\n", p);
    afree(freep, ATEMP);
    return 0;
  }
```

1237. "`echo` [`-Een`] [*arg* ...]" or "`print` ...". It was noted that `'\007'` is more portable than `'\a'` due to old C compilers which are quite likely to have been fixed by now.

```
#define PO_NL BIT(0) /* print newline */
#define PO_EXPAND BIT(1) /* expand backslash sequences */
#define PO_PMINUSMINUS BIT(2) /* print a "--" argument */
#define PO_HIST BIT(3) /* print to history instead of stdout */
#define PO_COPROC BIT(4) /* printing to coprocess: block SIGPIPE */
```

⟨ KSH commands 36 ⟩ +≡

```
int c_print(char **wp)
{
  int fd ← 1;
  int flags ← PO_EXPAND | PO_NL;
  char *s;
  const char *emsg;
  XString xs;
  char *xp;
  if (wp[0][0] ≡ 'e') {⟨ Scan echo arguments 1238 ⟩}
  else {⟨ Scan echo arguments 1238 ⟩}
  Xinit(xs, xp, 128, ATEMP);
  while (*wp ≠ Λ) {⟨ Copy and expand characters to print 1240 ⟩}
  if (flags & PO_NL) Xput(xs, xp, '\n');
  if (flags & PO_HIST) {⟨ "Print" to the history 1241 ⟩}
  else {⟨ Print to the terminal 1242 ⟩}
  return 0;
}
```

1238. A compromise between SysV and BSD echo commands: escape sequences are enabled by default and `-n`, `-e` and `-E` are recognized if they appear in arguments with no illegal options (ie. `echo -nq` will print "-nq").

Different from SysV echo since options are recognized, different from BSD echo since escape sequences are enabled by default.

```
⟨ Scan echo arguments 1238 ⟩ ≡
  int nflags ← flags;
  wp += 1;
  if (Flag(FPOSIX)) {
    if (*wp ∧ strcmp(*wp, "-n") ≡ 0) {
      flags &= ∼PO_NL;
      wp++;
    }
  }
  else {
    while ((s ← *wp) ∧ *s ≡ '-' ∧ s[1]) {
      while (*++s)
        if (*s ≡ 'n') nflags &= ∼PO_NL;
        else if (*s ≡ 'e') nflags |= PO_EXPAND;
        else if (*s ≡ 'E') nflags &= ∼PO_EXPAND;
        else break;      /* bad option: don't use nflags, print argument */
      if (*s) break;
      wp++;
      flags ← nflags;
    }
  }
```

See also section 1239.

This code is used in section 1237.

1239. ⟨ Scan `echo` arguments 1238 ⟩ +≡

```
int optc;
const char *options ← "Rnprsu,";
while ((optc ← ksh_getopt(wp, &builtin_opt, options)) ≠ −1)
  switch (optc) {
  case 'R':     /* fake BSD echo command */
    flags |= PO_PMINUSMINUS;
    flags &= ∼PO_EXPAND;
    options ← "ne";
    break;
  case 'e': flags |= PO_EXPAND; break;
  case 'n': flags &= ∼PO_NL; break;
  case 'p':
    if ((fd ← coproc_getfd(W_OK, &emsg)) < 0) {
      bi_errorf("-p: %s", emsg);
      return 1;
    }
    break;
  case 'r': flags &= ∼PO_EXPAND; break;
  case 's': flags |= PO_HIST; break;
  case 'u':
    if (¬*(s ← builtin_opt.optarg)) fd ← 0;
    else if ((fd ← check_fd(s, W_OK, &emsg)) < 0) {
      bi_errorf("-u: %s: %s", s, emsg);
      return 1;
    }
    break;
  case '?': return 1;
  }
if (¬(builtin_opt.info & GI_MINUSMINUS)) {     /* treat a lone - like − */
  if (wp[builtin_opt.optind] ∧ strcmp(wp[builtin_opt.optind], "-") ≡ 0) builtin_opt.optind ++;
}
else if (flags & PO_PMINUSMINUS) builtin_opt.optind −−;
wp += builtin_opt.optind;
```

1240. ⟨ Copy and expand characters to print 1240 ⟩ ≡

```
int c;
s ← *wp;
while ((c ← *s++) ≠ '\0') {
  Xcheck(xs, xp);
  if ((flags & PO_EXPAND) ∧ c ≡ '\\') {
    int i;
    switch ((c ← *s++)) {
    case 'a': c ← '\007'; break;
    case 'b': c ← '\b'; break;
    case 'c': flags &= ∼PO_NL; continue;      /* AT&T brain damage */
    case 'f': c ← '\f'; break;
    case 'n': c ← '\n'; break;
    case 'r': c ← '\r'; break;
    case 't': c ← '\t'; break;
    case 'v': c ← 0x0B; break;
    case '0':      /* Look for an octal number—three digits not counting the leading 0. */
      c ← 0;
      for (i ← 0; i < 3; i++) {
        if (*s ≥ '0' ∧ *s ≤ '7') c ← c * 8 + *s++ − '0';
        else break;
      }
      break;
    case '\0': s−−; c ← '\\'; break;
    case '\\': break;
    default: Xput(xs, xp, '\\');
    }
  }
  Xput(xs, xp, c);
}
if (*++wp ≠ Λ) Xput(xs, xp, '␣');
```

This code is used in section 1237.

1241. ⟨ "Print" to the history 1241 ⟩ ≡

```
Xput(xs, xp, '\0');
source¬line ++;
histsave(source¬line, Xstring(xs, xp), 1);
Xfree(xs, xp);
```

This code is used in section 1237.

1242. Ensure we aren't killed by a SIGPIPE while writing to a co-process. AT&T ksh doesn't seem to do this (seems to just check that the co-process is alive, which is not enough).

```
⟨ Print to the terminal 1242 ⟩ ≡
  int n, len ← Xlength(xs, xp);
  int opipe ← 0;
  if (coproc.write ≥ 0 ∧ coproc.write ≡ fd) {
    flags |= PO_COPROC;
    opipe ← block_pipe( );
  }
  for (s ← Xstring(xs, xp); len > 0; ) {
    n ← write(fd, s, len);
    if (n ≡ −1) {
      if (flags & PO_COPROC) restore_pipe(opipe);
      if (errno ≡ EINTR) {      /* allow user to Ctrl-C out */
        intrcheck( );
        if (flags & PO_COPROC) opipe ← block_pipe( );
        continue;
      }      /* doesn't really make sense & could break scripts (print -p generates an error message). */
             /* if (errno ≡ EPIPE) coproc_write_close(fd); */
      return 1;
    }
    s += n;
    len −= n;
  }
  if (flags & PO_COPROC) restore_pipe(opipe);
```

This code is used in section 1237.

1243. "`whence [-pv] [name ...]`" and variants "`command ...`" & "`type ...`".

```
⟨ KSH commands 36 ⟩ +≡
  int c_whence(char **wp)
  {
    struct tbl *tp;
    char *id;
    int pflag ← 0, vflag ← 0, Vflag ← 0;      /* Note v vs. V */
    int ret ← 0;
    int optc;
    int iam_whence;
    int fcflags;
    const char *options;
    switch (wp[0][0]) {
    case 'c': iam_whence ← 0; options ← "pvV"; break;      /* command */
    case 't': vflag ← 1;      /* type (FALLTHROUGH) */
    case 'w': iam_whence ← 1; options ← "pv"; break;      /* whence */
    default: bi_errorf("builtin not handled by %s", __func__);
      return 1;
    }
    while ((optc ← ksh_getopt(wp, &builtin_opt, options)) ≠ -1)
      switch (optc) {
      case 'p': pflag ← 1; break;
      case 'v': vflag ← 1; break;
      case 'V': Vflag ← 1; break;
      case '?': return 1;
      }
    wp += builtin_opt.optind;
    fcflags ← FC_BI | FC_PATH | FC_FUNC;
    if (¬iam_whence) {      /* Note that -p on its own is dealt with in comexec */
      if (pflag) fcflags |= FC_DEFPATH;      /* Convert command options to whence options. Note that
            command -pV and command -pv use a different path search than whence -v or whence -pv;
            this should be considered a feature. */
      vflag ← Vflag;
    }
    else if (pflag) fcflags &= ∼(FC_BI | FC_FUNC);
    while ((vflag ∨ ret ≡ 0) ∧ (id ← *wp++) ≠ Λ) {⟨ Display the provenance of a name 1244 ⟩}
    return ret;
  }
```

1244. ⟨ Display the provenance of a name 1244 ⟩ ≡

```
tp ← Λ;
if (¬iam_whence ∨ ¬pflag) tp ← ktsearch(&keywords, id, hash(id));
if (¬tp ∧ (¬iam_whence ∨ ¬pflag)) {
  tp ← ktsearch(&aliases, id, hash(id));
  if (tp ∧ ¬(tp→flag & ISSET)) tp ← Λ;
}
if (¬tp) tp ← findcom(id, fcflags);
if (vflag ∨ (tp→type ≠ CALIAS ∧ tp→type ≠ CEXEC ∧ tp→type ≠ CTALIAS)) shprintf("%s", id);
switch (tp→type) {
case CKEYWD:
  if (vflag) shprintf("␣is␣a␣reserved␣word");
  break;
case CALIAS: ⟨ Display the provenance of an alias 1245 ⟩ break;
case CFUNC: ⟨ Display the provenance of a function 1246 ⟩ break;
case CSHELL: ⟨ Display the provenance of a shell built-in 1247 ⟩ break;
case CTALIAS: case CEXEC: ⟨ Display the provenance of an executable 1248 ⟩ break;
default: shprintf("%s␣is␣*GOK*", id);
  break;
}
if (vflag ∨ ¬ret) shprintf("\n");
```

This code is used in section 1243.

1245. ⟨ Display the provenance of an alias 1245 ⟩ ≡

```
if (vflag) shprintf("␣is␣an␣%salias␣for␣", (tp→flag & EXPORT) ? "exported␣" : "");
if (¬iam_whence ∧ ¬vflag) shprintf("alias␣%s=", id);
print_value_quoted(tp→val.s);
```

This code is used in section 1244.

1246. ⟨ Display the provenance of a function 1246 ⟩ ≡

```
if (vflag) {
  shprintf("␣is␣a");
  if (tp→flag & EXPORT) shprintf("n␣exported");
  if (tp→flag & TRACE) shprintf("␣traced");
  if (¬(tp→flag & ISSET)) {
    shprintf("␣undefined");
    if (tp→u.fpath) shprintf("␣(autoload␣from␣%s)", tp→u.fpath);
  }
  shprintf("␣function");
}
```

This code is used in section 1244.

1247. ⟨ Display the provenance of a shell built-in 1247 ⟩ ≡

```
if (vflag) shprintf("␣is␣a%s␣shell␣builtin", (tp→flag & SPEC_BI) ? "␣special" : "");
```

This code is used in section 1244.

1248. ⟨ Display the provenance of an executable 1248 ⟩ ≡

```
if (tp→flag & ISSET) {
  if (vflag) {
    shprintf("␣is␣");
    if (tp→type ≡ CTALIAS)
      shprintf("a␣tracked␣%salias␣for␣", (tp→flag & EXPORT) ? "exported␣" : "");
  }
  shprintf("%s", tp→val.s);
}
else {
  if (vflag) shprintf("␣not␣found");
  ret ← 1;
}
```

This code is used in section 1244.

1249. The built-in commands `command`, `type` and `whence` are distinguished in *findcom* by the function that is pointed to, so explicit front-ends to *c_whence* are required.

⟨ KSH commands 36 ⟩ +≡

```
int c_command(char **wp)
{ return c_whence(wp); }
int c_type(char **wp)
{ return c_whence(wp); }
```

1250. "`typeset` ...". AT&T ksh seems to have 0-9 as options, which are multiplied to get a number that is used with `-L`, `-R`, `-Z` or `-i` (eg, `-1R2` sets right justify in a field of 12). This allows options to be grouped in an order (eg, `-Lu12`), but disallows `-i8 -L3` and does not allow the number to be specified as a separate argument

Here, the number must follow the `L/R/Z/i` option but is optional (see the `#` kludge in *ksh_getopt*).

⟨ KSH commands 36 ⟩ +≡

```
int c_typeset(char **wp)
{
  struct Block *l;
  struct tbl *vp, **p;
  int fset ← 0, fclr ← 0, thing ← 0, func ← 0, local ← 0, pflag ← 0;
  const char *options ← "L#R#UZ#fi#lprtux";
  char *fieldstr, *basestr;
  int field, base, optc, flag;
  ⟨ Parse options to typeset &c. 1251 ⟩
  if (wp[builtin_opt.optind]) {⟨ Set variables & attributes and return 1255 ⟩}
  flag ← fset | fclr;      /* no difference at this point.. */
  if (func) {⟨ List all user functions 1256 ⟩}
  else {⟨ List all variables 1257 ⟩}
  return 0;
}
```

1251. Determine which command is being called.

```
⟨ Parse options to typeset &c. 1251 ⟩ ≡
  switch (**wp) {
  case 'e': fset |= EXPORT; options ← "p"; break;     /* export */
  case 'r': fset |= RDONLY; options ← "p"; break;     /* readonly */
  case 's': break;     /* set—called with "typeset -" */
  case 't': local ← 1; break;     /* typeset */
  }
```

See also sections 1252, 1253, and 1254.

This code is used in section 1250.

1252. Scan the arguments for option flags.

```
⟨ Parse options to typeset &c. 1251 ⟩ +≡
  fieldstr ← basestr ← Λ;
  builtin_opt.flags |= GF_PLUSOPT;
  while ((optc ← ksh_getopt(wp, &builtin_opt, options)) ≠ −1) {
    flag ← 0;
    switch (optc) {
    case 'L': flag ← LJUST; fieldstr ← builtin_opt.optarg; break;
    case 'R': flag ← RJUST; fieldstr ← builtin_opt.optarg; break;
    case 'U': flag ← INT_U; break;     /* AT&T ksh uses -u which conflicts with upper case. */
    case 'Z': flag ← ZEROFIL; fieldstr ← builtin_opt.optarg; break;
    case 'f': func ← 1; break;
    case 'i': flag ← INTEGER; basestr ← builtin_opt.optarg; break;
    case 'l': flag ← LCASEV; break;
    case 'p': pflag ← 1; break;     /* default action; for POSIX/compatibility with ksh93 */
    case 'r': flag ← RDONLY; break;
    case 't': flag ← TRACE; break;
    case 'u': flag ← UCASEV_AL; break;     /* upper case/autoload */
    case 'x': flag ← EXPORT; break;
    case '?': return 1;
    }
    if (builtin_opt.info & GI_PLUS) {
      fclr |= flag;
      fset &= ~flag;
      thing ← '+';
    }
    else {
      fset |= flag;
      fclr &= ~flag;
      thing ← '-';
    }
  }
```

1253. After scanning, validation.

```
⟨ Parse options to typeset &c. 1251 ⟩ +≡
  field ← 0;
  if (fieldstr ∧ ¬bi_getn(fieldstr, &field)) return 1;
  base ← 0;
  if (basestr ∧ ¬bi_getn(basestr, &base)) return 1;
  if (¬(builtin_opt.info & GI_MINUSMINUS) ∧
        wp[builtin_opt.optind] ∧
        (wp[builtin_opt.optind][0] ≡ '-' ∨ wp[builtin_opt.optind][0] ≡ '+') ∧
        wp[builtin_opt.optind][1] ≡ '\0') {
    thing ← wp[builtin_opt.optind][0];
    builtin_opt.optind ++;
  }
  if (func ∧ ((fset | fclr) & ∼(TRACE | UCASEV_AL | EXPORT))) {
    bi_errorf("only␣-t,␣-u␣and␣-x␣options␣may␣be␣used␣with␣-f");
    return 1;
  }
```

1254. Take care of mutual exclusion—flags in *fset* are cleared in *fclr* and vise versa. This property should be preserved.

```
⟨ Parse options to typeset &c. 1251 ⟩ +≡
  if (wp[builtin_opt.optind]) {
    if (fset & LCASEV) fset &= ∼UCASEV_AL;      /* LCASEV has priority over UCASEV_AL */
    if (fset & LJUST) fset &= ∼RJUST;      /* LJUST has priority over RJUST */
    if ((fset & (ZEROFIL | LJUST)) ≡ ZEROFIL) {      /* -Z implies -ZR */
      fset |= RJUST;
      fclr &= ∼RJUST;
    }
    if (fset & (LJUST | RJUST | ZEROFIL | UCASEV_AL | LCASEV | INTEGER | INT_U | INT_L))
        /* Setting these attributes clears the others, unless they are also set in this command */
      fclr |= ∼fset & (LJUST | RJUST | ZEROFIL | UCASEV_AL | LCASEV | INTEGER | INT_U | INT_L);
  }
```

1255. ⟨ Set variables & attributes and **return** 1255 ⟩ ≡

```
int i;
int rval ← 0;
struct tbl *f;
if (local ∧ ¬func) fset |= LOCAL;
for (i ← builtin_opt.optind; wp[i]; i++) {
  if (func) {
    f ← findfunc(wp[i], hash(wp[i]), (fset & UCASEV_AL) ? true : false);
    if (¬f) {      /* AT&T ksh does ++rval; bogus */
      rval ← 1;
      continue;
    }
    if (fset | fclr) {
      f→flag |= fset;
      f→flag &= ∼fclr;
    }
    else fptreef(shl_stdout, 0,
        f→flag & FKSH ? "function␣%s␣%T\n" : "%s()␣%T\n",
        wp[i], f→val.t);
  }
  else if (¬typeset(wp[i], fset, fclr, field, base)) {
    bi_errorf("%s:␣not␣identifier", wp[i]);
    return 1;
  }
}
return rval;
```

This code is used in section 1250.

1256. ⟨ List all user functions 1256 ⟩ ≡

```
for (l ← genv→loc; l; l ← l→next) {
  for (p ← ktsort(&l→funs); (vp ← *p++); ) {
    if (flag ∧ (vp→flag & flag) ≡ 0) continue;
    if (thing ≡ '-')
      fptreef(shl_stdout, 0,
          vp→flag & FKSH ? "function␣%s␣%T\n" : "%s()␣%T\n",
          vp→name, vp→val.t);
    else shprintf("%s\n", vp→name);
  }
}
```

This code is used in section 1250.

1257. ⟨ List all variables 1257 ⟩ ≡

```
for (l ← genv→loc; l; l ← l→next) {
  for (p ← ktsort(&l→vars); (vp ← *p++); ) {
    struct tbl *tvp;
    int any_set ← 0;
    for (tvp ← vp; tvp; tvp ← tvp→u.array)      /* See if the variable (arrays: any element) is set */
      if (tvp→flag & ISSET) {
        any_set ← 1;
        break;
      }
    ⟨ List a single variable or array 1258 ⟩
  }
}
```

This code is used in section 1250.

1258. Check attributes. All array elements (should) have the same attributes so checking the first is sufficient. Report an unset variable only if the user has explicitly given it some attribute (like export), otherwise after "`echo $FOO`" we would report `FOO`.

⟨ List a single variable or array 1258 ⟩ ≡

```
if (¬any_set ∧ ¬(vp→flag & USERATTRIB)) continue;
if (flag ∧ (vp→flag & flag) ≡ 0) continue;
for ( ; vp; vp ← vp→u.array) {
  if ((vp→flag & ARRAY) ∧ any_set ∧ ¬(vp→flag & ISSET))      /* List the first element if none is set */
    continue;      /* ... otherwise ignore any other unset elements in an array */
  if (thing ≡ 0 ∧ flag ≡ 0) {      /* no arguments */
    ⟨ Print a variable's flags 1259 ⟩
    break;
  }
  else {
    if (pflag) shprintf("%s␣", (flag & EXPORT) ? "export" : "readonly");
    if ((vp→flag & ARRAY) ∧ any_set) shprintf("%s[%d]", vp→name, vp→index);
    else shprintf("%s", vp→name);
    if (thing ≡ '-' ∧ (vp→flag & ISSET)) {
      char *s ← str_val(vp);
      shprintf("=");
      if ((vp→flag & (INTEGER | LJUST | RJUST)) ≡ INTEGER) shprintf("%s", s);
      else print_value_quoted(s);      /* AT&T ksh can't have justified integers. */
    }
    shprintf("\n");
  }
  if (¬any_set) break;      /* ignore the rest of an un-set array */
}
```

This code is used in section 1257.

1259. AT&T ksh prints things like “export”, “integer”, “leftadj”, “zerofill”, etc., but POSIX says the output must be suitable for re-entry.

```
⟨ Print a variable's flags 1259 ⟩ ≡
  shprintf("typeset␣");
  if ((vp→flag & INTEGER)) shprintf("-i␣");
  if ((vp→flag & EXPORT)) shprintf("-x␣");
  if ((vp→flag & RDONLY)) shprintf("-r␣");
  if ((vp→flag & TRACE)) shprintf("-t␣");
  if ((vp→flag & LJUST)) shprintf("-L%d␣", vp→u2.field);
  if ((vp→flag & RJUST)) shprintf("-R%d␣", vp→u2.field);
  if ((vp→flag & ZEROFIL)) shprintf("-Z␣");
  if ((vp→flag & LCASEV)) shprintf("-l␣");
  if ((vp→flag & UCASEV_AL)) shprintf("-u␣");
  if ((vp→flag & INT_U)) shprintf("-U␣");
  shprintf("%s\n", vp→name); if (vp→flag & ARRAY)
```

This code is used in section 1258.

1260. "alias ...". Also implements "hash".

```
⟨ KSH commands 36 ⟩ +≡
  int c_alias(char **wp)
  {
    struct table *t ← &aliases;
    int rv ← 0, rflag ← 0, tflag, Uflag ← 0, pflag ← 0, prefix ← 0;
    int xflag ← 0;
    int optc;
    builtin_opt.flags |= GF_PLUSOPT;
    while ((optc ← ksh_getopt(wp, &builtin_opt, "dprtUx")) ≠ −1) {
      prefix ← builtin_opt.info & GI_PLUS ? '+' : '-';
      switch (optc) {
      case 'd': t ← &homedirs; break;
      case 'p': pflag ← 1; break;
      case 'r': rflag ← 1; break;
      case 't': t ← &taliases; break;
      case 'U': Uflag ← 1; break;      /* tracked alias kludge—no path search just make an entry */
      case 'x': xflag ← EXPORT; break;
      case '?': return 1;
      }
    }
    wp += builtin_opt.optind;
    if (¬(builtin_opt.info & GI_MINUSMINUS) ∧ *wp ∧ (wp[0][0] ≡ '-' ∨ wp[0][0] ≡ '+') ∧ wp[0][1] ≡ '\0') {
      prefix ← wp[0][0];
      wp++;
    }
    tflag ← t ≡ &taliases;
    if (rflag) {⟨ Reset all tracked aliases and return 1261 ⟩}
    if (*wp ≡ Λ) {⟨ List all aliases 1262 ⟩}
    for ( ; *wp ≠ Λ; wp++) {
      char *alias ← *wp;
      char *val ← strchr(alias, '=');
      char *newval;
      struct tbl *ap;
      int h;
      if (val) alias ← str_nsave(alias, val++ − alias, ATEMP);
      h ← hash(alias);
      if (val ≡ Λ ∧ ¬tflag ∧ ¬xflag) {⟨ List a named alias and continue 1263 ⟩}
      ap ← ktenter(t, alias, h);
      ap¬type ← tflag ? CTALIAS : CALIAS;
      if ((val ∧ ¬tflag) ∨ (¬val ∧ tflag ∧ ¬Uflag)) {⟨ Set an alias' value and/or flags 1264 ⟩}
      ap¬flag |= DEFINED;
      if (prefix ≡ '+') ap¬flag &= ∼xflag;
      else ap¬flag |= xflag;
      if (val) afree(alias, ATEMP);
    }
    return rv;
  }
```

1261. "`hash -r`" means reset all of the tracked aliases.

```
⟨ Reset all tracked aliases and return 1261 ⟩ ≡
  static const char *const args[] ← {"unalias","-ta",Λ};
  if (¬tflag ∨ *wp) {
    shprintf("alias:␣-r␣flag␣can␣only␣be␣used␣with␣-t""␣and␣without␣arguments\n");
    return 1;
  }
  ksh_getopt_reset(&builtin_opt,GF_ERROR);
  return c_unalias((char **) args);
```

This code is used in section 1260.

1262.

```
⟨ List all aliases 1262 ⟩ ≡
  struct tbl *ap, **p;
  for (p ← ktsort(t); (ap ← *p++) ≠ Λ; )
    if ((ap→flag & (ISSET | xflag)) ≡ (ISSET | xflag)) {
      if (pflag) shf_puts("alias␣",shl_stdout);
      shf_puts(ap→name,shl_stdout);
      if (prefix ≠ '+') {
        shf_putc('=',shl_stdout);
        print_value_quoted(ap→val.s);
      }
      shprintf("\n");
    }
```

This code is used in section 1260.

1263.

```
⟨ List a named alias and continue 1263 ⟩ ≡
  ap ← ktsearch(t,alias,h);
  if (ap ≠ Λ ∧ (ap→flag & ISSET)) {
    if (pflag) shf_puts("alias␣",shl_stdout);
    shf_puts(ap→name,shl_stdout);
    if (prefix ≠ '+') {
      shf_putc('=',shl_stdout);
      print_value_quoted(ap→val.s);
    }
    shprintf("\n");
  }
  else {
    shprintf("%s␣alias␣not␣found\n",alias);
    rv ← 1;
  }
  continue;
```

This code is used in section 1260.

1264. ⟨ Set an alias' value and/or flags 1264 ⟩ ≡

```
if (ap→flag & ALLOC) {
  ap→flag &= ∼(ALLOC | ISSET);
  afree(ap→val.s, APERM);
}
newval ← tflag ? search(alias, search_path, X_OK, Λ)      /* ignore values for -t (AT&T ksh does this) */
    : val;
if (newval) {
  ap→val.s ← str_save(newval, APERM);
  ap→flag |= ALLOC | ISSET;
}
else ap→flag &= ∼ISSET;
```

This code is used in section 1260.

1265. "`unalias [-adt] [name ...]`".

⟨ KSH commands 36 ⟩ +≡

```
int c_unalias(char **wp)
{
  struct table *t ← &aliases;
  struct tbl *ap;
  int rv ← 0, all ← 0;
  int optc;
  while ((optc ← ksh_getopt(wp, &builtin_opt, "adt")) ≠ −1)
    switch (optc) {
    case 'a': all ← 1; break;
    case 'd': t ← &homedirs; break;
    case 't': t ← &taliases; break;
    case '?': return 1;
    }
  wp += builtin_opt.optind;
  for ( ; *wp ≠ Λ; wp++) {⟨ Find and remove a named alias 1266 ⟩}
  if (all) {
    struct tstate ts;
    for (ktwalk(&ts, t); (ap ← ktnext(&ts)); ) {⟨ Remove an alias 1267 ⟩}
  }
  return rv;
}
```

1266. ⟨ Find and remove a named alias 1266 ⟩ ≡

```
ap ← ktsearch(t, *wp, hash(*wp));
if (ap ≡ Λ) {
  rv ← 1;      /* POSIX */
  continue;
}
if (ap→flag & ALLOC) {
  ap→flag &= ∼(ALLOC | ISSET);
  afree(ap→val.s, APERM);
}
ap→flag &= ∼(DEFINED | ISSET | EXPORT);
```

This code is used in section 1265.

1267. ⟨ Remove an alias 1267 ⟩ ≡

```
if (ap→flag & ALLOC) {
  ap→flag &= ∼(ALLOC | ISSET);
  afree(ap→val.s, APERM);
}
ap→flag &= ∼(DEFINED | ISSET | EXPORT);
```

This code is used in section 1265.

1268. "`let` *expression ...*".

```
⟨ KSH commands 36 ⟩ +≡
int c_let(char **wp)
{
  int rv ← 1;
  int64_t val;
  if (wp[1] ≡ Λ) bi_errorf("no␣arguments");      /* AT&T ksh does this */
  else
    for (wp++; *wp; wp++)
      if (¬evaluate(*wp, &val, KSH_RETURN_ERROR, true)) {
        rv ← 2;      /* distinguish error from zero result */
        break;
      }
      else rv ← val ≡ 0;
  return rv;
}
```

1269. "`jobs [-lnp]` [*job ...*]".

```
⟨ KSH commands 36 ⟩ +≡
int c_jobs(char **wp)
{
  int optc;
  int flag ← 0;
  int nflag ← 0;
  int rv ← 0;
  while ((optc ← ksh_getopt(wp, &builtin_opt, "lpnz")) ≠ −1)
    switch (optc) {
    case 'l': flag ← 1; break;
    case 'p': flag ← 2; break;
    case 'n': nflag ← 1; break;
    case 'z': nflag ← −1; break;      /* debugging: print zombies */
    case '?': return 1;
    }
  wp += builtin_opt.optind;
  if (¬*wp) {
    if (j_jobs(Λ, flag, nflag)) rv ← 1;
  }
  else {
    for ( ; *wp; wp++)
      if (j_jobs(*wp, flag, nflag)) rv ← 1;
  }
  return rv;
}
```

1270. "bg|fg [*job* ...]". POSIX says fg shall return 0 unless an error occurs. AT&T ksh returns the exit value of the job.

```
⟨ KSH commands 36 ⟩ +≡
  int c_fgbg(char **wp)
  {
    int bg ← strcmp(*wp, "bg") ≡ 0;
    int rv ← 0;
    if (¬Flag(FMONITOR)) {
      bi_errorf("job␣control␣not␣enabled");
      return 1;
    }
    if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
    wp += builtin_opt.optind;
    if (*wp)
      for ( ; *wp; wp++) rv ← j_resume(*wp, bg);
    else rv ← j_resume("%%", bg);
    return (bg ∨ Flag(FPOSIX)) ? 0 : rv;
  }
```

1271. "kill -l [*status* ...]" & "kill [-s *name*|-*NAME*|-*signal*] {*job*|*process*|*group*} ...".

```
⟨ KSH commands 36 ⟩ +≡
  int c_kill(char **wp)
  {
    Trap *t ← Λ;
    char *p;
    int lflag ← 0;
    int i, n, rv, sig;
    if ((p ← wp[1]) ∧ *p ≡ '-' ∧
        (digit(p[1]) ∨ isupper((unsigned char) p[1]))) {⟨ Parse brief options to kill 1272 ⟩}
    else {⟨ Parse full options to kill 1273 ⟩}
    if ((lflag ∧ t) ∨ (¬wp[i] ∧ ¬lflag)) {
      shf_fprintf(shl_out,
          "usage:␣kill␣[-s␣signame␣|␣-signum␣|␣-signame]␣{␣job␣|␣pid␣|␣pgrp␣}␣...\n"
          "␣␣␣␣␣␣␣␣kill␣-l␣[exit_status␣...]\n");
      bi_errorf(Λ);
      return 1;
    }
    if (lflag) {⟨ List signals and return 1274 ⟩}
    rv ← 0;
    sig ← t ? t→signal : SIGTERM;
    for ( ; (p ← wp[i]); i++) {⟨ Signal a process 1278 ⟩}
    return rv;
  }
```

1272. Assume "old" style options if `-`⟨*digits*⟩ or `-`*UPPERCASE*.

```
⟨ Parse brief options to kill 1272 ⟩ ≡
  if (¬(t ← gettrap(p + 1, true))) {
    bi_errorf("bad␣signal␣‘%s’", p + 1);
    return 1;
  }
  i ← (wp[2] ∧ strcmp(wp[2], "--") ≡ 0) ? 3 : 2;
```

This code is used in section 1271.

1273.

```
⟨ Parse full options to kill 1273 ⟩ ≡
  int optc;
  while ((optc ← ksh_getopt(wp, &builtin_opt, "ls:")) ≠ −1)
    switch (optc) {
    case 'l': lflag ← 1; break;
    case 's':
      if (¬(t ← gettrap(builtin_opt.optarg, true))) {
        bi_errorf("bad␣signal␣‘%s’", builtin_opt.optarg);
        return 1;
      }
      break;
    case '?': return 1;
    }
  i ← builtin_opt.optind;
```

This code is used in section 1271.

1274.

```
⟨ List signals and return 1274 ⟩ ≡
  if (wp[i]) {
    for ( ; wp[i]; i++) {
      if (¬bi_getn(wp[i], &n)) return 1;
      if (n > 128 ∧ n < 128 + NSIG) n −= 128;
      if (n > 0 ∧ n < NSIG ∧ sigtraps[n].name) shprintf("%s\n", sigtraps[n].name);
      else shprintf("%d\n", n);
    }
  }
  else if (Flag(FPOSIX)) {
    p ← null;
    for (i ← 1; i < NSIG; i++, p ← "␣")
      if (sigtraps[i].name) shprintf("%s%s", p, sigtraps[i].name);
    shprintf("\n");
  }
  else {⟨ List all possible signals 1275 ⟩}
  return 0;
```

This code is used in section 1271.

1275. ⟨ List all possible signals 1275 ⟩ ≡

```
int mess_width ← 0, w;
struct kill_info ki ← {.num_width ← 1, .name_width ← 0, };
for (i ← NSIG; i ≥ 10; i /= 10) ki.num_width++;
for (i ← 0; i < NSIG; i++) {
  w ← sigtraps[i].name ? (int) strlen(sigtraps[i].name) : ki.num_width;
  if (w > ki.name_width) ki.name_width ← w;
  w ← strlen(sigtraps[i].mess);
  if (w > mess_width) mess_width ← w;
}
print_columns(shl_stdout, NSIG − 1, kill_fmt_entry, (void *) &ki,
    ki.num_width + ki.name_width + mess_width + 3, 1);
```

This code is used in section 1274.

1276. ⟨ Type definitions 17 ⟩ +≡

```
struct kill_info {
  int num_width;
  int name_width;
};
```

1277. ⟨ KSH commands 36 ⟩ +≡

```
static char *kill_fmt_entry(void *arg, int i, char *buf, int buflen)
{
  struct kill_info *ki ← (struct kill_info *) arg;
  i++;
  if (sigtraps[i].name) shf_snprintf(buf, buflen, "%*d␣%*s␣%s",
      ki→num_width, i,
      ki→name_width, sigtraps[i].name,
      sigtraps[i].mess);
  else shf_snprintf(buf, buflen, "%*d␣%*d␣%s",
      ki→num_width, i,
      ki→name_width, sigtraps[i].signal,
      sigtraps[i].mess);
  return buf;
}
```

1278. Use *killpg* if the process ID is less than -1 since *kill*(−1) does special things that *killpg* doesn't for this special "process group" (see kill(2)).

```
⟨ Signal a process 1278 ⟩ ≡
  if (*p ≡ '%') {
    if (j_kill(p, sig)) rv ← 1;
  }
  else if (¬getn(p, &n)) {
    bi_errorf("%s: arguments must be jobs or process IDs", p);
    rv ← 1;
  }
  else {
    if ((n < −1 ? killpg(−n, sig) : kill(n, sig)) ≡ −1) {
      bi_errorf("%s: %s", p, strerror(errno));
      rv ← 1;
    }
  }
```

This code is used in section 1271.

1279. "`bind [-l]`" or "`bind [-m]` [*string*=`{`*substitute*|*editing-command*`}`]". For emacs mode only.

```
⟨ KSH commands 36 ⟩ +≡
#ifdef EMACS
  int c_bind(char **wp)
  {
    int optc, rv ← 0, macro ← 0, list ← 0;
    char *cp;
    while ((optc ← ksh_getopt(wp, &builtin_opt, "lm")) ≠ −1)
      switch (optc) {
      case 'l': list ← 1; break;
      case 'm': macro ← 1; break;
      case '?': return 1;
      }
    wp += builtin_opt.optind;
    if (*wp ≡ Λ) rv ← x_bind(Λ, Λ, 0, list);      /* list all bindings */
    for ( ; *wp ≠ Λ; wp++) {
      cp ← strchr(*wp, '=');
      if (cp ≠ Λ) *cp++ ← '\0';
      if (x_bind(*wp, cp, macro, 0)) rv ← 1;
    }
    return rv;
  }
#endif      /* EMACS */
```

1280. "`getopts` *option-string name* [*arg ...*]".

```
⟨ KSH commands 36 ⟩ +≡
  int c_getopts(char **wp)
  {
    int argc;
    const char *options;
    const char *var;
    int optc;
    int ret;
    char buf[3];
    struct tbl *vq, *voptarg;
    ⟨ Parse the arguments to getopts 1281 ⟩
    ⟨ Abort getopts if the state is invalid 1282 ⟩
    user_opt.optarg ← Λ;
    optc ← ksh_getopt(wp, &user_opt, options);
    if (optc ≥ 0 ∧ optc ≠ '?' ∧ (user_opt.info & GI_PLUS)) {
      buf[0] ← '+';
      buf[1] ← optc;
      buf[2] ← '\0';
    }
    else {      /* POSIX says "?" marks the end of options, AT&T ksh uses null. We go with POSIX. */
      buf[0] ← optc < 0 ? '?' : optc;
      buf[1] ← '\0';
    }
    if (optc ≠ '?') {      /* AT&T ksh does not change $OPTIND if it was an unknown option. Scripts
          counting on this are prone to break... (ie. don't count on this staying). */
      user_opt.uoptind ← user_opt.optind;
    }
    voptarg ← global("OPTARG");
    voptarg¬flag &= ∼RDONLY;      /* AT&T ksh clears read-only and integer */
    if (voptarg¬flag & INTEGER)      /* Paranoia: ensure no bizarre results. */
      typeset("OPTARG", 0, INTEGER, 0, 0);
    if (user_opt.optarg ≡ Λ) unset(voptarg, 0);
    else setstr(voptarg, user_opt.optarg, KSH_RETURN_ERROR);      /* can't fail (¬read-only/integer) */
    ret ← 0;
    vq ← global(var);
    if (¬setstr(vq, buf, KSH_RETURN_ERROR)) ret ← 1;      /* Error message already printed */
    if (Flag(FEXPORT)) typeset(var, EXPORT, 0, 0, 0);
    return optc < 0 ? 1 : ret;
  }
```

1281. ⟨ Parse the arguments to `getopts` 1281 ⟩ ≡

```
if (ksh_getopt(wp, &builtin_opt, null) ≡ '?') return 1;
wp += builtin_opt.optind;
options ← *wp++;
if (¬options) {
  bi_errorf("missing␣options␣argument");
  return 1;
}
var ← *wp++;
if (¬var) {
  bi_errorf("missing␣name␣argument");
  return 1;
}
if (¬*var ∨ *skip_varname(var, true)) {
  bi_errorf("%s:␣is␣not␣an␣identifier", var);
  return 1;
}
if (genv→loc→next ≡ Λ) {
  internal_warningf("%s:␣no␣argv", __func__);
  return 1;
}
if (*wp ≡ Λ) wp ← genv→loc→next→argv;      /* Parse arguments from the environment */
else *--wp ← genv→loc→next→argv[0];      /* Mock $0 to parse arguments to getopts */
```

This code is used in section 1280.

1282. If the arguments have changed then using *wp* could be a memory access violation.

⟨ Abort `getopts` if the state is invalid 1282 ⟩ ≡

```
for (argc ← 0; wp[argc]; argc++) ;
if (user_opt.optind > argc ∨ (user_opt.p ≠ 0 ∧ user_opt.p > strlen(wp[user_opt.optind − 1]))) {
  bi_errorf("arguments␣changed␣since␣last␣call");
  return 1;
}
```

This code is used in section 1280.

1283. The `ulimit` built-in is complex enough to command its own implementation file.

This started out as the BRL UNIX System V system call emulation for 4.*n*BSD, and was later extended by Doug Kingston to handle the extended 4.*n*BSD resource limits. It now includes the code that was originally under case `SYSULIMIT` in source file "`xec.c`".

The mysts of time were ended (the "last edit") on 1987-06-06 by D A Gwyn. Eric Gisin then adapted this to public domain Korn Shell by removing AT&T code in September 1988.

It was then reworked in May 1994 by Michael Rendell to use *getrtusage* and *ulimit* at once (as needed on some schizophrenic systems, eg. HP-UX 9.01), make argument parsing conform to AT&T and add autoconf support.

```
⟨ c_ulimit.c 1283 ⟩ ≡
#include <sys/resource.h>
#include <ctype.h>
#include <errno.h>
#include <inttypes.h>
#include <string.h>
#include "sh.h"
  static void print_ulimit(const struct Limits *, int);
  static int set_ulimit(const struct Limits *, const char *, int);
```

See also sections 1285, 1290, and 1291.

1284.
```
⟨ Type definitions 17 ⟩ +≡
  struct Limits {
    const char *name;
    int resource;      /* resource to get/set */
    int factor;      /* multiply by to get rlim_{cur,max} values */
    char option;      /* option character (-d, -f, ...) */
  };
```

1285. "ulimit [-acdfHlmnpSst [*value*]] ...".

```
#define SOFT 0x1
#define HARD 0x2
⟨ c_ulimit.c 1283 ⟩ +≡
  int c_ulimit(char **wp)
  {
    static const struct Limits limits[] ← {⟨ ulimit limits 1286 ⟩};
    const char *options ← "HSat#f#c#d#s#l#m#n#p#";
    int how ← SOFT | HARD;
    const struct Limits *l;
    int optc, all ← 0;      /* First check for -a, -S and -H. */
    while ((optc ← ksh_getopt(wp, &builtin_opt, options)) ≠ −1)
      switch (optc) {
      case 'H': how ← HARD; break;
      case 'S': how ← SOFT; break;
      case 'a': all ← 1; break;
      case '?': return 1;
      default: break;
      }
    if (wp[builtin_opt.optind] ≠ Λ) {
      bi_errorf("usage: ulimit [-acdfHlmnpSst] [value]");
      return 1;
    }
    ksh_getopt_reset(&builtin_opt, GF_ERROR);
    while ((optc ← ksh_getopt(wp, &builtin_opt, options)) ≠ −1) ⟨ Parse and act on a limit 1288 ⟩
    wp += builtin_opt.optind;
    if (all) {⟨ Print all ulimit limits 1287 ⟩}
    else if (builtin_opt.optind ≡ 1) {⟨ Set a single un-named ulimit limit 1289 ⟩}
    return 0;
  }
```

1286. Do not use options -H, -S or -a or change the order.

```
⟨ ulimit limits 1286 ⟩ ≡
  {"time(cpu-seconds)", RLIMIT_CPU, 1, 't'},
  {"file(blocks)", RLIMIT_FSIZE, 512, 'f'},
  {"coredump(blocks)", RLIMIT_CORE, 512, 'c'},
  {"data(kbytes)", RLIMIT_DATA, 1024, 'd'},
  {"stack(kbytes)", RLIMIT_STACK, 1024, 's'},
  {"lockedmem(kbytes)", RLIMIT_MEMLOCK, 1024, 'l'},
  {"memory(kbytes)", RLIMIT_RSS, 1024, 'm'},
  {"nofiles(descriptors)", RLIMIT_NOFILE, 1, 'n'},
  {"processes", RLIMIT_NPROC, 1, 'p'},
  {Λ}
```

This code is used in section 1285.

1287. ⟨ Print all ulimit limits 1287 ⟩ ≡

```
  for (l ← limits; l→name; l++) {
    shprintf("%-20s ", l→name);
    print_ulimit(l, how);
  }
```

This code is used in section 1285.

1288. ⟨ Parse and act on a limit 1288 ⟩ ≡

```
switch (optc) {
case 'a': case 'H': case 'S': break;
case '?': return 1;
default:
  for (l ← limits; l→name ∧ l→option ≠ optc; l++) ;
  if (¬l→name) {
    internal_warningf("%s: %c", __func__, optc);
    return 1;
  }
  if (builtin_opt.optarg) {
    if (set_ulimit(l, builtin_opt.optarg, how)) return 1;
  }
  else print_ulimit(l, how);
  break;
}
```

This code is used in section 1285.

1289. If no limit is specified use file size (*limits*[1]).

⟨ Set a single un-named `ulimit` limit 1289 ⟩ ≡

```
l ← &limits[1];
if (wp[0] ≠ Λ) {
  if (set_ulimit(l, wp[0], how)) return 1;
  wp++;
}
else {
  print_ulimit(l, how);
}
```

This code is used in section 1285.

1290. To avoid problems caused by typos that evaluate unset parameters to zero the source string is checked to ensure the value is numeric. Alternatively a parameter could be added to *evaluate* to control if unset variables are zero or an error.

```
⟨ c_ulimit.c 1283 ⟩ +≡
  static int set_ulimit(const struct Limits *l, const char *v, int how)
  {
    rlim_t val ← 0;
    struct rlimit limit;
    if (strcmp(v, "unlimited") ≡ 0) val ← RLIM_INFINITY;
    else {
      int64_t rval;
      if (¬evaluate(v, &rval, KSH_RETURN_ERROR, false)) return 1;
      if (¬rval ∧ ¬digit(v[0])) {
        bi_errorf("invalid limit: %s", v);
        return 1;
      }
      val ← (rlim_t)rval * l→factor;
    }
    getrlimit(l→resource, &limit);
    if (how & SOFT) limit.rlim_cur ← val;
    if (how & HARD) limit.rlim_max ← val;
    if (setrlimit(l→resource, &limit) ≡ −1) {
      if (errno ≡ EPERM) bi_errorf("-%c exceeds allowable limit", l→option);
      else bi_errorf("bad -%c limit: %s", l→option, strerror(errno));
      return 1;
    }
    return 0;
  }
```

1291. ⟨ c_ulimit.c 1283 ⟩ +≡

```
  static void print_ulimit(const struct Limits *l, int how)
  {
    rlim_t val ← 0;
    struct rlimit limit;
    getrlimit(l→resource, &limit);
    if (how & SOFT) val ← limit.rlim_cur;
    else if (how & HARD) val ← limit.rlim_max;
    if (val ≡ RLIM_INFINITY) shprintf("unlimited\n");
    else {
      val /= l→factor;
      shprintf("%" PRIi64 "\n", (int64_t) val);
    }
  }
```

1292. The `test`/`[`/`[[` command needs its own file and a header. Originally version 7-like by Erik Baalbergen, modified by:

* Eric Gisin to be used as a built-in.
* Arnold Robbins to add SVR3 compatility (`-xcbpugk` & Korn's `-L`, `-nt`, `-ot`, `-ef` & `-S` *socket*.
* Michael Rendell to add Korn's "`[[ ... ]]`" expressions.
* J.T. Conklin to add POSIX compatibility.

⟨ `c_test.c` 1292 ⟩ ≡

```
#include <sys/stat.h>
#include <string.h>
#include <unistd.h>
#include "sh.h"
#include "c_test.h"
  static int test_eaccess(const char *, int);
  static int test_oexpr(Test_env *, int);
  static int test_aexpr(Test_env *, int);
  static int test_nexpr(Test_env *, int);
  static int test_primary(Test_env *, int);
  static int ptest_isa(Test_env *, Test_meta);
  static const char *ptest_getopnd(Test_env *, Test_op, int);
  static int ptest_eval(Test_env *, Test_op, const char *, const char *, int);
  static void ptest_error(Test_env *, int, const char *);
```

See also sections 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1310, 1313, 1314, 1318, 1324, and 1325.

1293. ⟨ `c_test.h` 1293 ⟩ ≡

```
  Test_op test_isop(Test_env *, Test_meta, const char *);
  int test_eval(Test_env *, Test_op, const char *, const char *, int);
  int test_parse(Test_env *);
```

1294. ⟨ Shared function declarations 4 ⟩ +≡

```
  int c_test(char **);
```

1295. `test` accepts the following grammar:

```
or-expression ::= and-expression | and-expression "-o" or-expression ;
and-expression ::= not-expression | not-expression "-a" and-expression ;
not-expression ::= primary | "!" not-expression ;
primary ::= unary-operator operand
        | operand binary-operator operand
        | operand
        | "(" or-expression ")"
        ;
unary-operator ::= "-a"|"-r"|"-w"|"-x"|"-e"|"-f"|"-d"|"-c"|"-b"|"-p"|
                   "-u"|"-g"|"-k"|"-s"|"-t"|"-z"|"-n"|"-o"|"-O"|"-G"|
                   "-L"|"-h"|"-S"|"-H";
binary-operator ::= "="|"=="|"!="|"-eq"|"-ne"|"-ge"|"-gt"|"-le"|"-lt"|
                    "-nt"|"-ot"|"-ef"|"<"|">"
operand ::= ⟨anything⟩
```

1296. These are recognised via two tables of **t_op** objects.

```
⟨ Type definitions 17 ⟩ +≡
  ⟨ List of all test operators 1298 ⟩
  struct t_op {
    char op_text[4];
    Test_op op_num;
  };
```

1297. To scan for an operator (a meta-operator really) they are grouped into categories.

```
⟨ Type definitions 17 ⟩ +≡
  typedef enum Test_meta Test_meta;
  enum Test_meta {
    TM_OR,      /* -o or || */
    TM_AND,     /* -a or && */
    TM_NOT,     /* ! */
    TM_OPAREN,     /* ( */
    TM_CPAREN,     /* ) */
    TM_UNOP,     /* unary operator */
    TM_BINOP,     /* binary operator */
    TM_END     /* end of input */
  };
```

1298. The operators themselves are listed in full in **Test_op**, with unary and binary operators each collected into a group.

```
⟨ List of all test operators 1298 ⟩ ≡
  typedef enum Test_op Test_op;
  enum Test_op {
        TO_NONOP ← 0,      /* non-operator */
        TO_STNZE, TO_STZER, TO_OPTION,
        TO_FILAXST, TO_FILEXST,
        TO_FILREG, TO_FILBDEV, TO_FILCDEV, TO_FILSYM, TO_FILFIFO, TO_FILSOCK,
        TO_FILCDF, TO_FILID, TO_FILGID, TO_FILSETG, TO_FILSTCK, TO_FILUID,
        TO_FILRD, TO_FILGZ, TO_FILTT, TO_FILSETU, TO_FILWR, TO_FILEX,
        TO_STEQL, TO_STNEQ, TO_STLT, TO_STGT, TO_INTEQ, TO_INTNE, TO_INTGT,
        TO_INTGE, TO_INTLT, TO_INTLE, TO_FILEQ, TO_FILNT, TO_FILOT
  };
```

This code is used in section 1296.

1299. Evaluating a test happens in a **Test_env** object, which has nothing to do with the *shell* environment.

```
⟨ Type definitions 17 ⟩ +≡
  typedef struct test_env Test_env;
  struct test_env {
    int flags;      /* TEF_* */
    union {
      char **wp;      /* used by ptest_* */
      XPtrV *av;      /* used by dbtestp_* */
    } pos;
    char **wp_end;      /* used by ptest_* */
    int(*isa)(Test_env *, Test_meta);
    const char *(*getopnd)(Test_env *, Test_op, int);
    int(*eval)(Test_env *, Test_op, const char *, const char *, int);
    void(*error)(Test_env *, int, const char *);
  };
```

1300. One table of **t_op** objects for unary operators.

```
⟨ c_test.c 1292 ⟩ +≡
  static const struct t_op u_ops[] ← {
      {"-a",TO_FILAXST},{"-b",TO_FILBDEV},{"-c",TO_FILCDEV},{"-d",TO_FILID},
      {"-e",TO_FILEXST},{"-f",TO_FILREG},{"-G",TO_FILGID},{"-g",TO_FILSETG},
      {"-h",TO_FILSYM},{"-H",TO_FILCDF},{"-k",TO_FILSTCK},{"-L",TO_FILSYM},
      {"-n",TO_STNZE},{"-O",TO_FILUID},{"-o",TO_OPTION},{"-p",TO_FILFIFO},
      {"-r",TO_FILRD},{"-s",TO_FILGZ},{"-S",TO_FILSOCK},{"-t",TO_FILTT},
      {"-u",TO_FILSETU},{"-w",TO_FILWR},{"-x",TO_FILEX},{"-z",TO_STZER},
      {"",TO_NONOP}
  };
```

1301. Another table for binary operators.

```
⟨ c_test.c 1292 ⟩ +≡
  static const struct t_op b_ops[] ← {
      {"=",TO_STEQL},{"==",TO_STEQL},{"!=",TO_STNEQ},{"<",TO_STLT},
      {">",TO_STGT},{"-eq",TO_INTEQ},{"-ne",TO_INTNE},{"-gt",TO_INTGT},
      {"-ge",TO_INTGE},{"-lt",TO_INTLT},{"-le",TO_INTLE},{"-ef",TO_FILEQ},
      {"-nt",TO_FILNT},{"-ot",TO_FILOT},
      {"",TO_NONOP}
  };
```

1302. The common entry point to testing is *test_parse* which follows the grammar.

```
#define T_ERR_EXIT 2 /* POSIX says ¿ 1 for errors */
#define TEF_ERROR BIT(0) /* set if we've hit an error */
#define TEF_DBRACKET BIT(1) /* set if this is a [[ ... ]] test */
⟨ c_test.c 1292 ⟩ +≡
  int test_parse(Test_env *te)
  {
    int res;
    res ← test_oexpr(te, 1);
    if (¬(te→flags & TEF_ERROR) ∧ ¬(*te→isa)(te, TM_END))
      (*te→error)(te, 0, "unexpected␣operator/operand");
    return (te→flags & TEF_ERROR) ? T_ERR_EXIT : ¬res;
  }
```

1303. *ptest_isa* determines if we have the type of token we want. If so then the *wp* pointer is incremented to the next token, "accepting" this one.

```
⟨ c_test.c 1292 ⟩ +≡
  static int ptest_isa(Test_env *te, Test_meta meta)
  {   /* Order is important—indexed by Test_meta values */
    static const char *const tokens[] ← {"-o", "-a", "!", "(", ")"};
    int ret;
    if (te→pos.wp ≥ te→wp_end) return meta ≡ TM_END;
    if (meta ≡ TM_UNOP ∨ meta ≡ TM_BINOP) ret ← (int) test_isop(te, meta, *te→pos.wp);
    else if (meta ≡ TM_END) ret ← 0;
    else ret ← strcmp(*te→pos.wp, tokens[(int) meta]) ≡ 0;
    if (ret) te→pos.wp++; /* Accept the token. */
    return ret;
  }
```

1304. To see if a string is an operator the appropriate table is scanned.

```
⟨ c_test.c 1292 ⟩ +≡
  Test_op test_isop(Test_env *te, Test_meta meta, const char *s)
  {
    char sc1;
    const struct t_op *otab;
    otab ← meta ≡ TM_UNOP ? u_ops : b_ops;
    if (*s) {
      sc1 ← s[1];
      for ( ; otab→op_text[0]; otab++)
        if (sc1 ≡ otab→op_text[1] ∧ strcmp(s, otab→op_text) ≡ 0) return otab→op_num;
    }
    return TO_NONOP;
  }
```

1305. An or-expression (oexpr) does not evaluate its second argument if the first is a success.

```
⟨ c_test.c 1292 ⟩ +≡
  static int test_oexpr(Test_env *te, int do_eval)
  {
    int res;
    res ← test_aexpr(te, do_eval);
    if (res) do_eval ← 0;
    if (¬(te→flags & TEF_ERROR) ∧ (*te→isa)(te, TM_OR)) return test_oexpr(te, do_eval) ∨ res;
    return res;
  }
```

1306. Conversely an and-expression (aexpr) *only* evaluates its second argument if the first is a success.

```
⟨ c_test.c 1292 ⟩ +≡
  static int test_aexpr(Test_env *te, int do_eval)
  {
    int res;
    res ← test_nexpr(te, do_eval);
    if (¬res) do_eval ← 0;
    if (¬(te→flags & TEF_ERROR) ∧ (*te→isa)(te, TM_AND)) return test_aexpr(te, do_eval) ∧ res;
    return res;
  }
```

1307. A not-expression (nexpr) returns the inverse of the evaluation of its argument expression.

```
⟨ c_test.c 1292 ⟩ +≡
  static int test_nexpr(Test_env *te, int do_eval)
  {
    if (¬(te→flags & TEF_ERROR) ∧ (*te→isa)(te, TM_NOT)) return ¬test_nexpr(te, do_eval);
    return test_primary(te, do_eval);
  }
```

1308. A primary operator may finally call *te→eval*.

```
⟨ c_test.c 1292 ⟩ +≡
  static int test_primary(Test_env *te, int do_eval)
  {
    const char *opnd1, *opnd2;
    int res;
    Test_op op;
    if (te→flags & TEF_ERROR) return 0;
    if ((*te→isa)(te, TM_OPAREN)) {⟨ Test and return a parenthetical expression 1309 ⟩}
    if ((te→flags & TEF_DBRACKET) ∨
          (&te→pos.wp[1] < te→wp_end ∧ ¬test_isop(te, TM_BINOP, te→pos.wp[1]))) {
      if ((op ← (Test_op)(*te→isa)(te, TM_UNOP))) {⟨ Test and return a unary expression 1311 ⟩}
    }
    opnd1 ← (*te→getopnd)(te, TO_NONOP, do_eval);
    if (¬opnd1) {
      (*te→error)(te, 0, "expression␣expected");
      return 0;
    }
    if ((op ← (Test_op)(*te→isa)(te, TM_BINOP))) {⟨ Test and return a binary expression 1312 ⟩}
    if (te→flags & TEF_DBRACKET) {
      (*te→error)(te, −1, "missing␣expression␣operator");
      return 0;
    }
    return (*te→eval)(te, TO_STNZE, opnd1, Λ, do_eval);
  }
```

1309. ⟨ Test and **return** a parenthetical expression 1309 ⟩ ≡

```
  res ← test_oexpr(te, do_eval);
  if (te→flags & TEF_ERROR) return 0;
  if (¬(*te→isa)(te, TM_CPAREN)) {
    (*te→error)(te, 0, "missing␣closing␣paren");
    return 0;
  }
  return res;
```

This code is used in section 1308.

1310. Binary and unary operators have an operand.

```
⟨ c_test.c 1292 ⟩ +≡
  static const char *ptest_getopnd(Test_env *te, Test_op op, int do_eval)
  {
    if (te→pos.wp ≥ te→wp_end) return op ≡ TO_FILTT ? "1" : Λ;
    return *te→pos.wp++;
  }
```

1311. Binary has some precedence over unary in this case so that something like "`test \( -f = -f \)`" is accepted.

```
⟨ Test and return a unary expression 1311 ⟩ ≡
  opnd1 ← (∗te→getopnd)(te, op, do_eval);
  if (¬opnd1) {
    (∗te→error)(te, −1, "missing␣argument");
    return 0;
  }
  return (∗te→eval)(te, op, opnd1, Λ, do_eval);
```

This code is used in section 1308.

1312.

```
⟨ Test and return a binary expression 1312 ⟩ ≡
  opnd2 ← (∗te→getopnd)(te, op, do_eval);
  if (¬opnd2) {
    (∗te→error)(te, −1, "missing␣second␣argument");
    return 0;
  }
  return (∗te→eval)(te, op, opnd1, opnd2, do_eval);
```

This code is used in section 1308.

1313. Hinted at so far there are in fact various users of the test environment which are distinguished using function pointers. The `test` command's evaluator function is *ptest_eval*.

```
⟨ c_test.c 1292 ⟩ +≡
  static int ptest_eval(Test_env ∗te, Test_op op, const char ∗opnd1, const char ∗opnd2, int do_eval)
  {
    return test_eval(te, op, opnd1, opnd2, do_eval);
  }
```

1314. The evaluator selects between the different test operators.

```
⟨ c_test.c 1292 ⟩ +≡
  int test_eval(Test_env ∗te, Test_op op, const char ∗opnd1, const char ∗opnd2, int do_eval)
  {
    int res;
    int not;
    struct stat b1, b2;
    if (¬do_eval) return 0;
    switch ((int) op) {⟨ Evaluate and return a test operator 1315 ⟩}
    (∗te→error)(te, 0, "internal␣error:␣unknown␣op");
    return 1;
  }
```

1315. `-n`: The string operand is not empty; `-z`: the string operand *is* empty.

```
⟨ Evaluate and return a test operator 1315 ⟩ ≡
case TO_STNZE: return ∗opnd1 ≠ '\0';    /∗ -n ∗/
case TO_STZER: return ∗opnd1 ≡ '\0';    /∗ -z ∗/
```

See also sections 1316, 1317, 1319, 1320, 1321, 1322, and 1323.

This code is used in section 1314.

1316. `-o`: This is the unary operator to test if a shell operator is set not the 'or' operator; see *test_oexpr*.

```
⟨ Evaluate and return a test operator 1315 ⟩ +≡
case TO_OPTION:      /* -o */
  if ((not ← *opnd1 ≡ '!')) opnd1 ++;
  if ((res ← option(opnd1)) < 0) res ← 0;
  else {
    res ← Flag(res);
    if (not) res ← ¬res;
  }
  return res;
```

1317. File tests.

* `-a`: The file exists.
* `-b`: The file exists and is a block device.
* `-c`: The file exists and is a character device.
* `-d`: The file exists and is a directory.
* `-e`: The file exists. Apparently AT&T ksh does not appear to "do the /dev/fd/ thing for this" although the operating system itself may handle it.
* `-f`: The file exists and is a regular file.
* `-G`: The file exists and its group is the shell's effective group ID.
* `-g`: The file exists and its setgid bit is set.
* `-h` & `-L`: The file exists and is a symbolic link.
* `-H`: Unsupported; "HP context dependent files (directories)".
* `-k`: The file exists and its sticky bit is set.
* `-O`: The file exists and is owned by the shell's effective user ID.
* `-p`: The file exists and is a named pipe (FIFO).
* `-r`: The file exists and is readable.
* `-S`: The file exists and is a unix-domain socket.
* `-s`: The file exists and is not empty.
* `-t`: The (optional in POSIX mode) file descriptor is a tty.
* `-u`: The file exists and its setuid bit is set.
* `-w`: The file exists and is writable.
* `-x`: The file exists and is executable.

⟨ Evaluate and **return** a test operator 1315 ⟩ +≡

```
case TO_FILRD: return test_eaccess(opnd1,R_OK) ≡ 0;      /* -r */
case TO_FILWR: return test_eaccess(opnd1,W_OK) ≡ 0;      /* -w */
case TO_FILEX: return test_eaccess(opnd1,X_OK) ≡ 0;      /* -x */
case TO_FILAXST: return stat(opnd1,&b1) ≡ 0;      /* -a */
case TO_FILEXST: return stat(opnd1,&b1) ≡ 0;      /* -e */
case TO_FILREG: return stat(opnd1,&b1) ≡ 0 ∧ S_ISREG(b1.st_mode);      /* -f */
case TO_FILID: return stat(opnd1,&b1) ≡ 0 ∧ S_ISDIR(b1.st_mode);      /* -d */
case TO_FILCDEV: return stat(opnd1,&b1) ≡ 0 ∧ S_ISCHR(b1.st_mode);      /* -c */
case TO_FILBDEV: return stat(opnd1,&b1) ≡ 0 ∧ S_ISBLK(b1.st_mode);      /* -b */
case TO_FILFIFO: return stat(opnd1,&b1) ≡ 0 ∧ S_ISFIFO(b1.st_mode);      /* -p */
case TO_FILSYM: return lstat(opnd1,&b1) ≡ 0 ∧ S_ISLNK(b1.st_mode);      /* -h & -L */
case TO_FILSOCK: return stat(opnd1,&b1) ≡ 0 ∧ S_ISSOCK(b1.st_mode);      /* -S */
case TO_FILCDF: return 0;      /* -H */
case TO_FILSETU: return stat(opnd1,&b1) ≡ 0 ∧ (b1.st_mode & S_ISUID) ≡ S_ISUID;      /* -u */
case TO_FILSETG: return stat(opnd1,&b1) ≡ 0 ∧ (b1.st_mode & S_ISGID) ≡ S_ISGID;      /* -g */
case TO_FILSTCK: return stat(opnd1,&b1) ≡ 0 ∧ (b1.st_mode & S_ISVTX) ≡ S_ISVTX;      /* -k */
case TO_FILGZ: return stat(opnd1,&b1) ≡ 0 ∧ b1.st_size > 0L;      /* -s */
case TO_FILTT:      /* -t */
  if (opnd1 ∧ ¬bi_getn(opnd1,&res)) {
    te→flags |= TEF_ERROR;
    res ← 0;
  }
  else { res ← isatty(opnd1 ? res : 0); }      /* generate error if in FPOSIX mode? */
  return res;
```

```
case TO_FILUID: return stat(opnd1,&b1) ≡ 0 ∧ b1.st_uid ≡ ksheuid;      /* -O */
case TO_FILGID: return stat(opnd1,&b1) ≡ 0 ∧ b1.st_gid ≡ getegid();    /* -G */
```

1318. Deals with `X_OK` on non-directories when running as root. On most (all?) unixes *access* says everything is executable for root. Avoid this on files by using **stat**.

```
⟨ c_test.c 1292 ⟩ +≡
  static int test_eaccess(const char *path, int amode)
  {
    int res;
    res ← access(path, amode);
    if (res ≡ 0 ∧ ksheuid ≡ 0 ∧ (amode & X_OK)) {
      struct stat statb;
      if (stat(path,&statb) ≡ −1) res ← −1;
      else if (S_ISDIR(statb.st_mode)) res ← 0;
      else res ← (statb.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH)) ? 0 : −1;
    }
    return res;
  }
```

1319. Binary operators. `=`: Strings are equal, `!=`: strings are not equal.

```
⟨ Evaluate and return a test operator 1315 ⟩ +≡
case TO_STEQL:    /* = */
  if (te→flags & TEF_DBRACKET) return gmatch(opnd1, opnd2, false);
  return strcmp(opnd1, opnd2) ≡ 0;
case TO_STNEQ:    /* != */
  if (te→flags & TEF_DBRACKET) return ¬gmatch(opnd1, opnd2, false);
  return strcmp(opnd1, opnd2) ≠ 0;
```

1320. Strings compare "greater" (`>`) or "less" (`<`) based on the ASCII value of their characters.

```
⟨ Evaluate and return a test operator 1315 ⟩ +≡
case TO_STLT:    /* < */
  return strcmp(opnd1, opnd2) < 0;
case TO_STGT:    /* > */
  return strcmp(opnd1, opnd2) > 0;
```

1321. Numeric comparison; operands are equal (`-eq`) or not equal (`-ne`), or the left operand is greater than (`-ge`), greater than or equal to (`-gt`), less than (`-le`) or less than or equal to (`-lt`) the right operand.

```
⟨ Evaluate and return a test operator 1315 ⟩ +≡
case TO_INTEQ:      /* -eq */
case TO_INTNE:      /* -ne */
case TO_INTGE:      /* -ge */
case TO_INTGT:      /* -gt */
case TO_INTLE:      /* -le */
case TO_INTLT:      /* -lt */
  {
    int64_t v1, v2;
    if (¬evaluate(opnd1, &v1, KSH_RETURN_ERROR, false) ∨
          ¬evaluate(opnd2, &v2, KSH_RETURN_ERROR, false)) {
      te¬flags |= TEF_ERROR;      /* error has already been printed */
      return 1;
    }
    switch ((int) op) {
    case TO_INTEQ: return v1 ≡ v2;
    case TO_INTNE: return v1 ≠ v2;
    case TO_INTGE: return v1 ≥ v2;
    case TO_INTGT: return v1 > v2;
    case TO_INTLE: return v1 ≤ v2;
    case TO_INTLT: return v1 < v2;
    }
  }
```

1322. `-nt`: The first file is newer than the second file, `-ot`: the first file is older than the second file. ksh88/ksh93 succeed if the second file is inaccessible (not 'does not exist').

```
⟨ Evaluate and return a test operator 1315 ⟩ +≡
case TO_FILNT:      /* -nt */
  {
    int s2;
    return stat(opnd1, &b1) ≡ 0 ∧
        (((s2 ← stat(opnd2, &b2)) ≡ 0 ∧
        b1.st_mtime > b2.st_mtime) ∨ s2 < 0);
  }
case TO_FILOT:      /* -ot */
  {
    int s1;
    return stat(opnd2, &b2) ≡ 0 ∧
        (((s1 ← stat(opnd1, &b1)) ≡ 0 ∧
        b1.st_mtime < b2.st_mtime) ∨ s1 < 0);
  }
```

1323. `-ef`: The two filenames are the same file.

```
⟨ Evaluate and return a test operator 1315 ⟩ +≡
case TO_FILEQ:      /* -ef */
  return stat(opnd1, &b1) ≡ 0 ∧ stat(opnd2, &b2) ≡ 0 ∧
      b1.st_dev ≡ b2.st_dev ∧ b1.st_ino ≡ b2.st_ino;
```

1324. Errors during testing are reported.

```
⟨ c_test.c 1292 ⟩ +≡
  static void ptest_error(Test_env *te, int offset, const char *msg)
  {
    const char *op ← te→pos.wp + offset ≥ te→wp_end ? Λ : te→pos.wp[offset];
    te→flags |= TEF_ERROR;
    if (op) bi_errorf("%s: %s", op, msg);
    else bi_errorf("%s", msg);
  }
```

1325. *c_test* makes the above test evaluator accessible as a built-in.

```
⟨ c_test.c 1292 ⟩ +≡
  int c_test(char **wp)
  {
    int argc;
    int res;
    Test_env te;
    te.flags ← 0;
    te.isa ← ptest_isa;
    te.getopnd ← ptest_getopnd;
    te.eval ← ptest_eval;
    te.error ← ptest_error;
    for (argc ← 0; wp[argc]; argc++) ;
    if (strcmp(wp[0], "[") ≡ 0) {
      if (strcmp(wp[--argc], "]") ≠ 0) {
        bi_errorf("missing ]");
        return T_ERR_EXIT;
      }
    }
    te.pos.wp ← wp + 1;
    te.wp_end ← wp + argc;
    if (argc ≤ 5) {⟨ test special cases from POSIX 1326 ⟩}
    return test_parse(&te);
  }
```

1326. Implementation of all the rules isn't necessary since our parser does the right thing for the omitted steps. TODO: work out what this is doing and why; it seems to have no bearing on the big picture.

```
⟨ test special cases from POSIX 1326 ⟩ ≡
  char **owp ← wp;
  int invert ← 0;
  Test_op op;
  const char *opnd1, *opnd2;
  while (--argc ≥ 0) {
    if ((*te.isa)(&te, TM_END)) return ¬0;
    if (argc ≡ 3) {
      opnd1 ← (*te.getopnd)(&te, TO_NONOP, 1);
      if ((op ← (Test_op)(*te.isa)(&te, TM_BINOP))) {
        opnd2 ← (*te.getopnd)(&te, op, 1);
        res ← (*te.eval)(&te, op, opnd1, opnd2, 1);
        if (te.flags & TEF_ERROR) return T_ERR_EXIT;
        if (invert & 1) res ← ¬res;
        return ¬res;
      }     /* back up to opnd1 */
      te.pos.wp--;
    }
    if (argc ≡ 1) {
      opnd1 ← (*te.getopnd)(&te, TO_NONOP, 1);
        /* Historically -t by itself tests if fd 1 is a file descriptor, but POSIX says its a string test. */
      if (¬Flag(FPOSIX) ∧ strcmp(opnd1, "-t") ≡ 0) break;
      res ← (*te.eval)(&te, TO_STNZE, opnd1, Λ, 1);
      if (invert & 1) res ← ¬res;
      return ¬res;
    }
    if ((*te.isa)(&te, TM_NOT)) {
      invert++;
    }
    else break;
  }
  te.pos.wp ← owp + 1;
```

This code is used in section 1325.

1327. Double-Bracket Tests ([[...]]). This happens in two parts, first a test environment is set up using the `dbtestp_*` functions to parse the test's source, then another using `dbteste_*` to evaluate it. These are the sequences of tokens recognised in a double-bracket test. The order in *dbtest_tokens* must match **Test_meta**.

Note that "`||`", "`&&`", "`(`" and "`)`" can't appear as unquoted strings in normal shell input so these can be interpreted unambiguously in the evaluation pass.

⟨ `syn.c` 412 ⟩ +≡

```
static const char dbtest_or[] ← {CHAR, '|', CHAR, '|', EOS};
static const char dbtest_and[] ← {CHAR, '&', CHAR, '&', EOS};
static const char dbtest_not[] ← {CHAR, '!', EOS};
static const char dbtest_oparen[] ← {CHAR, '(', EOS};
static const char dbtest_cparen[] ← {CHAR, ')', EOS};
const char *const dbtest_tokens[] ← {dbtest_or, dbtest_and, dbtest_not, dbtest_oparen, dbtest_cparen};
const char db_close[] ← {CHAR, ']', CHAR, ']', EOS};
const char db_lthan[] ← {CHAR, '<', EOS};
const char db_gthan[] ← {CHAR, '>', EOS};
```

1328. ⟨ Externally-linked variables 6 ⟩ +≡

```
extern const char *const dbtest_tokens[];
extern const char db_close[];
```

1329. Borrow the next token and see if it's the type of thing *meta* says it wants.

```
⟨ syn.c 412 ⟩ +≡
  static int dbtestp_isa(Test_env *te, Test_meta meta)
  {
    int c ← tpeek(ARRAYVAR | (meta ≡ TM_BINOP ? 0 : CONTIN));
    int uqword ← 0;
    char *save ← Λ;
    int ret ← 0;
    uqword ← c ≡ LWORD ∧ *ident;      /* unquoted word? */
    if (meta ≡ TM_OR) ret ← c ≡ LOGOR;
    else if (meta ≡ TM_AND) ret ← c ≡ LOGAND;
    else if (meta ≡ TM_NOT) ret ← uqword ∧ strcmp(yylval.cp, dbtest_tokens[(int) TM_NOT]) ≡ 0;
    else if (meta ≡ TM_OPAREN) ret ← c ≡ '(';
    else if (meta ≡ TM_CPAREN) ret ← c ≡ ')';
    else if (meta ≡ TM_UNOP ∨ meta ≡ TM_BINOP) {
      if (meta ≡ TM_BINOP ∧ c ≡ REDIR ∧ (yylval.iop→flag ≡ IOREAD ∨ yylval.iop→flag ≡ IOWRITE)) {
        ret ← 1;
        save ← wdcopy(yylval.iop→flag ≡ IOREAD ? db_lthan : db_gthan, ATEMP);
      }
      else if (uqword ∧ (ret ← (int) test_isop(te, meta, ident))) save ← yylval.cp;
    }
    else ret ← uqword ∧ strcmp(yylval.cp, db_close) ≡ 0;      /* meta ≡ TM_END */
    if (ret) {
      reject ← false;
      if (meta ≠ TM_END) {
        if (¬save) save ← wdcopy(dbtest_tokens[(int) meta], ATEMP);
        XPput(*te→pos.av, save);
      }
    }
    return ret;
  }
```

1330. Unusual tokens are not valid operands otherwise it's copied into *te→pos.av*.

```
⟨ syn.c 412 ⟩ +≡
  static const char *dbtestp_getopnd(Test_env *te, Test_op op, int do_eval)
  {
    int c ← tpeek(ARRAYVAR);
    if (c ≠ LWORD) return Λ;
    reject ← false;
    XPput(*te→pos.av, yylval.cp);
    return null;
  }
```

1331. The test parser "passes" provided it got far enough to try without encountering a syntax error.

```
⟨ syn.c 412 ⟩ +≡
  static int dbtestp_eval(Test_env *te, Test_op op, const char *opnd1, const char *opnd2,
          int do_eval)
  { return 1; }
```

1332. If a token is unexpected it needs to be returned in a kludgy manner reminiscent of **token**/*tpeek*.

```
⟨ syn.c 412 ⟩ +≡
  static void dbtestp_error(Test_env *te, int offset, const char *msg)
  {
    te→flags |= TEF_ERROR;
    if (offset < 0) {
      reject ← true;
      symbol ← LWORD;
      yylval.cp ← *(XPptrv(*te→pos.av) + XPsize(*te→pos.av) + offset);
    }
    syntaxerr(msg);
  }
```

1333. Uses *test_isop* to look for the operators which are understood by traditional single-bracket tests or scans *dbtest_tokens* to look for double-bracket operators. Completely different to *dbtestp_isa* for reasons that aren't clear but probably have something to do with the tokens having been stored already by the parser.

```
⟨ exec.c 461 ⟩ +≡
  static int dbteste_isa(Test_env *te, Test_meta meta)
  {
    int ret ← 0;
    int uqword;
    char *p;
    if (¬*te→pos.wp) return meta ≡ TM_END;
    for (p ← *te→pos.wp; *p ≡ CHAR; p += 2) ; uqword ← *p ≡ EOS;      /* unquoted word? */
    if (meta ≡ TM_UNOP ∨ meta ≡ TM_BINOP) {
      if (uqword) {
        char buf[8];      /* longer than the longest operator */
        char *q ← buf;
        for (p ← *te→pos.wp; *p ≡ CHAR ∧ q < &buf[sizeof (buf) − 1]; p += 2) *q++ ← p[1];
        *q ← '\0';
        ret ← (int) test_isop(te, meta, buf);
      }
    }
    else if (meta ≡ TM_END) ret ← 0;
    else ret ← uqword ∧ strcmp(*te→pos.wp, dbtest_tokens[(int) meta]) ≡ 0;
    if (ret) te→pos.wp++;      /* "Accept" the token. */
    return ret;
  }
```

1334. This is where patterns within a test get their special tokenisation abilities.

```
⟨ exec.c 461 ⟩ +≡
  static const char *dbteste_getopnd(Test_env *te, Test_op op, int do_eval)
  {
    char *s ← *te→pos.wp;
    if (¬s) return Λ;
    te→pos.wp ++;
    if (¬do_eval) return null;
    if (op ≡ TO_STEQL ∨ op ≡ TO_STNEQ) s ← evalstr(s, DOTILDE | DOPAT);      /* else */
    else s ← evalstr(s, DOTILDE);
    return s;
  }
```

1335.

```
⟨ exec.c 461 ⟩ +≡
  static int dbteste_eval(Test_env *te, Test_op op, const char *opnd1, const char *opnd2, int
          do_eval)
  {
    return test_eval(te, op, opnd1, opnd2, do_eval);
  }
```

1336.

```
⟨ exec.c 461 ⟩ +≡
  static void dbteste_error(Test_env *te, int offset, const char *msg)
  {
    te→flags |= TEF_ERROR;
    internal_warningf("%s:␣%s␣(offset␣%d)", __func__, msg, offset);
  }
```

1337. Menu Selection. Who knew this was in here? The menu is printed if this is the first time around the select loop, the user enters a blank line, or the `$REPLY` parameter is empty.

```
⟨ exec.c 461 ⟩ +≡
  static char *do_selectargs(char **ap, bool print_menu)
  {
    static const char *const read_args[] ← {"read", "-r", "REPLY", Λ};
    const char *errstr;
    char *s;
    int i, argct;
    for (argct ← 0; ap[argct]; argct++) ;
    while (1) {
      if (print_menu ∨ ¬*str_val(global("REPLY"))) pr_menu(ap);
      shellf("%s", str_val(global("PS3")));
      if (call_builtin(findcom("read", FC_BI), (char **) read_args)) return Λ;
      s ← str_val(global("REPLY"));
      if (*s) {
        i ← strtonum(s, 1, argct, &errstr);
        if (errstr) return null;
        return ap[i − 1];
      }
      print_menu ← 1;
    }
  }
```

1338.

/* Width/column calculations were done once and saved, but this * means select can't be used recursively so we re-calculate each * time (could save in a structure that is returned, but its probably * not worth the bother). */

* we will print an index of the form * * in front of each entry * get the max width of this */

```
⟨ exec.c 461 ⟩ +≡
  int pr_menu(char *const *ap)
  {
    struct select_menu_info smi;
    char *const *pp;
    int nwidth, dwidth;
    int i, n;
    ⟨ Get dimensions of the select list 1339 ⟩
    smi.args ← ap;
    smi.arg_width ← nwidth;
    smi.num_width ← dwidth;
    print_columns(shl_out, n, select_fmt_entry, (void *) &smi, dwidth + nwidth + 2, 1);
    return n;
  }
```

1339. ⟨ Get dimensions of the `select` list 1339 ⟩ ≡

```
  for (n ← 0, nwidth ← 0, pp ← ap; *pp; n++, pp++) {
    i ← strlen(*pp);
    nwidth ← (i > nwidth) ? i : nwidth;
  }
  for (i ← n, dwidth ← 1; i ≥ 10; i /= 10) dwidth++;
```

This code is used in section 1338.

1340. Format a single `select` menu item.

```
⟨ Type definitions 17 ⟩ +≡
  struct select_menu_info {
    char *const *args;
    int arg_width;
    int num_width;
  };
```

1341.

```
⟨ exec.c 461 ⟩ +≡
  static char *select_fmt_entry(void *arg, int i, char *buf, int buflen)
  {
    struct select_menu_info *smi ← (struct select_menu_info *) arg;
    shf_snprintf(buf, buflen, "%*d) %s", smi→num_width, i + 1, smi→args[i]);
    return buf;
  }
```

1342. Not used by the `select` menu code, *pr_list* is similar enough to *pr_menu* to sit alongside it. They could perhaps be moved to "Formatted Output" in I/O.

```
⟨ exec.c 461 ⟩ +≡
  int pr_list(char *const *ap)
  {
    char *const *pp;
    int nwidth;
    int i, n;
    for (n ← 0, nwidth ← 0, pp ← ap; *pp; n++, pp++) {
      i ← strlen(*pp);
      nwidth ← (i > nwidth) ? i : nwidth;
    }
    print_columns(shl_out, n, plain_fmt_entry, (void *) ap, nwidth + 1, 0);
    return n;
  }
```

1343. Print a single list item.

```
⟨ exec.c 461 ⟩ +≡
  static char *plain_fmt_entry(void *arg, int i, char *buf, int buflen)
  {
    shf_snprintf(buf, buflen, "%s", ((char *const *) arg)[i]);
    return buf;
  }
```

1344. Index.

!=: 1182, 1295, 1301.
!=: 1319.
!: 340, 370, 414, 415, 428, 506, 512, 1177, 1307.
!(...): 330, 336, 340, 341, 363, 414, 442.
"...": 330, 336, 340, 352, 361, 362.
"${⟨var⟩}": 340, 356.
"` ... `": 350.
(...): 414, 427, 444, 506.
((...)): 330, 332, 337, 339, 340, 360, 366, 415, 451.
(): 448, 449, 450, 506, 532.
(...): 1177.
(...|...): 330, 336, 357, 363, 414, 442.
*(...): 330, 336, 340, 341, 363, 414, 442.
*: 1180.
+(...): 330, 336, 340, 341, 363, 414, 442.
++: 1177.
+: 1177, 1180.
,: 1185.
--: 1177.
-: 1177, 1180.
-a: 1295, 1300, 1306, 1317.
-b: 1295, 1300, 1317.
-c: 26, 27, 1295, 1300, 1317.
-d: 1295, 1300, 1317.
-e: 1295, 1300, 1317.
-ef: 1295, 1301, 1323.
-eq: 1295, 1301, 1321.
-f: 1295, 1300, 1317.
-g: 1295, 1300, 1317.
-G: 1295, 1300, 1317.
-ge: 1295, 1301, 1321.
-gt: 1295, 1301, 1321.
-h: 1295, 1300, 1317.
-H: 1295, 1300, 1317.
-i: 29, 30, 673.
-k: 1295, 1300, 1317.
-L: 1295, 1300, 1317.
-le: 1295, 1301, 1321.
-lt: 1295, 1301, 1321.
-m: 26, 673.
-n: 1295, 1300, 1315.
-ne: 1295, 1301, 1321.
-nt: 1295, 1301, 1322.
-o: 1295, 1300, 1305, 1316.
-O: 1295, 1300, 1317.
-ot: 1295, 1301, 1322.
-p: 25, 1295, 1300, 1317.
-r: 1295, 1300, 1317.
-s: 26, 29, 1295, 1300, 1317.
-S: 1295, 1300, 1317.
-t: 1295, 1300, 1317.
-u: 1295, 1300, 1317.
-w: 1295, 1300, 1317.
-x: 1295, 1300, 1317.
-z: 1295, 1300, 1315.
.: 1212.
/: 1180.
:#: 347.
:%: 347.
::: 450, 1205.
;;: 366, 415, 442.
;: 414, 420, 440, 506, 508.
<: 1320.
<<: 1181.
<</<<-: 330, 337, 340, 361, 362.
<=: 1182.
<: 1182, 1295, 1301.
==: 1295, 1301.
=: 337, 1179, 1182, 1185, 1295, 1301, 1319.
>: 1320.
>=: 1182.
>>: 1181.
>: 1182, 1295, 1301.
>/>&/>|/</<&/>>/<</<<-/<>: 338, 364, 365, 415, 427, 444.
?(...): 330, 336, 340, 341, 363, 414, 442.
?:: 1184.
@(...): 330, 336, 340, 341, 363, 414, 442.
[: 1325.
[[: 1325.
[[...]]: 414, 415, 452, 506, 513, 1329.
␣/⟨TAB⟩: 339.
#: 330, 339, 340, 347, 357.
#!: 523.
$: 345.
$!: 118, 147.
$(...): 5, 330, 332, 336, 340, 345, 346, 353, 565.
$((...)): 330, 332, 336, 340, 346, 354.
$*: 1224.
$-: 118, 147.
$?: 118, 147.
$@: 1224.
$#: 118, 147, 1224.
$$: 5, 118, 147.
${⟨var⟩[#%]...}: 330, 340, 347, 357.
${⟨var⟩}: 330, 336, 340, 345, 347, 355, 375.
$_: 525.
$⟨digit⟩: 118, 145, 146, 349, 534.
$⟨var⟩: 345, 348.
$CDPATH: 1234.
$COLUMNS: 149, 157, 860, 872.

$EDITOR: 149, 157, 325.
$ENV: 124.
$EXECSHELL: 523.
$FPATH: 533, 545, 549, 552, 763.
$HISTCONTROL: 149, 157, 160, 817.
$HISTFILE: 149, 157, 815, 819, 839.
$HISTSIZE: 149, 154, 157, 818, 822.
$HOME: 61, 612, 1231.
$IFS: 149, 152, 156, 160, 188, 189.
$KSH_VERSION: 13, 24, 61, 192.
$LINENO: 81, 149, 154, 159, 160, 465.
$LINES: 872.
$MAIL: 149, 158, 160, 1192.
$MAILCHECK: 61, 149, 158, 160.
$MAILPATH: 149, 158, 160, 1192.
$OLDPWD: 61, 612, 1232, 1235.
$OPTARG: 37, 1280.
$OPTIND: 61, 149, 154, 156, 1280.
$PATH: 61, 81, 124, 149, 156, 160, 545, 549, 763.
$POSIXLY_CORRECT: 149, 156, 160.
$PPID: 61, 192.
$PS1: 5, 81, 190.
$PS2: 81, 188.
$PS3: 188, 1337.
$PS4: 188, 463, 465.
$PWD: 61, 191, 612, 743, 1233, 1235, 1236.
$RANDOM: 61, 149, 154, 159, 160, 203.
$REPLY: 1337.
$SECONDS: 61, 149, 154, 159, 160.
$SH_VERSION: 13, 24, 61, 192.
$SHELL: 61, 124.
$TERM: 149, 157.
$TMOUT: 61, 149, 159, 160, 325, 645, 656, 660.
$TMPDIR: 81, 149, 156, 160.
$VISUAL: 149, 157, 325.
$0: 5, 118, 145, 146, 534.
%: 330, 340, 347, 357, 1180.
%;: 496.
%c: 490.
%d: 491.
%N: 496.
%R: 497.
%S: 493.
%s: 492.
%T: 495.
%u: 494.
&: 414, 420, 506, 510, 1183.
&&: 366, 414, 415, 419, 506, 511.
\: 337, 343, 344, 358, 361, 362.
\!: 405.
\": 342, 344, 358, 362.
\’: 342.
\@: 398.
\[: 409.
\#: 406.
\$: 343, 344, 358, 362, 382, 393.
\\: 342, 343, 344, 358, 362, 408.
\`: 343, 358, 362.
nl: 320, 344.
\]: 410.
\‘: 344.
\A: 399.
\a: 384.
\D: 386.
\d: 385.
\e: 387.
\H: 389.
\h: 382, 388.
\j: 390.
\l: 391.
\n: 392.
\p: 382, 393.
\r: 394.
\s: 395.
\T: 397.
\t: 396.
\u: 5, 400.
\V: 402.
\v: 401.
\W: 404.
\w: 403.
\0: 407.
\1: 407.
\2: 407.
\3: 407.
\4: 407.
\5: 407.
\6: 407.
\7: 407.
^: 1183.
{ ... }: 414, 427, 438, 443, 450, 506.
~: 403, 565.
'...': 330, 336, 340, 351, 361.
nl: 366, 420, 440.
` ... `: 5, 330, 332, 340, 350, 358, 565.
⟨*fn*⟩(): 448, 449, 450, 506, 532.
⟨*array*⟩[...]: 337, 374.
⟨*array*⟩[...]=⟨*word*⟩: 340, 371.
⟨*var*⟩=⟨*word*⟩: 337, 340.
|&: 366, 414, 415, 420, 506, 509.
|: 414, 418, 506, 507, 1183.
||: 366, 414, 415, 419, 506, 511.
__**attribute**__: 4, 243, 313, 412, 1158.
__**dead**: 304.

__format__: 243, 313.
__func__: 58, 164, 229, 232, 247, 249, 251, 274, 275, 281, 282, 283, 284, 285, 286, 287, 295, 504, 536, 633, 643, 665, 685, 686, 687, 690, 693, 703, 711, 716, 721, 726, 727, 729, 731, 797, 1146, 1243, 1281, 1288, 1336.
__noreturn__: 4, 243, 313, 412, 1158.
_CS_PATH: 193.
_PATH_BSHELL: 523.
_PATH_DEFPATH: 193.
_POSIX_VDISABLE: 869, 870.
_PW_NAME_LEN: 5.
a: 141, 778, 1043, 1110.
aa: 766, 778.
abs: 1173.
access: 156, 548, 765, 1236, 1318.
acos: 1173.
add: 881.
add_glob: 766, 772, 773, 796.
adj: 926, 971, 1014.
aedit: 890.
AEDIT: 890, 903, 906, 992, 1016.
AF_ARGS_ALLOCED: 106.
AF_ARGV_ALLOC: 106.
afree: 70, 78, 94, 113, 114, 121, 130, 131, 132, 141, 144, 156, 160, 163, 164, 172, 195, 221, 226, 245, 246, 252, 254, 308, 323, 334, 369, 371, 376, 426, 444, 453, 467, 500, 502, 548, 551, 553, 558, 565, 600, 646, 648, 745, 746, 772, 776, 780, 783, 796, 811, 816, 817, 818, 819, 821, 822, 828, 906, 992, 1030, 1042, 1152, 1173, 1174, 1194, 1196, 1220, 1230, 1234, 1236, 1260, 1264, 1266, 1267.
afreeall: 70, 79, 113, 197.
Again: 338, 366, 369, 455, 456.
age: 672, 690, 727.
ai: 215, 216.
AI_ARGC: 106.
AI_ARGV: 106.
ainit: 15, 70, 74, 111, 112, 185, 1016.
alarm: 325, 645, 660.
alarm_catcher: 659, 660.
alarm_init: 66, 638, 659.
alen: 198.
ALIAS: 337, 339, 367, 369, 426, 428, 429, 430, 431, 432, 436, 438, 440, 442, 443, 444, 450.
alias: 1260.
alias: 567, 1149, 1150, 1260, 1263, 1264.
aliases: 87, 183, 184, 186, 369, 774, 1150, 1244, 1260, 1265.
all: 553, 1265, 1285.
ALLOC: 85, 127, 129, 130, 131, 132, 139, 141, 164, 172, 532, 542, 544, 548, 551, 553, 613, 1264, 1266, 1267.
alloc: 70, 75, 77, 93, 111, 112, 132, 137, 165, 193, 212, 213, 221, 245, 249, 251, 272, 308, 365, 368, 417, 429, 450, 499, 501, 503, 574, 575, 581, 610, 614, 718, 719, 746, 903, 1040, 1152, 1187, 1197, 1233.
allow_cur: 832, 834, 835.
alt_expand: 554, 587, 606, 610.
amode: 1318.
anchored: 836, 837, 1147.
andor: 412, 418, 419, 420, 430.
any: 1077, 1109.
any_set: 1257, 1258.
any_subst: 829.
aok: 198, 199, 201.
ap: 74, 75, 76, 77, 78, 79, 89, 212, 213, 272, 463, 464, 465, 499, 500, 501, 502, 503, 506, 514, 516, 517, 520, 521, 523, 524, 525, 526, 527, 528, 529, 530, 531, 534, 535, 557, 613, 905, 1149, 1150, 1152, 1260, 1262, 1263, 1264, 1265, 1266, 1267, 1337, 1338, 1339, 1342.
aperm: 15, 60, 72, 73.
APERM: 72, 148, 156, 160, 186, 187, 193, 416, 548, 551, 553, 613, 646, 648, 718, 719, 745, 815, 816, 818, 819, 821, 822, 828, 1030, 1031, 1040, 1042, 1152, 1194, 1195, 1196, 1197, 1264, 1266, 1267.
app: 467.
approx: 834, 835.
Area: 10, 15, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 84, 88, 89, 105, 107, 197, 212, 213, 218, 237, 243, 272, 313, 316, 368, 378, 469, 470, 499, 500, 501, 502, 503, 523, 566, 890.
area: 72, 105, 107, 111, 112, 113, 185, 195, 197, 221, 1224.
areallocarray: 60, 70, 76, 94, 101, 226, 276, 333, 426, 450, 453, 466, 499, 501, 776, 795, 1224.
areap: 84, 88, 89, 93, 94, 112, 118, 130, 131, 132, 137, 139, 141, 145, 164, 171, 172, 218, 219, 221, 224, 237, 245, 246, 249, 251, 252, 254, 276, 316, 319, 368, 369, 370, 371, 532, 542, 544, 856, 1187.
aresize: 70, 76, 77, 219, 224, 745, 746, 1031.
arg: 38, 52, 56, 308, 461, 1201, 1206, 1222, 1223, 1277, 1341, 1343.
arg_info: 105, 106.
arg_width: 1338, 1340.
argc: 15, 26, 59, 60, 105, 112, 146, 147, 228, 229, 534, 595, 597, 1206, 1212, 1224, 1280, 1282, 1325, 1326.
*argc_*: 106.
argcnt: 1077, 1078, 1083, 1085, 1103, 1106, 1107, 1108, 1109, 1115, 1116, 1121, 1122, 1123, 1125,

1126, 1127, 1130, 1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1142, 1144, 1153, 1156.
argct: 1337.
argc1: 1063, 1066, 1068, 1069, 1074, 1075.
argc2: 1063, 1069, 1071.
argi: 15, 26, 27, 28, 59, 105, 1224.
args: 289, 290, 291, 294, 295, 296, 414, 417, 426, 440, 444, 446, 448, 450, 451, 452, 465, 470, 472, 499, 500, 513, 516, 521, 523, 535, 600, 897, 902, 903, 906, 922, 1224, 1261, 1338, 1340, 1341.
argv: 15, 26, 27, 28, 32, 37, 38, 39, 40, 41, 42, 43, 45, 59, 60, 67, 105, 106, 112, 146, 228, 229, 465, 514, 534, 560, 597, 1206, 1212, 1224, 1281.
arith: 172, 178, 1161, 1168, 1169, 1170, 1174, 1178, 1179, 1186.
array: 43, 45, 84, 93, 116, 119, 120, 127, 136, 137, 138, 141, 595, 598, 795, 1257, 1258.
`ARRAY`: 86, 101, 121, 131, 136, 595, 794, 1258, 1259.
array_index_calc: 103, 116, 119, 144.
array_ref: 131.
array_ref_len: 104, 122, 143, 144, 198, 1173.
arrayname: 104, 121, 142, 595, 598.
arrayp: 144.
arraysearch: 103, 116, 119, 120, 136, 140.
arrayset: 43, 45.
arraysub: 314, 371, 374, 376.
`ARRAYVAR`: 337, 340, 439, 444, 1329, 1330.
as: 215, 216.
asin: 1173.
asprintf: 794.
assign_check: 1158, 1178, 1179, 1188.
assign_command: 412, 446, 447.
async_job: 669, 698, 703, 721, 725, 726, 727.
async_pid: 669, 725, 726.
at: 900, 901, 914, 919, 921, 922, 923, 924.
atan: 1173.
`ATEMP`: 27, 28, 29, 51, 72, 101, 111, 112, 113, 114, 118, 121, 123, 142, 144, 145, 163, 165, 171, 174, 217, 226, 227, 230, 245, 249, 251, 308, 323, 333, 334, 339, 365, 369, 371, 373, 374, 376, 378, 417, 426, 429, 439, 444, 450, 451, 453, 466, 467, 551, 558, 565, 567, 574, 575, 581, 587, 595, 600, 604, 605, 610, 611, 614, 632, 633, 745, 746, 757, 763, 772, 773, 775, 776, 780, 781, 783, 784, 786, 795, 796, 797, 811, 817, 829, 845, 847, 854, 1173, 1174, 1187, 1215, 1220, 1230, 1233, 1234, 1236, 1237, 1260, 1329.
atoi: 838.
attempts: 231, 235.
`autoload`: 61.
av: 452, 1299, 1329, 1330, 1332.
a1: 907, 910.
a2: 907, 912.
b: 746, 778, 1043, 1110.
`B_`: 1026, 1027.
b_ops: 1301, 1304.
backslash_skip: 314, 319, 320, 339.
backword: 1023, 1091, 1121, 1135, 1145.
Backword: 1023, 1121, 1138.
`BAD`: 1162, 1163, 1175.
`BANG`: 415, 428.
base: 102, 121, 127, 172, 174, 175, 176, 177, 178, 179, 180, 181, 182, 214, 332, 333, 334, 339, 350, 573, 574, 579, 580, 581, 776, 777, 778, 1250, 1253, 1255.
basename: 391, 404.
basestr: 1250, 1252, 1253.
bb: 766, 778.
bcount: 1078, 1132.
beg: 218, 219, 221, 224, 225, 226.
`BEL`: 863, 925, 927, 928, 932, 933, 936, 937, 942, 943, 951, 952, 953, 958, 974, 975, 981, 985, 986, 988, 996, 997, 1006, 1007, 1013, 1019, 1035, 1089.
`BF_DOGETOPTS`: 105, 113, 534.
bg: 696, 699, 1270.
`bg`: 1270.
bi_errorf: 20, 37, 39, 40, 43, 46, 216, 243, 301, 696, 698, 699, 707, 717, 739, 820, 829, 832, 833, 835, 836, 840, 843, 844, 845, 846, 848, 854, 856, 903, 907, 920, 1206, 1209, 1210, 1212, 1214, 1215, 1221, 1223, 1225, 1228, 1230, 1231, 1232, 1233, 1234, 1236, 1239, 1243, 1253, 1255, 1268, 1270, 1271, 1272, 1273, 1278, 1281, 1282, 1285, 1290, 1324, 1325.
bi_getn: 10, 216, 1223, 1253, 1274, 1317.
`bind`: 1279.
`BIT`: 7, 32, 85, 86, 87, 105, 107, 209, 318, 337, 364, 463, 468, 524, 565, 634, 643, 766, 785, 1237, 1302.
`BITS`: 7, 174, 289.
block: 412, 418, 419, 420, 428, 429, 436, 460.
Block: 2, 15, 105, 107, 111, 112, 113, 114, 116, 117, 119, 133, 541, 771, 1206, 1224, 1250.
block_pipe: 638, 654, 655, 1242.
blocking_read: 10, 279, 281, 874.
bnest: 807, 809, 810.
bourne_function_call: 524, 529.
brace_end: 606, 610.
brace_start: 606, 610.
bracktype: 1023, 1132, 1133.
`break`: 515, 1223.
`BREAK`: 366, 415, 442.
Break: 463, 466, 515.

bsize: 237, 245, 247, 249, 251, 282, 283, 295, 746, 878, 880.
buf: 32, 35, 39, 40, 52, 56, 214, 237, 245, 247, 249, 251, 252, 253, 254, 274, 276, 277, 278, 279, 281, 282, 283, 285, 287, 295, 329, 461, 732, 733, 735, 746, 770, 791, 792, 803, 804, 805, 806, 873, 878, 879, 880, 914, 915, 1012, 1024, 1025, 1028, 1029, 1030, 1031, 1100, 1118, 1119, 1152, 1201, 1208, 1277, 1280, 1333, 1341, 1343.
buflen: 52, 56, 461, 770, 791, 803, 805, 1201, 1277, 1341, 1343.
`builtin`: 526, 1229.
builtin: 187, 462, 540, 1199, 1200, 1201, 1202.
builtin_argv0: 301, 537, 538, 539.
builtin_flag: 301, 537, 538, 539.
builtin_opt: 33, 34, 528, 539, 844, 845, 1206, 1207, 1212, 1213, 1214, 1220, 1221, 1222, 1223, 1225, 1230, 1236, 1239, 1243, 1250, 1252, 1253, 1254, 1255, 1260, 1261, 1265, 1269, 1270, 1273, 1279, 1281, 1285, 1288.
builtins: 87, 183, 184, 187, 536, 540, 546, 774.
b1: 1314, 1317, 1322, 1323.
b2: 1314, 1322, 1323.
c: 37, 47, 119, 143, 178, 200, 209, 217, 285, 286, 289, 308, 319, 320, 321, 338, 373, 374, 375, 380, 419, 420, 423, 426, 438, 440, 442, 443, 455, 459, 479, 480, 489, 504, 505, 565, 594, 601, 622, 748, 807, 821, 874, 875, 925, 928, 929, 930, 932, 933, 934, 935, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 955, 956, 957, 958, 962, 964, 965, 972, 974, 975, 976, 977, 978, 979, 980, 982, 983, 988, 991, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1007, 1009, 1010, 1011, 1012, 1013, 1015, 1018, 1019, 1020, 1028, 1052, 1061, 1062, 1156, 1172, 1214, 1240, 1329, 1330.
`C_`: 1026, 1027.
c_alias: 1201, 1204, 1260.
`C_ALPHA`: 205, 209, 345, 348.
c_bind: 1201, 1204, 1279.
c_brkcont: 515, 1200, 1203, 1223.
c_builtin: 524, 1200, 1203, 1229.
c_cc: 869.
c_cd: 1201, 1204, 1230.
c_command: 524, 528, 1201, 1204, 1249.
`C_DIGIT`: 209.
c_dot: 1200, 1203, 1212.
c_eval: 1200, 1203, 1220.
c_exec: 524, 629, 630, 1200, 1203, 1227.
c_exitreturn: 1200, 1203, 1222.
c_fc: 813, 844, 857, 1201.
c_fc_depth: 812, 844, 849, 856, 857.
c_fc_reset: 232, 813, 849, 856, 857.
c_fgbg: 1201, 1204, 1270.
c_getopts: 1201, 1204, 1280.
c_iflag: 869.
`C_IFS`: 152, 156, 160, 209, 210, 328, 586, 1215, 1216.
`C_IFSWS`: 209, 328, 586, 1215, 1216.
c_jobs: 1201, 1204, 1269.
c_kill: 1201, 1204, 1271.
c_label: 1200, 1203, 1205.
c_let: 1201, 1204, 1268.
`C_LEX1`: 209, 338, 803.
c_lflag: 869.
c_list: 412, 418, 420, 423, 430, 431, 432, 436, 437, 438, 442.
c_print: 250, 654, 655, 1201, 1204, 1237.
c_pwd: 1201, 1204, 1236.
`C_QUOTE`: 209, 307, 450.
c_read: 250, 266, 1200, 1203, 1214.
c_set: 1200, 1203, 1224.
c_shift: 1200, 1203, 1206.
`C_SUBOP1`: 209, 596.
`C_SUBOP2`: 209, 594, 596.
c_suspend: 1200, 1228.
c_test: 1200, 1294, 1325.
c_times: 1200, 1203, 1226.
c_trap: 1200, 1203, 1221.
c_type: 1201, 1204, 1249.
c_typeset: 1201, 1204, 1224, 1250.
c_ulimit: 1200, 1203, 1285.
c_umask: 1200, 1203, 1207.
c_unalias: 1201, 1204, 1261, 1265.
c_unset: 1200, 1203, 1225.
`C_VAR1`: 209, 345, 376, 599.
c_wait: 1200, 1203, 1213.
c_whence: 545, 1201, 1204, 1243, 1249.
`CALIAS`: 84, 1244, 1260.
call_builtin: 461, 531, 536, 539, 1337.
calloc: 76.
can_seek: 29, 243, 255, 1214.
`CASE`: 415, 441.
`case`: 414, 415, 441, 506, 520.
caselist: 412, 441, 443.
casepart: 412, 442, 443.
`cat`: 61.
cb: 869.
`CBRACE`: 208, 586, 607, 610.
`CBRACK`: 208.
cbuf: 1028, 1031, 1036, 1038, 1039, 1040, 1041, 1042, 1043, 1049, 1050, 1054, 1059, 1067, 1076, 1077, 1079, 1083, 1085, 1089, 1090, 1091, 1099, 1100, 1103, 1106, 1107, 1110, 1118,

1119, 1122, 1124, 1125, 1126, 1129, 1130, 1132, 1134, 1135, 1136, 1137, 1138, 1139, 1145, 1146, 1147, 1153, 1156, 1157.
cbufsize: 1031, 1036, 1040, 1099, 1100, 1145, 1146, 1147, 1153, 1157.
cc: 61.
cc: 286, 621, 1012.
cclass: 207, 615, 622.
CClass: 2, 204, 621.
cclasses: 204, 621.
cd: 1230.
cdnode: 1230, 1234, 1235.
cdpath: 1230, 1234.
cdpathp: 747, 749.
CEXEC: 84, 535, 550, 1244.
cf: 338, 339, 340, 342, 344, 366, 367, 369, 412, 418, 426, 444, 445, 454, 459.
CFUNC: 84, 529, 532, 541, 552, 1244.
ch: 905, 1051, 1053, 1054, 1063, 1064, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072, 1088, 1099, 1124, 1133, 1145.
Chain: 471, 472.
change_flag: 10, 20, 23, 46, 58, 156, 868.
change_random: 104, 203, 689.
CHAR: 199, 200, 202, 336, 338, 342, 343, 344, 345, 347, 351, 361, 362, 367, 371, 450, 451, 481, 504, 505, 561, 568, 569, 596, 611, 1327, 1333.
CHAR_BIT: 7.
char_len: 1023, 1051, 1052, 1145.
chdir: 745, 1234.
check: 785, 786, 787, 788, 789, 790.
check_fd: 243, 261, 269, 626, 1214, 1239.
check_job: 668, 701, 703, 715.
check_sigwinch: 860, 862, 867, 872, 873.
child_max: 669, 677, 726.
CHILD_MAX: 669, 677.
chmod: 61.
ci: 1058.
CKEYWD: 84, 416, 1244.
classify: 1026, 1027.
cleanup_parents_env: 4, 115, 195, 693.
cleanup_proc_env: 4, 196, 521.
clear_screen: 967, 1045.
cleared: 967.
cleartraps: 638, 652, 693.
clock_gettime: 154, 159, 468, 1192.
CLOCK_MONOTONIC: 154, 159, 468, 1192.
close: 195, 246, 252, 253, 258, 259, 260, 266, 267, 268, 270, 463, 509, 600, 623, 630, 631, 664, 665, 689, 693, 839, 1227.
close_fd: 463, 689.
CLOSE_PAREN: 1163, 1177.
closedir: 790.
closepipe: 243, 260, 507.
clr: 121, 125, 127, 128.
cmd: 791, 792, 793, 794, 821, 826, 1077, 1078, 1083, 1102, 1103, 1121, 1122, 1123, 1127, 1143, 1150, 1151, 1152.
cmd_offset: 316, 368, 406, 840.
cmd_opts: 43, 44.
cmdlen: 791, 792, 793, 794, 1063, 1066, 1068, 1069, 1071, 1072.
CMDWORD: 337, 339, 444.
CNONE: 84.
cnt: 1053, 1056, 1124.
col: 932, 942, 991, 1049, 1051, 1053, 1054, 1055, 1056, 1058.
col_width: 308, 310.
colon: 621.
cols: 308, 309, 310.
comexec: 461, 506, 524, 1227, 1229, 1243.
comm: 227.
comma: 606, 607.
command: 528, 1243, 1249.
command: 4, 227, 646, 670, 689, 697, 723, 724, 735, 736, 826, 855, 1118, 1119.
Comp_type: 884, 1005, 1006.
compile: 233, 412, 413, 418, 424, 600.
COMPLETE: 1111, 1119.
complete_word: 1023, 1094, 1098, 1101, 1115, 1116, 1119.
completed: 1006.
comsub: 554, 570, 600.
COMSUB: 336, 346, 350, 353, 354, 480, 483, 504, 505, 565, 568.
CONFIG_H: 11.
confstr: 193.
content: 631, 633.
CONTIN: 337, 366, 418, 419, 438, 440, 442, 443, 450, 1329.
continue: 515, 1223.
Coproc: 2, 262, 263, 264.
coproc: 263, 264, 265, 266, 267, 268, 269, 270, 509, 694, 705, 1215, 1242.
COPROC: 366, 415, 420.
coproc_cleanup: 243, 270, 509, 693.
coproc_getfd: 243, 261, 269, 705, 1214, 1239.
Coproc_id: 262, 672.
coproc_id: 672, 690, 694, 705.
coproc_init: 15, 243, 265.
coproc_read_close: 243, 266, 630, 1215.
coproc_readw_close: 243, 266, 267, 705, 1215.
coproc_write_close: 243, 268, 630, 705, 1242.
copy: 116.

coredumped: 732, 733, 735.
cos: 1173.
cosh: 1173.
count: 606, 607, 610, 903, 1076, 1119, 1120.
counting: 380, 381, 382, 383, 409, 410.
cp: 51, 143, 314, 331, 367, 369, 379, 380, 381, 382, 386, 392, 394, 407, 411, 439, 440, 441, 442, 446, 449, 451, 454, 457, 463, 517, 520, 524, 526, 529, 530, 533, 535, 555, 556, 558, 559, 565, 567, 600, 612, 623, 624, 625, 626, 627, 628, 633, 696, 699, 707, 717, 722, 723, 724, 739, 769, 784, 791, 793, 797, 821, 926, 927, 931, 936, 937, 938, 958, 959, 960, 961, 963, 966, 974, 975, 988, 989, 990, 992, 1172, 1173, 1174, 1175, 1207, 1208, 1209, 1210, 1211, 1212, 1214, 1215, 1216, 1218, 1233, 1279, 1329, 1330, 1332.
`cp`: 61.
`CPAREN`: 208.
`CPAT`: 336, 363, 486, 504, 505, 568.
cpy: 575.
`CQUOTE`: 336, 351, 352, 360, 362, 480, 484, 504, 505, 561, 568.
create: 541.
cru0: 468.
cru1: 468.
cs: 1214, 1215, 1216, 1218.
`CSHELL`: 84, 466, 524, 531, 540, 630, 1244.
csstate: 332, 346, 353, 354.
`CSUBST`: 336, 348, 349, 355, 356, 357, 485, 504, 505, 562, 568, 572, 578, 594, 596, 611.
`CT_COMPLETE`: 998, 1000, 1002, 1005.
`CT_COMPLIST`: 1004, 1005, 1006.
`CT_LIST`: 999, 1001, 1003, 1005, 1006.
`CTALIAS`: 84, 535, 545, 550, 1244, 1248, 1260.
`CTERN`: 1163, 1184.
`CTRL`: 882, 891, 892, 893, 894, 895, 896, 901, 983, 1017, 1021, 1044, 1046, 1065, 1072, 1088, 1093, 1094, 1095, 1096, 1097, 1098, 1113, 1114, 1115, 1116, 1117, 1125, 1141, 1142, 1145.
ctype: 205, 307, 328, 338, 345, 348, 376, 450, 586, 594, 596, 599, 803, 1215, 1216.
ctypes: 205, 206, 209, 210.
cur: 226, 753, 754, 756, 780, 783, 1049, 1053, 1054, 1077, 1083, 1085, 1106, 1107.
cur_col: 1031, 1037, 1047, 1053, 1056, 1057, 1058.
cur_prompt: 81, 82, 328, 378.
cur_term: 157, 967, 1045.
curcmd: 1063, 1066, 1068, 1069, 1070, 1071, 1072, 1074.
curr: 136, 137, 138, 720, 721.
current: 812, 830, 831.
current_lineno: 81, 82, 154, 159, 465, 646.
current_wd: 65, 191, 743, 744, 745, 1233, 1234, 1235, 1236.
current_wd_size: 743, 744, 745.
cursig: 634, 642, 644, 654.
cursor: 1031, 1036, 1039, 1040, 1041, 1049, 1050, 1054, 1059, 1065, 1066, 1067, 1076, 1077, 1079, 1080, 1082, 1083, 1085, 1086, 1089, 1090, 1091, 1099, 1100, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1118, 1119, 1121, 1122, 1124, 1125, 1126, 1127, 1132, 1134, 1135, 1136, 1137, 1138, 1139, 1144, 1145, 1146, 1147, 1156, 1157.
curstate: 1170, 1171.
cwd: 747, 750.
c1: 1077, 1083, 1102, 1103, 1143.
c2: 320, 338, 341, 354, 360, 365, 366, 370, 1077, 1102, 1103, 1143.
c3: 1077, 1143.
d: 622, 768, 790.
d_name: 790.
`date`: 61.
`DB_AND`: 470.
`DB_BE`: 470.
db_close: 1327, 1328, 1329.
db_gthan: 1327, 1329.
db_lthan: 1327, 1329.
`DB_NORM`: 470.
`DB_OR`: 470.
`DB_PAT`: 470.
`DBRACKET`: 415, 452.
dbtest_and: 1327.
dbtest_cparen: 1327.
dbtest_not: 1327.
dbtest_oparen: 1327.
dbtest_or: 1327.
dbtest_tokens: 1327, 1328, 1329, 1333.
dbteste_error: 461, 513, 1336.
dbteste_eval: 461, 513, 1335.
dbteste_getopnd: 461, 513, 1334.
dbteste_isa: 461, 513, 1333.
dbtestp_error: 412, 452, 1332.
dbtestp_eval: 412, 452, 1331.
dbtestp_getopnd: 412, 452, 1330.
dbtestp_isa: 412, 452, 1329, 1333.
`DEBUG`: 885, 889, 966, 1015.
debunk: 555, 579, 581, 587, 609, 614, 768, 769, 789.
def_path: 81, 82, 160, 193, 549.
`DEFAULT_ENV`: 11, 65.
Define: 2, 462, 506, 542, 1225.
`DEFINED`: 85, 90, 94, 96, 99, 101, 116, 118, 119, 131, 136, 139, 145, 148, 416, 540, 541, 543, 550, 552, 571, 613, 1168, 1260, 1266, 1267.
del_curterm: 157.

del_range: 1023, 1043, 1067, 1080, 1083, 1085, 1086, 1102, 1104, 1105, 1106, 1107, 1118, 1119.
delim: 364, 365, 373, 454, 471, 478, 501, 502.
delimiter: 380, 381.
delimitthis: 380, 381, 382, 383.
depth: 143, 200, 374.
difference: 154.
digit: 42, 122, 145, 167, 205, 290, 338, 345, 348, 376, 599, 641, 722, 1172, 1207, 1271, 1290.
digits: 174, 175, 176, 177, 292.
dir: 272, 1230, 1231, 1232, 1233, 1234.
DIR: 790.
dirent: 790.
dirp: 790.
disable_subst: 5, 6, 558, 1170, 1171, 1186.
display: 1023, 1048, 1053.
dlen: 768.
do: 415, 438.
DO: 415, 438.
do_clear_screen: 1023, 1044, 1045, 1096.
do_close: 623, 626, 665.
do_complete: 884, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006.
do_eval: 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1330, 1331, 1334, 1335.
do_gmatch: 9, 614, 615, 616, 617, 618, 619.
do_open: 623, 625, 626.
do_phys_path: 741, 757, 758, 762.
do_ppmm: 1158, 1177, 1178.
do_selectargs: 461, 517, 1337.
DOASNTILDE: 529, 565, 567, 577, 587, 592, 593, 611.
doblank: 565, 567, 570, 572, 579, 580, 581, 582, 583, 584, 585, 586.
DOBLANK: 465, 514, 565, 567, 570, 572, 576, 577, 579, 580, 581, 582, 583, 584, 585, 586.
DOBRACE_: 565, 567, 576, 577, 587, 590.
DOGLOB: 465, 514, 559, 561, 564, 565, 567, 577, 586, 587, 588, 589, 609, 623, 797.
dogroup: 412, 437, 438, 439.
DOMAGIC_: 561, 564, 565, 586, 587, 588, 589, 590.
DOMARKDIRS: 565, 567, 587, 609, 797.
domove: 1023, 1077, 1078, 1082, 1086, 1102, 1103.
Done: 338, 360.
done: 415, 438.
done: 565, 586.
DONE: 415, 438.
DONTRUNCOMMAND: 565, 570, 571, 576.
DOPAT: 520, 565, 573, 576, 587, 588, 589, 772, 1334.
dopprompt: 314, 379, 380.
doprint: 380, 381, 382, 383.
DOTEMP_: 565, 572, 576, 577, 591, 592.
DOTILDE: 65, 465, 514, 520, 565, 567, 572, 576, 577, 587, 593, 600, 623, 772, 797, 1334.
dotty: 692.
DOVACHECK: 446, 565, 567.
dowrite: 821.
dp: 101, 338, 365, 367, 561, 565, 567, 570, 571, 572, 574, 576, 579, 580, 581, 582, 586, 587, 588, 589, 590, 593, 611, 612, 768.
dp_x: 593.
dpp: 611.
dropped_privileges: 9, 20.
ds: 561, 565, 567, 570, 571, 572, 574, 579, 580, 581, 582, 586, 587, 593.
dsp: 611.
dst: 329.
dummy: 368, 463.
dup2: 256, 629.
dwidth: 1338, 1339.
DYNAMIC: 275.
e: 1059, 1184.
E_: 1026, 1027.
E_ERRH: 108, 236, 378, 509, 631, 1170.
E_EXEC: 108, 463, 509.
E_FUNC: 108, 112, 236, 532, 1222, 1223.
E_INCL: 108, 228, 236, 1222, 1223.
E_LOOP: 107, 108, 236, 515, 1223.
E_NONE: 108, 115, 185, 236, 1223.
E_PARSE: 108, 231, 236, 1223.
EACCES: 765.
EAGAIN: 279, 691, 825.
early_assign: 127.
EB_GROW: 275, 276, 286, 287.
EB_READSW: 275, 277, 281, 285.
EBADF: 256, 257.
ebuf: 1022, 1031.
ec: 1017.
ECHO: 869.
echo: 1237.
ecode: 696, 707, 717, 739, 1214, 1215, 1217.
ecodep: 722, 723, 724.
ed: 61.
ed: 868.
ed_mov_opt: 1023, 1053, 1056, 1057, 1058.
edchars: 862, 865, 867, 869, 870, 948, 1028, 1033, 1034, 1088, 1145.
edit_flags: 868.
edit_reset: 1023, 1029, 1031.
editor: 844, 845, 846, 855.
edstate: 1022, 1023, 1036, 1040, 1041, 1042, 1059, 1077, 1118, 1119, 1145.
EF_BRKCONT_PASS: 107, 515, 1223.
EF_FAKE_SIGDIE: 107, 115, 236.

EF_FUNC_PARSE: 107, 450.
EINTR: 245, 278, 281, 286, 287, 327, 650, 825, 874, 1217, 1242.
EINVAL: 761.
EISDIR: 765, 783.
elapsed: 1192.
ele: 43, 44, 50, 51, 54, 55, 57, 58, 868, 903, 904.
elen: 1233.
elif: 414, 432, 506, 519.
ELIF: 415, 432.
else: 415, 431, 432.
ELSE: 415, 432.
elsepart: 412, 431, 432.
emacs: 61.
EMACS: 11, 17, 21, 23, 48, 325, 867, 868, 870, 873, 882, 1201, 1279.
emacs/abort: 1018.
emacs/auto-insert: 928.
emacs/backward-char: 942.
emacs/backward-word: 940.
emacs/beginning-of-history: 976.
emacs/beginning-of-line: 944.
emacs/capitalize-word: 957.
emacs/clear-screen: 965.
emacs/comment: 951.
emacs/complete: 1000.
emacs/complete-command: 998.
emacs/complete-file: 1002.
emacs/complete-list: 1004.
emacs/delete-char-backward: 932.
emacs/delete-char-forward: 933.
emacs/delete-word-backward: 934.
emacs/delete-word-forward: 935.
emacs/down-history: 979.
emacs/downcase-word: 956.
emacs/end-of-history: 977.
emacs/end-of-line: 945.
emacs/eot: 948.
emacs/eot-or-delete: 949.
emacs/error: 1019.
emacs/exchange-point-and-mark: 996.
emacs/expand-file: 1007.
emacs/forward-char: 943.
emacs/forward-word: 941.
emacs/goto-history: 980.
emacs/kill-line: 950.
emacs/kill-region: 997.
emacs/kill-to-eol: 991.
emacs/list: 1001.
emacs/list-command: 999.
emacs/list-file: 1003.
emacs/macro-string: 929.
emacs/newline: 946.
emacs/newline-and-next: 947.
emacs/no-op: 1020.
emacs/prev-hist-word: 988.
emacs/quote: 930.
emacs/redraw: 964.
emacs/search-character-backward: 974.
emacs/search-character-forward: 975.
emacs/search-history: 983.
emacs/set-arg: 925.
emacs/set-mark-command: 995.
emacs/transpose-chars: 952.
emacs/up-history: 978.
emacs/upcase-word: 955.
emacs/yank: 993.
emacs/yank-pop: 994.
emark: 436.
emit_word: 584, 587.
empty: 112.
emsg: 626, 1214, 1237, 1239.
emsgp: 261, 269.
encoded: 824, 840, 843.
end: 218, 219, 221, 223, 225, 226, 283, 329, 332, 333, 334, 339, 578, 601, 602, 603, 604, 605, 606, 609, 610, 803, 805, 806, 1006, 1007, 1059, 1118, 1119.
END: 1162, 1163, 1170, 1172, 1188.
endc: 578.
endp: 770.
endtok: 442.
endword: 1023, 1122, 1136.
Endword: 1023, 1122, 1139.
ENOEXEC: 521.
entry: 897, 899, 903, 906, 909, 910, 911, 912, 913, 921, 982.
env: 15, 133, 185.
Env: 2, 15, 107, 109, 110, 111, 114, 195, 196, 541, 706, 1222, 1223.
env_file: 65.
environ: 3, 189.
eof: 373, 859, 867, 869, 870, 948, 1028, 1034.
EOF: 252, 253, 274, 275, 276, 278, 281, 282, 283, 284, 285, 286, 287, 288, 289, 415, 528, 585, 631, 820, 854, 1215, 1216, 1217, 1218.
eofp: 373.
EOS: 201, 336, 367, 371, 450, 451, 480, 503, 504, 505, 561, 564, 568, 578, 611, 1327, 1333.
ep: 111, 114, 115, 195, 196, 621, 706, 1222, 1223.
EPERM: 1290.
EPIPE: 1242.
erase: 859, 867, 869, 870, 1017, 1088, 1145.
err: 699, 765, 1212.

ERR: 157, 236, 967, 1045.
errexit: 15, 64, 67.
errline: 305, 316, 368, 456.
errno: 28, 146, 237, 256, 257, 274, 275, 278, 279, 281, 286, 287, 521, 523, 533, 628, 630, 631, 632, 640, 660, 665, 685, 686, 687, 691, 698, 699, 701, 707, 711, 729, 731, 761, 765, 814, 818, 825, 843, 854, 874, 1212, 1234, 1236, 1242, 1278, 1290.
*errno_*: 84, 237, 247, 249, 251, 274, 275, 278, 281, 286, 287, 327, 533, 535, 547, 549, 640, 660, 701, 856, 1217.
errnop: 763, 764, 765.
errok: 256.
error: 11, 105, 112, 452, 513, 807, 1299, 1302, 1308, 1309, 1311, 1312, 1314, 1325.
error_ok: 127, 130, 163, 1168, 1170, 1171.
error_prefix: 243, 299, 300, 301, 302, 305, 335.
error_type: 1158, 1160, 1161.
errorf: 15, 27, 28, 32, 119, 124, 125, 127, 140, 144, 163, 182, 229, 243, 256, 257, 259, 299, 300, 335, 466, 467, 509, 521, 523, 526, 578, 579, 594, 595, 600, 691, 1171.
errstr: 722, 1337.
es: 1022, 1028, 1031, 1034, 1038, 1039, 1043, 1049, 1050, 1054, 1057, 1065, 1066, 1067, 1076, 1077, 1079, 1080, 1082, 1083, 1085, 1086, 1089, 1090, 1091, 1095, 1099, 1100, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1113, 1118, 1119, 1121, 1122, 1124, 1125, 1126, 1127, 1129, 1130, 1131, 1132, 1134, 1135, 1136, 1137, 1138, 1139, 1144, 1145, 1146, 1147, 1153, 1154, 1156, 1157, 1161, 1162, 1170, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1183, 1184, 1186, 1188.
esac: 337, 415, 443.
ESAC: 369, 415, 443.
ESACONLY: 337, 369, 443.
ET_BADLIT: 1160, 1161, 1174.
ET_LVALUE: 1160, 1161, 1188.
ET_RDONLY: 1160, 1161, 1188.
ET_RECURSIVE: 1160, 1161, 1186.
ET_STR: 1160, 1161, 1173, 1177, 1179, 1184.
ET_UNEXPECTED: 1160, 1161, 1170, 1177.
eval: 414, 452, 465, 513, 514, 555, 557, 1299, 1308, 1311, 1312, 1325, 1326.
eval: 1220.
eval_done: 464.
eval_execute_args: 464.
evalerr: 1158, 1161, 1170, 1173, 1174, 1177, 1179, 1184, 1186, 1188.
evalexpr: 1158, 1170, 1176, 1177, 1179, 1183, 1184.
evalflags: 414, 417, 444, 446, 465, 499.
evaling: 1169, 1170, 1171, 1186.
evalonestr: 555, 559, 623.
evalstr: 373, 471, 520, 529, 555, 556, 558, 559, 600, 633, 772, 1334.
evaluate: 144, 1159, 1168, 1206, 1268, 1290, 1321.
exchild: 463, 507, 509, 535, 667, 689.
exec: 245, 257, 414, 630, 634, 643, 648, 653, 711.
exec: 521, 527, 1227.
execute: 233, 462, 463, 468, 506, 508, 510, 511, 512, 516, 517, 518, 519, 520, 532, 600, 689, 693, 1220, 1224.
execve: 196, 521, 523.
exit: 16, 26, 105, 112, 115.
exit: 1222.
EXIT: 236.
exp: 1173.
exp_start: 606.
EXPAND: 1111, 1118.
Expand: 554, 565, 566, 594, 600.
expand: 189, 555, 556, 557, 559, 560, 565, 566, 614, 797, 1214, 1216.
expand_mode: 1111, 1112.
expand_word: 1023, 1093, 1117, 1118.
expanded: 1089, 1090, 1091, 1092, 1099, 1112, 1118, 1119.
expanding: 1214, 1215, 1216, 1218, 1219.
expect: 566.
expected: 478.
EXPORT: 84, 85, 86, 103, 117, 125, 126, 127, 133, 134, 164, 189, 529, 535, 1215, 1245, 1246, 1248, 1251, 1252, 1253, 1258, 1259, 1260, 1266, 1267, 1280.
export: 103, 126, 132, 164.
export: 1250.
expr: 1168, 1170.
Expr_state: 1158, 1161, 1169, 1170, 1172, 1176, 1178, 1186, 1188.
expr_state: 1169.
expression: 1161, 1162, 1169, 1170, 1172.
EXPRINEVAL: 86, 1169, 1171, 1186.
EXPRLVALUE: 86, 1173, 1186, 1188.
EXPRSUB: 336, 346, 354, 483, 504, 505, 568.
exstat: 5, 6, 28, 115, 147, 229, 231, 233, 299, 301, 463, 466, 524, 532, 646, 647, 1220, 1222.
f: 20, 84, 468, 556, 557, 558, 559, 565, 573, 642, 648, 839, 983, 1255.
F_DUPFD: 246.
F_DUPFD_CLOEXEC: 257, 665.
F_GETFL: 248, 261, 280.
F_SETFD: 247, 249, 257, 1227.
F_SETFL: 280.
factor: 1284, 1290, 1291.
fake_assign: 127, 128.

false: 1205.
false: 43, 65, 121, 122, 144, 157, 182, 227, 230, 338, 371, 415, 423, 444, 448, 451, 452, 454, 464, 507, 509, 517, 520, 532, 533, 547, 600, 602, 603, 604, 605, 645, 665, 685, 686, 687, 693, 729, 731, 773, 775, 818, 844, 849, 850, 851, 873, 874, 920, 926, 931, 932, 933, 969, 973, 981, 994, 1006, 1007, 1033, 1161, 1177, 1198, 1206, 1220, 1221, 1255, 1290, 1319, 1321, 1329, 1330.
fatal_trap: 635, 636, 640, 645, 652, 710.
fatal_trap_check: 638, 650, 1217.
FBGNICE: 17, 693.
FBRACEEXPAND: 17, 20, 23, 567.
FC_BI: 465, 524, 526, 528, 546, 1243, 1337.
FC_DEFPATH: 524, 528, 545, 549, 550, 1243.
FC_FUNC: 465, 524, 545, 549, 1243.
FC_PATH: 524, 528, 545, 1243.
FC_REGBI: 524, 545.
FC_SPECBI: 524, 546.
FC_UNREGBI: 524, 545.
fcflags: 524, 528, 530, 1243, 1244.
fclose: 841.
fclr: 1250, 1252, 1253, 1254, 1255.
fcntl: 245, 246, 247, 248, 249, 257, 261, 280, 665, 1227.
FCOMMAND: 17, 26.
FCSHHISTORY: 17, 340.
fd: 114, 157, 195, 237, 245, 246, 247, 248, 249, 251, 252, 253, 255, 257, 258, 261, 266, 267, 268, 269, 272, 274, 275, 277, 279, 280, 281, 286, 287, 629, 631, 632, 693, 706, 839, 856, 1214, 1215, 1218, 1237, 1239, 1242.
FD_CLOEXEC: 247, 249, 257, 1227.
FDBASE: 7, 245, 246, 257, 665.
fddup: 839.
FDELETE: 87, 532, 543.
fdir: 1230, 1233, 1234.
fdo: 561, 564, 565, 567, 586, 587, 588, 589, 590, 606, 609, 610.
fdopen: 839.
FEMACS: 17, 21, 23, 325, 868, 873.
feof: 820.
FERREXIT: 17, 64, 67, 236, 463, 466, 1220.
ferror: 820.
FEXPORT: 17, 529, 1215, 1280.
fflush: 824, 841, 843.
fg: 1270.
fgetc: 815.
fgets: 840.
FGMACS: 17, 21, 325, 868, 873, 952.
fi: 415, 430.
FI: 415, 430.
field: 84, 93, 117, 121, 127, 139, 166, 167, 168, 289, 290, 291, 293, 1250, 1253, 1255, 1259.
field_start: 606, 610.
fieldstr: 1250, 1252, 1253.
FIGNOREEOF: 17, 232, 235.
file: 28, 29, 30, 230, 305, 316, 368, 747, 748, 752, 1212.
fileline: 305.
fileno: 823, 824, 825, 843.
filler: 732, 734, 736.
findch: 1023, 1123, 1124.
findcom: 462, 465, 524, 526, 530, 545, 1244, 1249, 1337.
findfunc: 462, 533, 541, 542, 543, 547, 1255.
findhist: 370, 813, 836, 837, 1147.
findhistrel: 370, 813, 838.
finished: 701.
FINUSE: 87, 94, 532, 542.
first: 844, 845, 848, 849, 851, 925, 1192.
first_insert: 1022, 1029, 1088, 1092.
FKEYWORD: 17, 446.
FKSH: 87, 529, 534, 542, 1255, 1256.
fl: 261.
FL_BLANK: 289, 290, 292.
FL_DOT: 289, 290, 291, 293.
FL_HASH: 289, 290, 292.
FL_LLONG: 289, 290, 291.
FL_LONG: 289, 290, 291.
FL_NUMBER: 289, 291, 293.
FL_PLUS: 289, 290, 292.
FL_RIGHT: 289, 290, 293.
FL_SHORT: 289, 290, 291.
FL_UPPER: 289, 292.
FL_ZERO: 289, 290, 291, 293.
Flag: 18, 20, 21, 22, 23, 24, 25, 26, 29, 30, 31, 36, 43, 45, 46, 51, 56, 57, 63, 64, 65, 66, 67, 115, 124, 231, 232, 233, 234, 235, 236, 301, 325, 339, 340, 366, 382, 446, 447, 463, 465, 525, 528, 529, 530, 532, 535, 545, 567, 594, 595, 600, 623, 624, 627, 642, 648, 673, 677, 682, 683, 684, 685, 688, 689, 690, 693, 694, 703, 709, 713, 737, 740, 815, 844, 873, 952, 1052, 1054, 1061, 1062, 1098, 1114, 1116, 1145, 1195, 1198, 1215, 1218, 1220, 1224, 1227, 1228, 1230, 1236, 1238, 1270, 1274, 1280, 1316, 1326.
flag: 38, 53, 55, 56, 84, 90, 93, 94, 95, 96, 99, 101, 103, 113, 116, 117, 118, 119, 121, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 139, 140, 141, 145, 146, 147, 148, 150, 154, 156, 157, 158, 159, 163, 164, 165, 166, 167, 168, 171, 172, 173, 174, 175, 178, 179, 190, 194, 364, 365, 369, 373, 416, 454, 466, 471, 478, 529, 532, 533,

534, 535, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 550, 551, 552, 553, 565, 571, 575, 577, 595, 598, 600, 613, 623, 624, 625, 626, 630, 645, 794, 795, 1150, 1168, 1170, 1171, 1173, 1174, 1178, 1179, 1186, 1187, 1188, 1192, 1215, 1225, 1244, 1245, 1246, 1247, 1248, 1250, 1252, 1255, 1256, 1257, 1258, 1259, 1260, 1262, 1263, 1264, 1266, 1267, 1269, 1280, 1329.
flags: 30, 32, 35, 38, 39, 40, 44, 46, 47, 58, 105, 106, 107, 111, 112, 113, 115, 231, 233, 234, 236, 237, 244, 245, 247, 248, 249, 251, 252, 254, 274, 275, 276, 277, 278, 280, 281, 282, 283, 284, 285, 286, 287, 289, 290, 291, 292, 293, 316, 318, 321, 324, 325, 339, 340, 368, 421, 450, 452, 463, 464, 465, 496, 506, 507, 508, 509, 510, 511, 512, 513, 515, 516, 517, 518, 519, 520, 524, 527, 532, 534, 535, 545, 546, 549, 550, 623, 624, 627, 634, 639, 640, 642, 643, 644, 645, 646, 647, 648, 650, 651, 652, 653, 654, 659, 672, 682, 684, 688, 689, 690, 692, 693, 694, 695, 698, 699, 700, 701, 703, 709, 710, 711, 712, 713, 715, 716, 717, 721, 725, 726, 727, 732, 739, 740, 766, 770, 771, 774, 781, 783, 796, 867, 1006, 1223, 1237, 1238, 1239, 1240, 1242, 1252, 1260, 1299, 1302, 1305, 1306, 1307, 1308, 1309, 1317, 1319, 1321, 1324, 1325, 1326, 1332, 1336.
flock: 825.
FLOGIN: 17, 43, 65, 688, 737, 1228.
flushcom: 156, 160, 462, 553, 1235.
FMARKDIRS: 17, 567.
FMONITOR: 17, 20, 26, 232, 535, 673, 677, 682, 683, 688, 689, 693, 694, 703, 709, 713, 732, 740, 1270.
fmt: 289, 290, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 335, 487, 488, 489.
FNFLAGS: 17, 18, 19, 51.
fnmatch: 204.
FNOCLOBBER: 17, 624.
FNOEXEC: 17, 233.
FNOGLOB: 17, 567.
FNOHUP: 17, 22, 688, 737.
FNOLOG: 17.
FNOTIFY: 17, 703.
FNOUNSET: 17, 594, 595.
for: 414, 415, 439, 506, 514, 516.
FOR: 415, 433, 439.
fork: 115, 195, 203, 414, 509, 535, 634, 652, 691, 711.
forksleep: 689, 691.
formatstr: 103, 164, 165, 174.
forw: 1124.
forwword: 1023, 1127, 1134.
Forwword: 1023, 1127, 1137.
found: 622, 701, 702.
Found: 520.
fpath: 84, 533, 545, 547, 549, 552, 771, 774, 1246.
FPHYSICAL: 17, 1230, 1236.
FPIPEFAIL: 17, 690, 714.
FPOSIX: 17, 20, 23, 36, 45, 156, 301, 339, 447, 545, 722, 1195, 1220, 1224, 1238, 1270, 1274, 1326.
fprintf: 824, 843.
FPRIVILEGED: 17, 20, 25, 65.
fptreef: 470, 471, 472, 473, 474, 475, 476, 477, 478, 487, 1255, 1256.
fread: 282.
free: 71, 78, 79, 794, 824, 843, 906.
free_edstate: 1022, 1041, 1042, 1118, 1119.
free_jobs: 669, 718, 721.
free_me: 127, 129, 130.
free_procs: 669, 719, 721.
freelist: 71, 74, 75, 77, 78, 79.
freeme: 316, 323, 370, 371.
freep: 1236.
FRESTRICTED: 17, 64, 67, 124, 528, 530, 627, 1230.
fs: 163, 164.
fsavech: 1022, 1123.
fsavecmd: 1022, 1123.
fseeko: 824.
fset: 1250, 1251, 1252, 1253, 1254, 1255.
FSH: 17, 24, 366, 382, 525, 535, 1198, 1227.
fstat: 31, 255, 823, 824, 839, 843, 856.
FSTDIN: 17, 26, 29.
ftab: 897, 902, 903, 920, 922, 982.
FTALKING: 17, 20, 30, 31, 63, 65, 66, 115, 231, 232, 325, 525, 600, 642, 648, 677, 682, 684, 685, 693, 694, 815.
FTALKING_I: 17, 20, 30, 48, 623, 844, 1218.
ftp: 533.
FTRACKALL: 17, 66, 545.
ftruncate: 843.
full: 1047, 1060.
func: 187, 308, 540, 903, 904, 905, 1199, 1250, 1252, 1253, 1255.
function: 414, 415, 448, 449, 450, 506, 532.
FUNCTION: 415, 449.
function_body: 412, 448, 449, 450.
functions: 87.
functions: 61.
fundcom: 545.
fundfunc: 541.
funs: 105, 112, 185, 541, 774, 1256.
FVERBOSE: 17, 45, 234.
FVI: 17, 21, 325, 868, 873.
FVIESCCOMPLETE: 17, 1114.
FVIRAW: 17.

FVISHOW8: 17, 1052, 1054, 1061, 1062, 1145.
FVITABCOMPLETE: 17, 23, 1098, 1116.
fwd: 837, 1147.
fwrite: 287.
FXTRACE: 17, 45, 465, 529, 532, 623.
generate: 217.
genv: 59, 72, 109, 110, 111, 112, 113, 114, 116, 119, 120, 121, 133, 185, 195, 196, 197, 228, 229, 232, 236, 378, 450, 466, 507, 509, 514, 515, 532, 534, 541, 595, 597, 623, 629, 631, 632, 706, 774, 854, 1170, 1206, 1212, 1222, 1223, 1224, 1227, 1256, 1257, 1281.
get: 370.
get_brace_var: 314, 347, 375.
get_command: 412, 418, 425, 426, 450.
get_phys_path: 742, 757, 1235, 1236.
getcwd: 191, 746.
getegid: 25, 1317.
geteuid: 190.
getgid: 20, 25.
gethere: 314, 366, 372.
gethostname: 388, 389.
getint: 104, 157, 172, 178, 182.
getlogin: 194.
getn: 10, 215, 216, 641, 834, 1222, 1278.
getopnd: 452, 513, 1299, 1308, 1311, 1312, 1325, 1326.
Getopt: 10, 32, 33, 34, 35, 37, 43, 105, 467.
getopt: 37.
getoptions: 10, 51, 147.
getopts: 1280.
getopts: 160.
getopts_reset: 36, 59, 156, 534, 1204.
getopts_state: 105, 113, 534.
getpgid: 1228.
getpgrp: 115, 685.
getpid: 189, 691.
getppid: 192, 1228.
getpwnam: 613.
getrlimit: 1290, 1291.
getrtusage: 1283.
getrusage: 468, 701, 1226.
getsc: 320, 338, 339, 341, 342, 344, 345, 346, 347, 348, 354, 358, 360, 361, 362, 365, 366, 370, 373, 374, 375, 376.
*getsc_*: 320, 321.
getsc__: 314, 320, 321, 324.
getsc_bn: 314, 320.
getsc_line: 314, 320, 322, 325.
getsid: 1228.
getspec: 103, 154, 173, 178.
gettrap: 638, 641, 1221, 1272, 1273.
getuid: 25, 839.
GF_ERROR: 32, 37, 39, 40, 43, 539, 1261, 1285.
GF_EXCHECK: 785, 786, 787, 790.
GF_GLOBBED: 785, 786, 788, 790.
GF_MARKDIR: 784, 785, 786, 788, 790.
GF_NONAME: 32, 36, 39, 40.
GF_NONE: 784, 785, 790.
GF_PLUSOPT: 32, 36, 37, 38, 43, 1252, 1260.
gflag: 844, 845, 849.
GI_MINUS: 32, 38.
GI_MINUSMINUS: 32, 38, 45, 1239, 1253, 1260.
GI_PLUS: 32, 37, 38, 45, 1252, 1260, 1280.
gid: 20.
gid_t: 20.
glob: 204, 554, 564, 587, 609, 769.
glob_path: 766, 774, 781.
glob_str: 555, 769, 781, 784.
glob_table: 766, 774, 775.
global: 65, 67, 104, 113, 119, 121, 140, 145, 157, 190, 191, 192, 193, 194, 378, 403, 404, 465, 516, 517, 523, 529, 547, 549, 595, 597, 598, 599, 612, 623, 774, 794, 815, 829, 1173, 1192, 1215, 1225, 1230, 1231, 1234, 1280, 1337.
globbing: 807.
globit: 554, 784, 785, 789, 790.
gmatch: 10, 520, 602, 603, 604, 605, 614, 775, 790, 1319.
GNU: 711.
go: 35, 37, 38, 39, 40, 41, 42, 43, 45, 46.
got_sigwinch: 234, 636, 862, 867, 871, 872.
grabhist: 1023, 1140, 1141, 1142, 1146, 1155.
grabsearch: 1023, 1143, 1147.
grep: 61.
h: 83, 90, 91, 96, 116, 119, 133, 369, 370, 541, 545, 631, 1260.
HARD: 1285, 1290, 1291.
has_globbing: 10, 614, 785, 807.
hash: 61.
hash: 80, 83, 94, 96, 116, 119, 133, 148, 150, 151, 369, 416, 533, 536, 540, 542, 545, 613, 1150, 1244, 1255, 1260, 1266.
hashbang: 523.
have_base: 178, 179, 180.
have_sep: 420.
have_tty: 325.
held_sigchld: 669, 695, 701.
HEREDELIM: 337, 339, 361, 454.
heredoc: 364, 365, 373, 471, 501, 502, 625.
heredoc: 330, 337, 339, 340, 361, 362, 372, 373.
HEREDOC: 337, 342, 344, 633.
herein: 461, 625, 628, 631.
herep: 314, 315, 372, 424, 454.

heres: 314, 315, 372, 424, 454.
HERES: 313, 314, 315, 454.
hfirst: 844, 850, 851, 852, 853, 854.
hist: 1147.
hist_execute: 812, 826, 829, 856.
hist_finish: 115, 813, 842.
hist_get: 812, 834, 837, 849, 850, 851.
hist_get_newest: 370, 813, 832, 835, 849, 850, 851.
hist_get_oldest: 812, 833, 835.
hist_init: 66, 812, 813, 815, 819.
hist_replace: 812, 829, 849.
hist_source: 812, 815, 816, 819, 823, 826, 828, 835, 853, 856.
histbackup: 812, 826, 827.
histbase: 812, 814, 818.
histfh: 812, 815, 820, 821, 823, 824, 825, 840, 841, 843.
histnum: 813, 831, 1029, 1144, 1146, 1147.
history: 314, 315, 370, 812, 814, 816, 818, 821, 822, 828, 831, 832, 833, 835, 836, 837, 838, 840, 843, 918, 976, 980, 981, 986.
history: 61.
history_close: 812, 819, 820, 841, 842.
history_load: 812, 815, 823, 840.
history_lock: 812, 815, 823, 824, 825.
history_open: 812, 815, 839.
history_write: 812, 824, 840, 843.
HISTORYSIZE: 814.
histpos: 813, 830, 1144, 1146, 1147.
histptr: 314, 315, 812, 814, 816, 818, 821, 822, 828, 831, 832, 833, 835, 836, 837, 838, 843, 853, 915, 918, 947, 977, 980, 981, 986, 988.
histreset: 816, 819, 823.
histsave: 328, 813, 821, 826, 840, 1153, 1215, 1241.
histsize: 154, 314, 315, 812, 814, 818, 822, 843.
hlast: 844, 850, 851, 852, 853, 854, 1022, 1029, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1102, 1104, 1106, 1107, 1108, 1109, 1118, 1119, 1140, 1143, 1144, 1146, 1147, 1153, 1156.
HMAGIC1: 815, 820.
HMAGIC2: 815, 820.
hname: 812, 815, 819, 839.
hnum: 1022, 1029, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1102, 1104, 1106, 1107, 1108, 1109, 1118, 1119, 1140, 1141, 1142, 1143, 1144, 1153, 1155, 1156.
holdbuf: 1037, 1038, 1039, 1147.
holdcol1: 1050.
holdcol2: 1050.
holdcur1: 1050.
holdcur2: 1050.
holdlen: 1031, 1037, 1038, 1039.
homedir: 183, 184, 554, 612, 613.
homedirs: 183, 184, 186, 613, 1260, 1265.
how: 601, 732, 733, 734, 735, 736, 739, 1222, 1285, 1287, 1288, 1289, 1290, 1291.
hp: 816, 818, 829, 834, 835, 836, 837, 843, 844, 849, 853, 854, 981, 986.
hptr: 1146, 1147.
hstarted: 812, 815, 819.
i: 15, 43, 52, 56, 60, 84, 94, 101, 113, 122, 133, 140, 148, 210, 228, 231, 236, 308, 331, 370, 461, 463, 467, 472, 496, 501, 524, 631, 639, 640, 641, 645, 646, 650, 651, 652, 653, 682, 683, 689, 718, 776, 780, 781, 791, 799, 802, 811, 873, 878, 900, 905, 907, 931, 939, 950, 961, 966, 986, 1007, 1078, 1118, 1119, 1145, 1156, 1170, 1175, 1201, 1207, 1212, 1221, 1227, 1240, 1255, 1271, 1277, 1337, 1338, 1341, 1342, 1343.
iam_whence: 1243, 1244, 1245.
ibuf: 1022, 1076, 1087, 1099.
ICANON: 869.
ICRNL: 869.
id: 262, 265, 509, 694, 705, 1225, 1243, 1244, 1245.
ident: 314, 315, 367, 369, 439, 446, 454, 1329.
IDENT: 313, 314, 315, 367.
if: 414, 415, 430, 506, 519.
IF: 415, 430.
ifs: 881.
IFS_IWS: 572, 580, 581, 582, 586.
IFS_NWS: 586.
IFS_QUOTE: 561, 569, 572, 586.
IFS_WORD: 561, 563, 567, 569, 570, 571, 572, 580, 581, 582, 584, 586, 593.
IFS_WS: 561, 562, 563, 567, 584, 586.
ifs0: 152, 153, 156, 160, 565, 584.
IFW_WS: 563.
igncase: 641.
ignore_backslash_newline: 314, 320, 339, 342, 351, 353, 361, 373.
ignoredups: 812, 817, 821.
ignorespace: 812, 817, 821.
ilen: 1233.
IMPORT: 86, 122, 123, 189.
in: 415, 440, 443.
in: 907, 910, 911, 912, 913.
IN: 415, 440, 443.
in_bracket: 807, 808, 809, 810.
inalias: 412, 420, 421.
incl: 1124.
Include: 2, 4, 65, 228, 533, 1212.
incr: 837.
indelimit: 380, 381, 382, 383.

indent: 471, 472, 473, 474, 475, 476, 477, 478, 487, 489, 495, 496, 497.
INDENT: 469, 473, 474, 475, 476, 477.
index: 84, 136, 137, 139, 1258.
indquotes: 332, 350, 358.
info: 32, 35, 37, 38, 45, 776, 1239, 1252, 1253, 1260, 1280.
init_histvec: 15, 813, 814.
INIT_TBLS: 81, 91.
init_ttystate: 665.
init_username: 3, 190, 194.
initcoms: 61, 62, 154, 159.
initctypes: 10, 15, 209.
initifs: 188, 189.
initio: 15, 243, 250.
initio_done: 240, 250, 297.
initkeywords: 15, 413, 416.
initsubs: 188, 189.
inittraps: 15, 638, 639.
initvar: 15, 104, 148.
INLCR: 869.
inquote: 306.
INSERT: 1029, 1063, 1079, 1080, 1081, 1082, 1085, 1086, 1089, 1099, 1102, 1104, 1118, 1119, 1144.
insert: 545, 550, 1022, 1029, 1063, 1065, 1074, 1075, 1076, 1077, 1079, 1080, 1081, 1082, 1084, 1085, 1086, 1089, 1099, 1102, 1104, 1118, 1119, 1144.
inslen: 1022, 1029, 1074, 1076, 1089, 1090, 1091, 1092, 1099.
inspace: 1144.
INT_L: 86, 117, 1254.
INT_MAX: 92, 144, 157, 215, 491, 722.
INT_MIN: 214, 215, 491.
INT_U: 86, 117, 174, 175, 1252, 1254, 1259.
INTEGER: 84, 86, 103, 117, 121, 126, 127, 128, 129, 130, 133, 135, 145, 147, 159, 163, 171, 172, 173, 178, 525, 535, 571, 577, 1168, 1170, 1174, 1178, 1179, 1186, 1187, 1252, 1254, 1258, 1259, 1280.
integer: 61.
interactive: 231, 232, 233, 234, 325, 327, 328.
internal_error_vwarn: 302, 303, 304.
internal_errorf: 58, 75, 76, 77, 92, 164, 229, 232, 243, 247, 249, 251, 274, 275, 281, 282, 283, 284, 285, 286, 287, 295, 298, 304, 532, 536, 558, 567, 586, 633, 643, 690, 794, 814.
internal_warningf: 243, 303, 504, 693, 703, 716, 721, 726, 727, 797, 1146, 1281, 1288, 1336.
interval: 1193.
intr: 638, 859, 867, 869, 870, 1017, 1028, 1033.
intr_ok: 228, 229.
intrcheck: 638, 649, 785, 1242.
intrsig: 635, 636, 640, 645, 649, 652, 660, 691.
intval: 104, 156, 157, 158, 159, 182.
intvar: 1158, 1170, 1177, 1178, 1179, 1183, 1186.
int64_t: 84, 104, 144, 154, 157, 171, 172, 174, 178, 182, 192, 489, 872, 1159, 1168, 1176, 1190, 1193, 1206, 1268, 1290, 1291, 1321.
invert: 1326.
io: 600.
ioact: 414, 417, 453, 466, 471, 499, 500, 600.
IOCAT: 364, 365, 478, 623.
IOCLOB: 364, 365, 478, 624.
iocopy: 469, 499, 501.
ioctl: 872.
IODUP: 364, 365, 478, 623, 628, 630.
IOEVAL: 364, 373, 454, 625.
iofree: 469, 500, 502.
IOHERE: 364, 365, 454, 471, 478, 623, 628.
IONAMEXP: 364, 478, 623.
iop: 331, 365, 373, 426, 453, 454, 457, 471, 478, 502, 623, 624, 625, 626, 629, 630, 1329.
iopn: 426, 445, 446, 448, 453.
iops: 426, 445, 453.
ior: 501.
IORDUP: 364, 365, 478, 626, 630.
IORDWR: 364, 365, 478, 623.
IOREAD: 364, 365, 478, 600, 623, 628, 1329.
iosetup: 461, 466, 559, 623, 631, 632.
IOSKIP: 364, 365, 373, 478.
iotmp: 623, 626, 630.
IOTYPE: 364, 454, 471, 478, 600, 623.
iotype: 623, 628, 630.
iow: 501, 502.
ioword: 314, 315, 331, 364, 365, 372, 373, 412, 414, 426, 453, 454, 461, 463, 469, 471, 478, 497, 501, 502, 600, 623.
iowp: 463, 466.
IOWRITE: 364, 365, 478, 623, 1329.
IS_ASSIGNOP: 1163, 1179.
is_bad: 1027, 1067.
is_base: 338.
IS_BINOP: 1163, 1176.
is_cfs: 936, 989, 990.
is_cmd: 1027, 1064.
is_command: 770, 799, 1006, 1007, 1059, 1119, 1120.
is_commandp: 770, 803, 806.
is_extend: 1027, 1064.
is_first: 514, 517.
is_long: 1027, 1064.
is_mfs: 936, 937, 958.
is_move: 1027, 1070, 1077, 1102.
is_prefix: 1178.

is_restricted: 3, 67, 68.
is_srch: 1027, 1064.
is_undoable: 1027, 1077.
is_unique: 1119.
is_wdvarassign: 104, 202, 446, 567.
is_wdvarname: 104, 201, 371, 439.
IS_WORDC: 803, 804, 805.
is_zerocount: 1027, 1077.
isa: 452, 513, 1299, 1302, 1305, 1306, 1307, 1308, 1309, 1325, 1326.
isalnum: 204, 792, 936.
isalpha: 204, 370.
isassign: 611.
isatty: 29, 30, 665, 1214, 1218, 1317.
isblank: 204.
ischild: 689, 691, 692.
iscmd: 806.
iscntrl: 204, 900, 962, 972.
isctype: 204, 621.
isdigit: 181, 204, 205, 261, 370, 925, 1068, 1069.
isfile: 614.
isgraph: 204.
ishere: 454.
ISIG: 869.
islower: 169, 181, 204, 959, 960, 1156.
ISMAGIC: 586, 607, 610, 611, 615, 616, 620, 622, 767, 768, 782, 807, 808.
isprint: 204.
ispunct: 204.
isrooted: 753, 756.
ISSET: 85, 113, 126, 128, 130, 131, 133, 137, 139, 145, 147, 148, 150, 154, 157, 163, 164, 171, 172, 173, 178, 190, 194, 369, 416, 532, 533, 535, 542, 544, 545, 547, 548, 550, 551, 552, 553, 571, 595, 598, 613, 794, 795, 1150, 1186, 1187, 1192, 1244, 1246, 1248, 1257, 1258, 1262, 1263, 1264, 1266, 1267.
issetugid: 16, 20.
issp: 1144.
isspace: 167, 168, 204, 324, 792, 793, 806, 1103, 1129, 1134, 1135, 1136, 1137, 1138, 1139, 1144, 1172.
isupper: 170, 181, 204, 959, 960, 1156, 1271.
isu8cont: 884, 932, 933, 942, 943, 962, 971, 991, 1011, 1012, 1013, 1023, 1051, 1054, 1056, 1058, 1061, 1076, 1077, 1079, 1083, 1085, 1089, 1106, 1107, 1122, 1125, 1126, 1130, 1145.
isxdigit: 204.
j: 467, 688, 689, 695, 696, 701, 703, 707, 708, 709, 716, 717, 718, 720, 721, 722, 726, 732, 737, 738, 739, 740, 780, 781, 802, 878, 931, 950, 966, 1208.
j_async: 147, 667, 725.
j_change: 20, 667, 676, 677, 682, 683, 688.
j_exit: 115, 535, 667, 688.
j_init: 63, 667, 673, 677, 683.
j_jobs: 667, 671, 739, 1269.
j_kill: 667, 707, 708, 1278.
j_lookup: 668, 696, 707, 717, 722, 739.
j_njobs: 390, 667, 738.
j_notify: 234, 667, 671, 688, 703, 740.
j_print: 668, 703, 709, 732, 739, 740.
j_resume: 667, 696, 1270.
j_set_async: 668, 694, 696, 726.
j_sigchld: 648, 668, 681, 695, 701.
j_startjob: 668, 694, 695, 715.
j_stopped_running: 235, 667, 737, 1222.
j_suspend: 667, 728, 1228.
j_systime: 468, 669, 709.
j_usrtime: 468, 669, 709.
j_wait: 468, 705.
j_waitj: 634, 668, 671, 694, 698, 700, 703, 709, 716, 717.
j_waitjed: 669.
jbuf: 107, 108, 229, 232, 236, 378, 509, 515, 532, 631, 1170.
JF_CHANGED: 671, 703, 732, 739, 740.
JF_FG: 671, 688, 690, 698, 699, 703, 709.
JF_KNOWN: 671, 698, 703, 725, 726.
JF_PIPEFAIL: 671, 690, 709.
JF_REMOVE: 671, 739, 740.
JF_SAVEDTTY: 671, 698, 699, 700, 711.
JF_SAVEDTTYPGRP: 671, 698, 700, 711.
JF_STARTED: 671, 695, 701, 703, 716, 726.
JF_USETTYMODE: 671, 690, 712.
JF_W_ASYNCNOTIFY: 671, 703, 709, 710.
JF_WAITING: 671, 703, 709, 710, 715.
JF_XXCOM: 463, 671, 690, 703.
JF_ZOMBIE: 671, 703, 721, 726, 727, 739.
jl: 726, 727.
JL_AMBIG: 722, 723, 724.
JL_INVALID: 722.
JL_NOSUCH: 717, 722.
JL_OK: 722.
job: 262, 509, 672, 690, 694, 696, 703, 705, 718, 722, 732, 734.
Job: 668, 669, 671, 672, 688, 689, 695, 696, 701, 703, 707, 708, 709, 716, 717, 718, 720, 721, 722, 725, 726, 732, 737, 738, 739, 740.
job_list: 666, 669, 688, 701, 702, 717, 718, 720, 721, 722, 723, 724, 727, 732, 737, 738, 739, 740.
jobchar: 732, 734.
jobs: 1269.
JP_LONG: 732, 735, 736, 739.

JP_MEDIUM: 703, 732, 736, 739, 740.
JP_NONE: 732.
JP_PGRP: 732, 739.
JP_SHORT: 709, 732, 733, 734, 735.
jstate: 703, 704.
JW_ASYNCNOTIFY: 709, 717.
JW_INTERRUPT: 709, 710, 717.
JW_NONE: 694, 700, 709, 716.
JW_STOPPEDWAIT: 709, 710.
k: 899, 902, 903, 906, 907, 914, 982.
kb: 907, 911, 912, 913.
kb_add: 891, 892, 893, 894, 895, 896, 905, 1017.
kb_add_string: 903, 905, 912, 913.
kb_decode: 900, 902, 903, 910.
kb_del: 906, 911, 912, 913.
kb_encode: 901, 907, 912.
kb_entry: 897, 898, 899, 902, 903, 905, 906, 907, 914, 982.
kb_find_hist_func: 982, 983.
kb_func: 888, 903, 905, 982, 983.
kb_list: 898.
kb_match: 899, 903, 982.
kb_print: 902, 909, 910.
kblist: 898, 899, 903, 906, 909, 910, 911, 912, 913, 921, 982, 1016.
KEEPASN: 87, 301, 529, 540.
keepasn_ok: 524, 528, 529.
KEOL: 920, 946, 948.
KEYWORD: 337, 367, 369, 426, 428, 429, 430, 431, 432, 436, 438, 440, 442, 443, 444, 450, 451, 452.
keywords: 87, 183, 184, 369, 416, 774, 1244.
ki: 1275, 1277.
kill: 115, 686, 695, 707, 708, 730, 859, 867, 869, 870, 1017, 1088, 1145, 1278.
kill: 1271.
kill_fmt_entry: 1201, 1275, 1277.
kill_info: 1275, 1276, 1277.
kill_job: 668, 688, 691, 707, 708.
killed: 688.
killpg: 688, 696, 707, 1278.
KILLSIZE: 887, 992, 993, 994.
killsp: 887, 992, 993, 994.
killstack: 887, 992, 993, 994.
killtp: 887, 993, 994.
KINTR: 920, 1018.
kmatch: 914, 920, 921, 922.
KSH: 615.
KSH_DEBUG: 12, 240, 243.
ksh_dup2: 243, 256, 258, 507, 509, 600, 630, 693.
ksh_func: 414, 450, 477, 542.
ksh_get_wd: 10, 745, 746, 1236.
ksh_getopt: 10, 37, 45, 467, 528, 844, 1206, 1207, 1212, 1213, 1214, 1220, 1221, 1222, 1223, 1225, 1230, 1236, 1239, 1243, 1250, 1252, 1260, 1265, 1269, 1270, 1273, 1279, 1280, 1281, 1285.
ksh_getopt_reset: 10, 35, 36, 43, 467, 528, 539, 1261, 1285.
KSH_IGNORE_RDONLY: 121, 127, 163, 1170.
KSH_RETURN_ERROR: 121, 127, 135, 140, 146, 171, 190, 191, 192, 193, 525, 535, 844, 1170, 1171, 1198, 1215, 1235, 1268, 1280, 1290, 1321.
KSH_SYSTEM_PROFILE: 65.
ksh_tmout: 159, 325, 657, 658.
ksh_tmout_state: 325, 645, 657, 658, 660.
KSH_UNWIND_ERROR: 121, 144, 516, 517, 571, 581, 1170, 1186, 1206.
ksh_version: 13, 14, 192, 401, 402.
kshbuiltins: 187, 1201, 1202.
kshdebug_dump: 12.
kshdebug_dump_: 12, 243.
kshdebug_init: 12, 250.
kshdebug_init_: 12, 243.
kshdebug_printf: 12.
kshdebug_printf_: 12, 243.
kshdebug_shf: 240.
ksheuid: 5, 6, 20, 25, 190, 194, 393, 1317, 1318.
kshname: 5, 6, 15, 24, 27, 28, 60, 305, 395, 532, 534.
kshpid: 5, 6, 20, 115, 147, 189, 685, 687, 688, 731, 737, 872.
KSTD: 920, 922, 923, 925, 928, 930, 932, 933, 934, 935, 940, 941, 942, 943, 944, 945, 950, 951, 952, 953, 958, 964, 965, 974, 975, 976, 977, 978, 979, 980, 983, 988, 991, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1007, 1019, 1020.
ktdelete: 80, 95, 151, 544.
ktenter: 80, 91, 116, 119, 148, 416, 540, 541, 550, 613, 1260.
ktinit: 80, 89, 112, 148, 186, 187, 416.
ktnext: 80, 96, 99, 553, 775, 1265.
ktsearch: 80, 90, 91, 94, 96, 117, 120, 134, 150, 151, 369, 536, 541, 546, 548, 1150, 1244, 1263, 1266.
ktsort: 80, 101, 1256, 1257, 1262.
ktwalk: 80, 96, 98, 553, 775, 1265.
l: 15, 75, 77, 78, 79, 112, 113, 116, 119, 133, 157, 541, 771, 801, 900, 901, 1206, 1224, 1250, 1285, 1290, 1291.
LAEXPR: 229, 1161, 1171.
last: 831, 844, 845, 848, 851.
last_ep: 1223.
last_info: 776.
last_job: 669, 690, 715, 716, 721.

last_line: 828.
last_match: 722, 723, 724.
last_proc: 672, 689, 690, 695, 704, 713, 722, 726, 732.
last_sb: 812, 823, 824, 839, 843.
lastac: 1022, 1075, 1077, 1092, 1118, 1119.
lastb: 1053, 1056, 1057.
lastcmd: 1022, 1075, 1077, 1092.
lastcol: 991.
lastp: 524, 525.
lastsearch: 1022, 1143.
`LBREAK`: 229, 515, 1223.
lbuf: 758, 761, 762.
`LCASEV`: 86, 117, 127, 128, 164, 165, 1252, 1254, 1259.
`LCONTIN`: 229, 515, 1223.
Leave: 427, 444, 524, 528, 530.
left: 97, 98, 99, 414, 417, 430, 431, 432, 437, 439, 441, 442, 450, 460, 468, 472, 473, 474, 475, 476, 477, 499, 500, 506, 507, 508, 509, 510, 511, 516, 517, 518, 519, 520, 535, 542, 660.
leftside: 1048, 1053, 1054.
len: 55, 91, 93, 122, 144, 193, 212, 218, 219, 221, 272, 289, 291, 293, 380, 382, 383, 503, 565, 581, 614, 621, 722, 724, 745, 747, 750, 752, 758, 760, 770, 790, 829, 837, 845, 873, 878, 879, 880, 881, 897, 899, 903, 914, 915, 921, 927, 951, 994, 1012, 1024, 1028, 1029, 1031, 1052, 1100, 1101, 1152, 1165, 1173, 1175, 1242.
length: 221.
lenp: 878, 879, 880.
`LERROR`: 229, 232, 236, 299, 301, 304, 463, 532.
`let`: 339, 1268.
let_cmd: 451.
`LETEXPR`: 337, 339, 451.
letnum: 178, 198, 199, 205, 376, 1134, 1135, 1136, 1173, 1174.
letter: 116, 119, 198, 199, 205, 376, 1172.
lex_state: 332.
Lex_state: 314, 332, 333, 334, 338, 350.
`LEXIT`: 229, 232, 235, 236, 463, 532, 645, 1222.
lflag: 844, 845, 846, 850, 851, 1271, 1273.
libc: 204.
limit: 966, 970, 1290, 1291.
Limits: 2, 1283, 1284, 1285, 1290, 1291.
limits: 1285, 1287, 1288, 1289.
line: 227, 305, 316, 327, 328, 368, 383, 405, 406, 434, 444, 450, 451, 816, 826, 828, 829, 835, 840, 853, 905, 914, 918, 919, 921, 922, 923, 947, 980, 982, 1140, 1153, 1215, 1241.
`LINE`: 7, 325, 326, 840, 900, 901, 905, 907, 914, 982, 1022, 1028, 1037.
line_co: 812, 824, 840, 843.
linelen: 1028, 1031, 1034, 1036, 1038, 1039, 1040, 1041, 1043, 1054, 1059, 1066, 1077, 1079, 1080, 1083, 1085, 1086, 1089, 1090, 1091, 1099, 1100, 1102, 1104, 1105, 1106, 1108, 1118, 1119, 1122, 1124, 1126, 1127, 1129, 1130, 1131, 1132, 1134, 1136, 1137, 1139, 1144, 1145, 1146, 1147, 1153, 1156, 1157.
lineno: 414, 444, 450, 451, 465, 499.
link: 71, 75, 77, 78, 79.
`LINTER`: 232.
`LINTR`: 229, 232, 236, 532, 634, 647.
list: 438, 907, 1279.
`LIT`: 1162, 1163, 1164, 1170, 1174, 1177.
`lit?TODO`: 1177.
`LJUST`: 86, 117, 127, 128, 164, 165, 166, 174, 1252, 1254, 1258, 1259.
`LJUSTIFY`: 86.
ll: 117.
`LLEAVE`: 229, 232, 236, 524, 532, 647, 693.
llen: 758, 761.
llnum: 289, 291, 292.
lnext: 77.
lno: 821.
loc: 59, 107, 111, 112, 113, 114, 116, 119, 120, 121, 133, 228, 229, 514, 534, 541, 595, 597, 774, 1206, 1212, 1224, 1256, 1257, 1281.
local: 104, 116, 121, 844, 881, 1198, 1250, 1251, 1255.
`local`: 61.
`LOCAL`: 86, 121, 525, 529, 535, 1255.
`LOCAL_COPY`: 86, 121, 529.
localtime: 385, 386, 396, 397, 398, 399.
`LOCK_EX`: 823.
`LOCK_UN`: 815, 824.
locpat: 1063, 1145.
log: 1173.
`LOGAND`: 366, 415, 419, 1329.
`login`: 61.
`LOGOR`: 366, 415, 419, 1329.
`LONG_MIN`: 1180.
longjmp: 1170.
lookup_msgs: 669, 696, 707, 717, 739.
lprev: 77.
lpv: 259.
`LRETURN`: 229, 232, 532, 1222.
`ls`: 61.
ls_info: 332, 333, 334, 339, 350.
ls_sasparen: 332, 346, 354.
ls_sbquote: 332, 350, 358.
ls_scsparen: 332, 346, 353, 354.
ls_sletparen: 332, 339, 360.

ls_state: 332, 333, 334, 339, 350, 354, 361, 362.
lseek: 274.
LSHELL: 229, 232, 532, 920, 1033, 1222.
lstat: 786, 798, 1317.
lstatb: 786, 787, 788.
LWORD: 338, 367, 369, 415, 427, 439, 440, 441, 442, 444, 449, 451, 454, 457, 558, 633, 797, 1329, 1330, 1332.
l1: 610.
l2: 77, 79, 133, 134, 610.
L2P: 75, 77.
l3: 610.
m: 51, 1197.
M_: 1026, 1027.
macro: 907, 1025, 1028, 1030, 1032, 1077, 1151, 1152, 1279.
macro_args: 887, 922, 1010.
macro_state: 1024, 1025.
magic: 807.
MAGIC: 561, 564, 565, 576, 580, 586, 588, 589, 590, 606, 611, 614, 621, 622, 767, 768, 782, 789, 807.
mail: 61.
mailcheck_interval: 1189, 1192, 1193.
main: 15, 59, 185, 262, 301, 867.
make: 61.
make_argv: 15, 59, 60.
make_magic: 565, 567, 568, 586.
make_path: 742, 747, 1234.
makenv: 104, 133, 521.
maketemp: 243, 272, 632, 854.
malloc: 69, 71, 75.
markdirs: 565.
markdirs: 769, 784.
match: 370, 1119, 1120, 1147.
match_len: 1119.
match_sep: 620.
matched: 619.
MAX_PREC: 1164, 1170, 1177, 1184.
max_width: 308, 309, 310.
maxhist: 837, 838.
maxncmp: 96.
MAXVICMD: 1022, 1063, 1077.
maybe_expand_tilde: 554, 593, 611.
mb_msg: 1191, 1194, 1197, 1198.
mb_mtime: 1191, 1192, 1194, 1197.
mb_next: 1191, 1192, 1195, 1196, 1197.
mb_path: 1191, 1192, 1194, 1196, 1197, 1198.
mballoc: 1189, 1195, 1197.
MBMESSAGE: 1198.
Mbox: 1191.
mbox: 1189, 1192, 1194.
mbox_t: 1189, 1191, 1192, 1195, 1196, 1197, 1198.
mbp: 1192, 1195, 1196, 1197, 1198.
mbset: 158, 160, 1190, 1194.
mc: 1053, 1057.
mcheck: 234, 1190, 1192.
mcset: 158, 1190, 1193.
MDPAREN: 366, 415, 451.
memchr: 283, 329.
memcmp: 870, 899, 921.
memcpy: 93, 132, 165, 282, 283, 287, 503, 575, 610, 745, 750, 751, 752, 760, 764, 768, 781, 790, 829, 1040, 1041, 1089, 1152, 1233.
memmove: 278, 329, 354, 783, 818, 822, 927, 931, 1028, 1038, 1039, 1043, 1077, 1089, 1090, 1091, 1099, 1100, 1110, 1146, 1147, 1195.
memset: 185, 367, 368, 466, 535, 567, 730, 1030, 1031, 1047.
mess: 634, 639, 733, 1275, 1277.
mess_width: 1275.
meta: 1303, 1304, 1329, 1333.
mflagset: 677.
MIN_COLS: 157, 860, 872.
MIN_EDIT_SPACE: 860, 1031.
mkstemp: 272.
mlastchkd: 1189, 1192.
mlen: 370.
mlist: 1196.
mmsg: 1195.
mode: 245, 261, 269, 763, 764, 765.
mode_t: 1207.
modified: 1022, 1029, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1102, 1104, 1106, 1107, 1108, 1109, 1118, 1119, 1140, 1141, 1142, 1143, 1144, 1153, 1155, 1156.
more: 217, 219, 223.
morec: 1031, 1037, 1047, 1057.
moreright: 1053, 1054, 1055, 1057.
mpath: 1195.
mplist: 1189, 1192, 1195.
mprintit: 1189, 1192, 1198.
mpset: 158, 160, 1190, 1195.
mptoparse: 1195.
msg: 1324, 1332, 1336.
Mtoken: 2, 1158, 1169, 1170, 1172, 1177, 1179, 1184.
MUL_NO_OVERFLOW: 76.
multi: 420.
munset: 1189, 1195, 1196.
musthave: 412, 430, 431, 436, 438, 439, 441, 442, 443, 448, 449, 450, 451, 454, 459.
mv: 61.
mval: 1195.

n: 50, 55, 83, 90, 91, 96, 102, 116, 119, 144, 171, 174, 213, 214, 215, 226, 277, 286, 287, 294, 295, 308, 380, 488, 489, 595, 641, 791, 818, 831, 834, 856, 874, 880, 925, 1145, 1146, 1206, 1222, 1223, 1242, 1271, 1338, 1342.
name: 3, 47, 50, 53, 55, 56, 57, 68, 84, 90, 91, 93, 94, 96, 100, 113, 132, 133, 134, 136, 139, 145, 148, 150, 151, 154, 155, 160, 163, 164, 187, 204, 228, 230, 245, 261, 271, 272, 273, 364, 365, 415, 416, 450, 454, 458, 478, 501, 502, 540, 541, 542, 543, 545, 546, 547, 548, 549, 550, 551, 571, 579, 600, 613, 621, 623, 631, 632, 634, 639, 641, 643, 763, 764, 775, 790, 791, 794, 819, 844, 854, 856, 868, 1162, 1165, 1175, 1186, 1187, 1188, 1199, 1221, 1225, 1256, 1258, 1259, 1262, 1263, 1274, 1275, 1277, 1284, 1287, 1288.
name_width: 1275, 1276, 1277.
namelen: 132, 136, 137, 139, 763, 764.
names: 148.
name1: 100.
name2: 100.
nargv: 60.
nbuf: 276, 380, 383, 1149, 1152.
nbytes: 279, 287, 329.
nc: 931, 936, 937.
NCCLASSES: 204.
nchars: 992.
ncmp: 96.
ncol: 1053, 1054.
ncopy: 282, 283, 287.
ncursor: 1077, 1078, 1103, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1134, 1135, 1136, 1137, 1138, 1139.
ndel: 991.
need_nl: 471.
neednl: 1045, 1047.
neg: 178, 489, 491.
NELEM: 7, 43, 44, 50, 51, 53, 55, 57, 58, 682, 684, 685, 693, 868, 904, 907, 908.
nentries: 96.
nest: 504, 620, 807, 810.
nested: 412, 427, 436.
nesting: 412, 424, 434, 435, 456.
nesting_pop: 412, 430, 435, 436, 437, 439, 441.
nesting_push: 412, 430, 434, 436, 437, 439, 441.
nesting_state: 412, 426, 433, 434, 435, 436.
new: 136, 137, 138, 139, 193, 333, 610, 1022, 1040, 1041.
new_es: 1145.
new_job: 668, 690, 718.
new_proc: 668, 690, 719.
new_umask: 1207, 1209, 1210, 1211.
new_val: 1210, 1211.
newblock: 104, 112, 185, 529.
newcol: 1023, 1049, 1050, 1051, 1058.
newenv: 4, 111, 228, 231, 378, 463, 631, 632, 1170.
newj: 718.
newlines: 565, 570, 585.
newsize: 781, 783.
newst: 574.
newtp: 412, 414, 417, 423, 430, 431, 432, 437, 439, 441, 442, 444, 450, 451, 452, 460.
newval: 20, 21, 1260, 1264.
next: 71, 75, 77, 78, 79, 97, 98, 99, 105, 112, 113, 117, 120, 133, 134, 234, 271, 272, 273, 313, 316, 319, 323, 324, 335, 368, 369, 370, 371, 421, 541, 565, 573, 574, 670, 672, 688, 690, 694, 695, 697, 701, 702, 704, 708, 714, 717, 718, 719, 720, 721, 722, 723, 724, 727, 732, 735, 736, 737, 738, 739, 740, 774, 1256, 1257, 1281.
nextstate: 1023, 1064, 1066, 1068, 1070.
nfd: 246, 256, 257.
nflag: 739, 844, 845, 846, 853, 1269.
nflags: 1238.
nfmt: 853.
nfree: 88, 89, 91, 92, 94, 96.
nice: 693.
nj: 738.
njobs: 262, 265, 509, 669, 690, 694, 705, 1215.
nleft: 933, 943.
nlen: 165, 166, 167, 168, 1006, 1149, 1152, 1233.
nmemb: 76.
no: 807.
no_ro_check: 163.
noassign: 1169, 1170, 1173, 1179, 1183, 1184.
Nodir: 790.
nohup: 22, 61.
NONE: 1089, 1090, 1091, 1092, 1099, 1111, 1112, 1118, 1119.
not: 622, 1314, 1316.
now: 154, 1192.
np: 785, 789, 790.
nparen: 332, 339, 346, 353, 354, 360.
npath: 545, 549, 551, 552.
nread: 326.
ns: 213.
NSIG: 635, 636, 637, 639, 641, 645, 650, 651, 652, 653, 1221, 1274, 1275.
nsize: 94.
nspace: 308, 310.
ntblp: 94.
ntowrite: 277, 278.
ntries: 914, 919, 923, 924.
ntruncate: 379, 380, 381, 382, 383.

NUFILE: 7, 114, 195, 426, 445, 453, 466, 1227.
null: 65, 112, 126, 161, 162, 167, 173, 178, 189, 191, 232, 319, 321, 322, 335, 368, 547, 549, 556, 559, 594, 595, 597, 598, 612, 669, 745, 747, 774, 815, 1206, 1212, 1213, 1220, 1221, 1222, 1223, 1231, 1232, 1235, 1270, 1274, 1281, 1330, 1334, 1337.
num: 119, 146, 172, 178, 180, 181, 182.
num_width: 1275, 1276, 1277, 1338, 1340, 1341.
numbuf: 289, 291.
nump: 178.
nwidth: 1338, 1339, 1342.
nwords: 770, 771, 776, 780, 791, 795, 796, 798, 799, 801, 802, 811, 1006, 1007, 1059, 1118, 1119, 1120.
nwritten: 289, 293.
nzombie: 669, 693, 703, 721, 726, 727.
nzombies: 721, 727.
n0: 1175.
o: 37, 331.
O_ACCMODE: 245, 248, 261.
O_APPEND: 623.
O_ASN: 1163, 1179, 1185.
O_BAND: 1163, 1183.
O_BANDASN: 1163, 1183.
O_BNOT: 1163, 1177.
O_BOR: 1163, 1183.
O_BORASN: 1163, 1183.
O_BXOR: 1163, 1183.
O_BXORASN: 1163, 1183.
O_COMMA: 1163, 1185.
O_CREAT: 623, 624, 627, 839.
O_DIV: 1163, 1179, 1180.
O_DIVASN: 1163, 1179, 1180.
O_EQ: 1163, 1182.
O_EXCL: 624.
O_EXLOCK: 839.
O_GE: 1163, 1182.
O_GT: 1163, 1182.
O_LAND: 1163, 1179, 1183.
O_LE: 1163, 1182.
O_LNOT: 1163, 1177.
O_LOR: 1163, 1179, 1183.
O_LSHIFT: 1163, 1181.
O_LSHIFTASN: 1163, 1181.
O_LT: 1163, 1182.
O_MINUS: 1163, 1177, 1180.
O_MINUSASN: 1163, 1180.
O_MINUSMINUS: 1163, 1177.
O_MOD: 1163, 1179, 1180.
O_MODASN: 1163, 1179, 1180.
O_NE: 1163, 1182.
O_NONBLOCK: 280.
O_PLUS: 1163, 1177, 1180.
O_PLUSASN: 1163, 1180.
O_PLUSPLUS: 1163, 1177, 1178.
O_RDONLY: 28, 228, 245, 248, 261, 600, 623, 632, 693, 856.
O_RDWR: 248, 261, 623, 665, 839.
O_RSHIFT: 1163, 1181.
O_RSHIFTASN: 1163, 1181.
O_TERN: 1163, 1179, 1184.
O_TIMES: 1163, 1180.
O_TIMESASN: 1163, 1180.
O_TRUNC: 624.
O_WRONLY: 245, 248, 261, 623, 624.
OBRACE: 208, 586, 590, 606, 607, 610.
OBRACK: 208.
odirsep: 785.
oenv: 107, 111, 114, 195, 196, 706, 1222, 1223.
oerrno: 279.
oexstat: 646.
OF_ANY: 47, 48.
OF_CMDLINE: 20, 26, 43, 44, 47, 48.
OF_INTERNAL: 47, 48.
OF_SET: 20, 43, 44, 47, 58, 1224.
OF_SPECIAL: 23, 47, 156, 868.
ofd: 256, 258.
ofd1: 600.
off: 918, 1012.
off_t: 274.
offset: 818, 983, 984, 985, 986, 1324, 1332, 1336.
oflags: 245.
ohnum: 1022, 1029, 1143, 1146, 1155.
oi: 55, 56.
ok: 127, 130.
old: 1022, 1040, 1041, 1042.
old_argc: 228, 229.
old_argv: 228, 229.
old_base: 334.
old_beg: 219.
old_changed: 646.
old_end: 333, 334.
old_func_parse: 450.
old_inuse: 532.
old_kshname: 532, 534.
old_nesting: 426, 430, 436, 437, 439, 441.
old_source: 231, 232.
old_umask: 1207, 1208, 1210, 1211.
old_xflag: 532.
oldchars: 869, 870.
oldest: 726, 727.
oldf: 710.
oldmagic1: 815, 820.

oldmagic2: 815, 820.
oldpwd_s: 1230, 1232, 1235.
oldsize: 769, 781, 783, 784, 951, 981.
oldval: 20.
olen: 165, 166, 167, 1006, 1149, 1152, 1233.
omask: 509, 689, 691, 693, 696, 698, 699, 707, 715, 716, 717, 725, 738, 739, 740.
ONEWORD: 337, 339, 367, 558, 633, 797.
onoff: 869.
op: 132, 414, 460, 1163, 1176, 1177, 1178, 1179, 1188, 1210, 1211, 1308, 1310, 1311, 1312, 1313, 1314, 1321, 1324, 1326, 1330, 1331, 1334, 1335.
Op: 2, 84, 132, 231, 331, 412, 413, 414, 417, 418, 419, 420, 424, 426, 431, 432, 436, 438, 442, 443, 450, 460, 461, 462, 463, 467, 468, 469, 470, 471, 495, 499, 500, 523, 524, 542, 600, 667, 689, 1203.
op_num: 1296, 1304.
op_text: 1296, 1304.
OPAREN: 208.
OPAT: 336, 341, 357, 363, 486, 504, 505, 568.
open: 245, 627, 632, 665, 693, 839.
OPEN_PAREN: 1163, 1177.
opendir: 789, 790.
openpipe: 243, 259, 507, 509, 600.
operation: 825.
Opinfo: 1165, 1167.
opinfo: 1162, 1163, 1167, 1175, 1176, 1188.
opipe: 1242.
opnd1: 1308, 1311, 1312, 1313, 1314, 1315, 1316, 1317, 1319, 1320, 1321, 1322, 1323, 1326, 1331, 1335.
opnd2: 1308, 1312, 1313, 1314, 1319, 1320, 1321, 1322, 1323, 1326, 1331, 1335.
opt: 467.
opt_width: 53, 55, 56.
optarg: 32, 35, 39, 40, 41, 42, 45, 46, 467, 845, 1214, 1239, 1252, 1273, 1280, 1288.
optc: 43, 45, 58, 467, 528, 844, 845, 1207, 1214, 1225, 1230, 1236, 1239, 1243, 1250, 1252, 1260, 1265, 1269, 1273, 1279, 1280, 1285, 1288.
optind: 32, 35, 36, 37, 38, 40, 41, 42, 43, 45, 467, 528, 844, 1206, 1207, 1212, 1213, 1214, 1220, 1221, 1222, 1223, 1225, 1230, 1236, 1239, 1243, 1250, 1253, 1254, 1255, 1260, 1265, 1269, 1270, 1273, 1279, 1280, 1281, 1282, 1285.
option: 10, 46, 47, 48, 49, 50, 1284, 1288, 1290, 1316.
options: 37, 39, 40, 1239, 1243, 1250, 1251, 1252, 1280, 1281, 1285.
options_fmt_entry: 52, 55, 56.
options_info: 53, 55, 56.
opts: 43, 45, 53, 55, 56.
OQUOTE: 336, 339, 342, 361, 480, 484, 504, 505, 561, 568.
orig_bsize: 282.
orig_buf: 283.
orig_nbytes: 287.
orig_p: 622.
osa: 728, 730, 731.
osize: 94.
osource: 631, 633.
OSUBST: 336, 347, 348, 349, 485, 504, 505, 562, 568, 572.
otab: 1304.
otblp: 94.
our_pgrp: 669, 685, 686, 687, 699, 711.
outofwin: 1023, 1048, 1049.
output: 732, 735.
outtree: 412, 423, 424.
oval: 1178.
owp: 1224, 1326.
p: 32, 43, 44, 68, 90, 91, 94, 95, 96, 99, 101, 142, 144, 165, 194, 200, 201, 202, 212, 214, 215, 306, 327, 328, 329, 369, 371, 372, 376, 377, 380, 416, 418, 419, 420, 450, 489, 501, 502, 571, 586, 587, 593, 594, 601, 606, 611, 614, 615, 620, 622, 640, 641, 642, 643, 645, 646, 648, 650, 651, 652, 653, 654, 689, 695, 696, 701, 703, 708, 714, 719, 721, 722, 732, 745, 758, 763, 779, 781, 802, 803, 807, 826, 840, 844, 914, 974, 983, 1024, 1031, 1144, 1149, 1156, 1194, 1195, 1197, 1221, 1236, 1250, 1262, 1271, 1333.
P_ADD: 1164, 1167.
P_ASSIGN: 1164, 1167, 1179.
P_BAND: 1164, 1167.
P_BOR: 1164, 1167.
P_BXOR: 1164, 1167.
P_COMMA: 1164, 1167.
P_EQUALITY: 1164, 1167.
P_LAND: 1164, 1167.
P_LOR: 1164, 1167.
P_MULT: 1164, 1167.
P_PRIMARY: 1164, 1167, 1176, 1177.
P_RELATION: 1164, 1167.
P_SHIFT: 1164, 1167.
P_TERN: 1164, 1167, 1184.
p_ts: 311, 468, 1200.
p_tv: 312, 468, 1200, 1226.
parent: 1228.
parse_args: 10, 26, 43, 48, 301, 1224.
parse_state: 375.
passwd: 613.

pat: 601, 602, 603, 604, 605, 766, 771, 772, 774, 775, 781, 829, 846, 847, 849, 983, 984, 985, 986, 987, 1147.
pat_len: 829.
pat_scan: 10, 617, 618, 619, 620.
path: 272, 745, 753, 754, 757, 758, 763, 764, 765, 766, 781, 1318.
PATH_MAX: 746, 758, 1234.
path_order: 776, 777, 778.
path_order_cmp: 766, 776, 778.
path_order_info: 776, 777, 778.
pathlen: 781, 782, 783.
patlen: 781.
pattern: 621.
pc: 615.
pe: 614, 615, 616, 617, 618, 619, 620, 807.
pend: 749.
PERF_DEBUG: 96.
perror: 16.
PEXITED: 670, 701, 704, 705, 709, 712, 714, 733, 739, 740.
pflag: 1243, 1244, 1250, 1252, 1258, 1260, 1262, 1263.
pgrp: 672, 688, 690, 692, 694, 696, 698, 707, 709, 722, 732.
phys_path: 1230, 1234.
phys_pathp: 747.
physical: 1230, 1234, 1235, 1236.
pid: 271, 272, 273, 670, 690, 691, 692, 694, 701, 702, 708, 722, 726, 732, 735, 736.
pid_t: 5, 6, 15, 667, 669, 670, 672, 686, 725, 1228.
pioact: 469, 471, 478, 497.
pipe: 259.
pipeline: 412, 418, 419, 428, 429.
PJ_ON_FRONT: 703, 720.
PJ_PAST_STOPPED: 690, 696, 720.
plain_fmt_entry: 461, 1342, 1343.
pledge: 16, 20.
plen: 747, 749, 751.
plist: 747, 749, 751.
pnext: 615, 617, 618, 619.
PO_COPROC: 1237, 1242.
PO_EXPAND: 1237, 1238, 1239, 1240.
PO_HIST: 1237, 1239.
PO_NL: 1237, 1238, 1239, 1240.
PO_PMINUSMINUS: 1237, 1239.
POP_STATE: 334, 351, 352, 353, 354, 355, 356, 357, 358, 363.
*pop_state_*: 314, 334.
popblock: 104, 113, 114.
pos: 452, 513, 770, 803, 1299, 1303, 1308, 1310, 1324, 1325, 1326, 1329, 1330, 1332, 1333, 1334.
positions: 1210, 1211.
POSIX: 23, 28, 31, 280.
posix: 311, 312.
posix_cclass: 621, 622.
POSIXLY_CORRECT: 11, 23.
pp: 90, 91, 96, 622, 1338, 1339, 1342.
ppid: 15, 192, 672, 688, 690, 701, 717, 721, 737.
pprompt: 313, 327, 379, 917, 968, 1060, 1218.
pr: 61.
pr_list: 462, 799, 801, 1342.
pr_menu: 462, 1337, 1338, 1342.
Prec: 2, 1158, 1164, 1165, 1176.
prec: 1165, 1176, 1179, 1183.
precision: 289, 290, 291, 293.
prefcol: 10, 308, 309.
prefix: 311, 312, 1260, 1262, 1263.
prefix_len: 790, 799, 801, 802.
prest: 615, 617, 618, 619.
prev: 71, 75, 77, 78, 136, 137, 138, 573, 574, 579, 580, 581, 720, 721, 869.
PRIi64: 144, 1291.
print: 1237.
PRINT: 1111, 1119.
print_columns: 10, 55, 308, 1275, 1338, 1342.
print_expansions: 1023, 1059, 1095, 1113, 1119.
print_menu: 1337.
print_ulimit: 1283, 1287, 1288, 1289, 1291.
print_value_quoted: 10, 306, 1221, 1245, 1258, 1262, 1263.
printf: 243, 297, 298, 313.
printoptions: 46, 52, 54.
printpath: 1230, 1232, 1233, 1235.
proc: 670.
Proc: 668, 669, 670, 672, 689, 695, 696, 701, 703, 708, 714, 719, 721, 722, 732.
proc_list: 672, 690, 694, 695, 697, 702, 704, 708, 714, 721, 723, 724, 732, 734.
procpid: 5, 6, 20, 189, 272, 273, 688, 690, 691, 695, 701, 717, 721, 737, 872.
prompt: 81, 82, 327, 378, 916, 917, 968, 1031, 1060, 1218.
prompt_redraw: 887, 916, 917, 968.
prompt_skip: 887, 916, 968, 1031, 1037, 1060.
prompt_trunc: 1031, 1037, 1060.
promptlen: 314, 379, 863, 916, 968, 1031.
PRUNNING: 670, 688, 690, 696, 697, 698, 704, 710, 717, 733, 737.
PS_END: 375, 376.
PS_IDENT: 375, 376.
PS_INITIAL: 375, 376.
PS_NUMBER: 375, 376.
PS_SAW_HASH: 375, 376.

PS_VAR1: 375, 376.
PSIGNALLED: 670, 701, 704, 705, 709, 713, 714, 733, 739, 740.
PSTOPPED: 670, 688, 697, 701, 703, 707, 709, 710, 711, 712, 720, 733, 737.
psub: 615, 617, 618, 619.
ps1: 378.
PS1: 234, 328, 378.
PS2: 325, 378, 1218.
PS4_SUBSTITUTE: 463, 465, 529, 623.
ptest_error: 1292, 1324, 1325.
ptest_eval: 1292, 1313, 1325.
ptest_getopnd: 1292, 1310, 1325.
ptest_isa: 1292, 1303, 1325.
ptmp: 1235.
ptns: 442.
ptr: 76, 77, 78.
ptree: 469, 471, 495.
push: 931.
PUSH_STATE: 333, 341, 342, 346, 347, 350, 357, 361, 363.
*push_state_*: 314, 333.
pushs: 27, 28, 29, 227, 230, 313, 319, 368, 369, 370, 371, 558, 600, 633, 797, 1220.
put_job: 668, 690, 696, 703, 720.
putbuf: 1023, 1066, 1076, 1083, 1100, 1101, 1108, 1109, 1118, 1119, 1144.
putbuf_func: 881.
pv: 259, 260, 463, 507, 509, 600.
pw: 613.
pw_dir: 613.
pwd: 1236.
pwd: 191, 1230, 1235.
PWD: 191.
pwd_s: 1230, 1235.
pwd_v: 191.
pwdx: 191.
pwidth: 1031, 1037, 1047, 1054, 1056, 1057, 1058.
p1: 100, 211.
p2: 100, 211.
P2L: 75, 77, 78.
q: 44, 165, 329, 501, 758, 826, 987, 1333.
QCHAR: 336, 342, 344, 351, 361, 482, 504, 505, 561, 568, 569.
qsort: 102, 776.
qsortp: 10, 43, 101, 102, 769, 780.
quit: 859, 867, 869, 870, 1017, 1028, 1223.
quitenv: 4, 107, 114, 229, 230, 231, 232, 236, 378, 463, 466, 509, 515, 532, 631, 1170, 1171, 1222.
quote: 561, 565, 568, 569, 572, 573, 574, 576, 579, 580, 581, 584, 586.
quoted: 480, 482, 484.
r: 167, 308, 499, 611.
R_OK: 261, 269, 547, 549, 626, 763, 1212, 1214, 1236, 1317.
rand: 154, 203.
rbsize: 237, 247, 249, 251, 276, 281, 285.
rcp: 868, 961, 988, 989, 990.
RDONLY: 86, 117, 118, 125, 127, 135, 140, 145, 146, 163, 565, 575, 1188, 1215, 1225, 1251, 1252, 1259, 1280.
read: 245, 262, 265, 266, 267, 269, 270, 279, 509, 705.
read: 1214, 1337.
read_args: 1337.
readdir: 790.
readhere: 314, 372, 373.
readlink: 761.
readonly: 567.
readonly: 1250.
readw: 262, 265, 267, 270, 509.
realloc: 77.
reallocarray: 814, 818.
really_exit: 231, 233, 235, 674, 675, 1222.
rec: 838.
reclaim: 3, 114, 115, 197, 233.
redir: 455, 457, 458.
REDIR: 365, 415, 427, 444, 454, 457, 1329.
redo_insert: 1023, 1075, 1076, 1092.
redraw_line: 1023, 1045, 1046, 1047, 1059, 1097, 1119.
refresh_line: 1023, 1048, 1065, 1066, 1067, 1074, 1075, 1118, 1119, 1143, 1145.
REG_BI: 87, 540, 545.
reject: 338, 412, 418, 419, 420, 423, 426, 432, 440, 442, 444, 448, 450, 451, 452, 454, 455, 1329, 1330, 1332.
remove_job: 668, 691, 693, 703, 705, 709, 721, 726, 727, 739, 740.
remove_temps: 3, 196, 197, 273.
rep: 829, 846, 847, 849.
rep_len: 829.
repl: 1100.
replace: 370.
REPLACE: 1063, 1076, 1084, 1089, 1099.
res: 1176, 1179, 1180, 1181, 1182, 1183, 1185, 1302, 1305, 1306, 1308, 1309, 1314, 1316, 1317, 1318, 1325, 1326.
reserved: 415, 416.
reset: 140.
reset_nonblock: 10, 31, 279, 280.
resource: 1284, 1290, 1291.
restfd: 114, 243, 258, 507, 600, 629.
restore: 643.

restore_cbuf: 1023, 1039, 1143, 1145, 1146, 1147.
restore_dfl: 654, 655.
restore_edstate: 1022, 1041, 1118, 1119.
restore_pipe: 638, 655, 1242.
restore_ttypgrp: 669, 687, 688, 729, 731.
restoresigs: 521, 638, 653.
restr_com: 67.
restricted: 15, 64, 67.
ret: 157, 252, 253, 256, 275, 279, 746, 765, 826, 844, 849, 855, 856, 914, 920, 922, 923, 951, 1157, 1168, 1243, 1244, 1248, 1280, 1303, 1329, 1333.
`return`: 1222.
reuse: 270.
rewind: 840, 843.
rewindow: 1023, 1048, 1050.
rflag: 844, 845, 846, 852, 853, 854, 1260.
right: 414, 417, 418, 420, 430, 431, 432, 437, 443, 460, 472, 474, 475, 476, 499, 500, 507, 508, 511, 512, 518, 519, 520.
`RJUST`: 86, 117, 127, 128, 164, 165, 166, 174, 1252, 1254, 1258, 1259.
`RJUSTIFY`: 86.
rlim_cur: 1290, 1291.
`RLIM_INFINITY`: 1290, 1291.
rlim_max: 1290, 1291.
rlim_t: 1290, 1291.
rlimit: 1290, 1291.
`RLIMIT_CORE`: 1286.
`RLIMIT_CPU`: 1286.
`RLIMIT_DATA`: 1286.
`RLIMIT_FSIZE`: 1286.
`RLIMIT_MEMLOCK`: 1286.
`RLIMIT_NOFILE`: 1286.
`RLIMIT_NPROC`: 1286.
`RLIMIT_RSS`: 1286.
`RLIMIT_STACK`: 1286.
`rm`: 61.
rnleft: 237, 244, 245, 247, 249, 251, 253, 274, 281, 282, 283, 284, 285.
row: 872.
rows: 308, 309.
rp: 237, 244, 245, 247, 249, 251, 253, 274, 276, 281, 282, 283, 284, 285.
rsize: 997.
ru_stime: 468, 701, 1226.
ru_utime: 468, 701, 1226.
running: 696, 697.
runtrap: 236, 634, 638, 645, 646, 648.
runtraps: 234, 236, 327, 463, 638, 645, 649, 710, 874.
rusage: 468, 701, 1226.
`RUSAGE_CHILDREN`: 468, 701, 1226.
`RUSAGE_SELF`: 468, 1226.
ru0: 468, 701.
ru1: 468, 701.
rv: 216, 463, 466, 468, 506, 507, 508, 510, 511, 512, 513, 515, 516, 517, 518, 519, 520, 524, 528, 530, 531, 532, 533, 535, 539, 622, 689, 694, 696, 700, 707, 709, 710, 714, 716, 717, 1213, 1220, 1260, 1263, 1265, 1266, 1268, 1269, 1270, 1271, 1278, 1279.
rval: 144, 621, 708, 747, 751, 881, 1118, 1119, 1168, 1230, 1234, 1255, 1290.
rw: 499.
s: 15, 84, 127, 143, 155, 163, 165, 173, 178, 198, 199, 201, 202, 210, 212, 213, 227, 228, 231, 238, 254, 288, 289, 306, 319, 321, 325, 350, 354, 368, 369, 370, 371, 421, 424, 447, 455, 463, 488, 558, 579, 600, 614, 615, 631, 648, 768, 773, 779, 782, 791, 796, 815, 829, 840, 853, 876, 881, 899, 900, 901, 926, 1014, 1101, 1161, 1220, 1221, 1237, 1258, 1304, 1334, 1337.
`S_`: 1026, 1027.
s_dot: 191.
`S_ISBLK`: 1317.
`S_ISCHR`: 31, 1317.
`S_ISDIR`: 156, 765, 787, 788, 1317, 1318.
`S_ISFIFO`: 1317.
`S_ISGID`: 1317.
`S_ISLNK`: 787, 788, 1317.
`S_ISREG`: 255, 624, 765, 1192, 1194, 1197, 1317.
`S_ISSOCK`: 1317.
`S_ISUID`: 1317.
`S_ISVTX`: 1317.
`S_IXGRP`: 765, 1318.
`S_IXOTH`: 765, 1318.
`S_IXUSR`: 765, 1318.
s_pwd: 191.
s_stdin: 31.
sa: 728, 730.
sa_flags: 639, 642.
sa_handler: 639, 642, 644, 730.
sa_mask: 639, 642, 730.
safe_prompt: 5, 6, 190, 378.
`SALIAS`: 316, 317, 324, 335, 369, 421.
sameline: 986.
`SASPAREN`: 330, 339, 340, 346.
sasparen_info: 332.
save: 434, 1146, 1147, 1329.
save_cbuf: 1023, 1038, 1066, 1146, 1147.
save_disable_subst: 1170, 1171.
save_edstate: 1022, 1040, 1118, 1119.
save_es: 1145.
save_xerrok: 463, 516.

saved: 435.
saved_atemp: 378.
saved_inslen: 1022, 1029, 1088, 1092.
saved_ttypgrp: 671, 672, 698, 711.
savef: 1220.
savefd: 107, 111, 114, 195, 243, 257, 258, 259, 466, 507, 509, 600, 623, 629, 706, 839, 1227.
savehist: 1214, 1215, 1216.
savepos: 758, 760, 762.
saves: 1220.
saw_eq: 565, 567, 587, 591.
saw_glob: 807, 808, 810.
saw_nl: 879.
saw_p: 528.
saw_slash: 773.
sb: 823, 839.
SBASE: 330, 338, 339, 340, 341.
Sbase1: 340, 355, 357, 363.
Sbase2: 340, 356, 360.
SBQUOTE: 330, 340, 350.
sbquote_info: 332.
SBRACE: 330, 340, 347, 355.
SBRACEQ: 330, 340, 342, 347, 351, 356.
sc: 615.
scriptexec: 461, 521, 523.
SCSPAREN: 330, 332, 340, 346, 353, 354.
scsparen_info: 332.
sc1: 1304.
SDQUOTE: 330, 332, 340, 342, 347, 350, 361.
se: 614, 615, 616, 617, 618, 619, 779, 785, 789.
Search: 91, 92, 545.
search: 462, 523, 545, 547, 549, 763, 1212, 1264.
search_access: 462, 763, 764, 765, 783.
search_path: 81, 82, 156, 160, 523, 549, 774, 1212, 1264.
seconds: 154, 159.
sed: 61.
SEEK_CUR: 274.
SEEK_END: 824.
SELECT: 415, 439.
select: 414, 415, 439, 506, 514, 517, 1337.
select_fmt_entry: 461, 1338, 1341.
select_menu_info: 1338, 1340, 1341.
SEOF: 232, 317, 321, 322.
seq: 897, 899, 902, 903, 910, 911, 912, 913, 921, 982.
set: 1224, 1250.
set: 45, 46, 58, 121, 122, 123, 125, 127, 128, 567, 634, 640, 645, 646, 650, 651, 652.
set_array: 43, 104, 140.
set_current_wd: 191, 742, 745, 1235.
set_editmode: 157, 864, 868.
set_opts: 43, 44.
set_prompt: 234, 313, 325, 378, 379, 1218.
set_ulimit: 1283, 1288, 1289, 1290.
setargs: 1224.
setargsp: 43, 45.
setctypes: 10, 156, 160, 209, 210.
setexecsig: 535, 638, 642, 643.
setgroups: 20.
sethistcontrol: 157, 160, 813, 817.
sethistfile: 157, 813, 819.
sethistsize: 157, 813, 818.
setint: 104, 154, 171, 192, 872, 1178, 1179.
setint_v: 104, 171, 172, 1170, 1174, 1178, 1179, 1186.
setpgid: 687, 688, 692, 701, 729, 731.
setresgid: 20.
setresuid: 20.
setrlimit: 1290.
setsig: 115, 638, 639, 640, 642, 648, 653, 654, 655, 659, 681, 682, 684, 685, 686, 693, 867.
setspec: 103, 113, 155, 163, 171, 172.
setstr: 104, 121, 130, 135, 140, 146, 163, 171, 190, 191, 192, 193, 516, 517, 525, 535, 581, 844, 1170, 1198, 1215, 1235, 1280.
settrap: 638, 646, 648, 652, 1221.
setupterm: 157.
SF_ALIAS: 318, 324, 339.
SF_ALIASEND: 318, 324, 421.
SF_ECHO: 234, 318, 321.
SF_TTY: 30, 231, 232, 233, 318, 325, 340.
SFILE: 28, 230, 317, 322, 328, 368.
sflag: 844, 845.
sflags: 245, 247, 248, 249, 251.
sh: 61.
sh_flag: 10, 17, 20, 46, 48, 58, 868.
sh_options: 17, 43, 44, 46, 48, 49, 50, 51, 53, 55, 57, 58, 868.
shbuiltins: 187, 1200, 1202.
shcomexec: 62, 67, 462, 536.
shell: 4, 15, 227, 228, 229, 230, 231, 236, 463, 523, 1220.
shell_flags: 18, 19.
shellf: 15, 96, 232, 235, 243, 297, 623, 737, 826, 1015, 1198, 1214, 1337.
SHEREDELIM: 330, 338, 339, 340, 361, 362.
SHEREDQUOTE: 330, 340, 361.
shf: 28, 29, 30, 114, 115, 228, 229, 230, 244, 245, 246, 247, 249, 251, 252, 253, 254, 271, 272, 274, 275, 276, 277, 278, 281, 282, 283, 284, 285, 286, 287, 288, 289, 293, 294, 295, 296, 308, 311, 312, 316, 327, 328, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487,

488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 505, 566, 585, 600, 631, 632, 633, 732, 734, 735, 736, 844, 854, 856, 1214, 1215, 1217.
Shf: 2, 4, 10, 114, 228, 237, 238, 239, 240, 242, 245, 247, 249, 251, 252, 253, 254, 271, 274, 275, 276, 281, 282, 283, 284, 285, 286, 287, 288, 289, 294, 295, 296, 308, 311, 312, 316, 469, 470, 471, 478, 479, 480, 487, 488, 489, 505, 566, 600, 631, 668, 669, 682, 732, 839, 844, 1200, 1214.
SHF_ACCMODE: 245.
SHF_ALLOCB: 245, 247, 249, 251, 252, 276.
SHF_ALLOCS: 245, 247, 249, 251, 252, 254.
SHF_BSIZE: 237, 245, 247, 249.
shf_clearerr: 244, 327, 1217.
SHF_CLEXEC: 28, 228, 245, 247, 249, 600.
shf_close: 114, 115, 238, 252, 585, 631, 632, 854, 856.
SHF_DYNAMIC: 245, 251, 276, 296, 488, 505.
shf_emptybuf: 239, 247, 249, 274, 275, 281, 285, 286, 287.
shf_eof: 244.
SHF_EOF: 244, 245, 274, 281, 285.
SHF_ERROR: 244, 245, 274, 275, 278, 281, 285, 286, 287.
shf_error: 244, 282, 289, 327, 1217.
shf_fdclose: 238, 253, 328.
shf_fdopen: 29, 238, 249, 250, 272, 600, 682.
shf_fillbuf: 239, 281, 282, 283, 284.
shf_flush: 238, 252, 253, 258, 274, 275, 297, 299, 300, 301, 302, 321, 327, 380, 465, 468, 529, 539, 694, 697, 703, 709, 740, 853, 877, 1215.
shf_fprintf: 238, 294, 301, 302, 305, 308, 311, 312, 465, 529, 694, 732, 734, 735, 736, 853, 854, 1271.
shf_getc: 244, 585, 1217.
shf_getchar: 238, 244, 284.
SHF_GETFL: 245, 247, 249.
shf_getse: 238, 283, 327.
SHF_H: 238.
SHF_INTERRUPT: 30, 245, 278, 281, 286, 287, 1214.
shf_iob: 240, 242, 250, 258.
SHF_MAPHI: 28, 228, 245, 600.
shf_open: 28, 228, 238, 245, 600, 856.
shf_putc: 244, 254, 289, 293, 306, 381, 875, 876, 1262, 1263.
shf_putchar: 238, 244, 286, 299, 300, 301, 302, 308, 479, 505, 694.
shf_puts: 238, 288, 321, 471, 631, 633, 1262, 1263.
SHF_RD: 29, 245, 247, 248, 249, 251, 282, 283, 284, 285, 600, 1214.
SHF_RDWR: 245, 248.
shf_read: 238, 282, 856.
SHF_READING: 245, 274, 275, 281.
shf_reopen: 114, 238, 245, 247, 539, 623, 706, 1214.
shf_sclose: 238, 251, 254, 295, 296, 488, 505.
shf_smprintf: 238, 296, 845.
shf_snprintf: 56, 167, 168, 238, 272, 295, 383, 458, 733, 1153, 1277, 1341, 1343.
shf_sopen: 238, 251, 254, 295, 296, 488, 505.
SHF_STRING: 245, 251, 274, 275, 281, 285, 496.
SHF_UNBUF: 237, 245, 247, 249, 255, 286.
shf_ungetc: 238, 285, 585.
shf_vfprintf: 238, 289, 294, 295, 296, 297, 298, 299, 300, 301, 302, 335.
SHF_WR: 114, 245, 247, 248, 249, 250, 251, 254, 272, 274, 286, 287, 295, 296, 488, 505, 539, 623, 682, 706.
shf_write: 238, 287, 288, 382, 383.
SHF_WRITING: 245, 274, 275, 277, 281, 285.
shift: 1206.
shl_j: 669, 682, 703, 706.
shl_out: 114, 157, 250, 297, 299, 300, 301, 302, 305, 321, 335, 380, 381, 382, 383, 465, 468, 529, 623, 694, 703, 709, 740, 875, 876, 877, 1271, 1338, 1342.
shl_spare: 250, 1214.
shl_stdout: 55, 250, 298, 306, 539, 697, 739, 853, 1226, 1255, 1256, 1262, 1263, 1275.
shl_stdout_ok: 241, 242, 298, 299, 301, 539.
show_lineno: 300.
shprintf: 55, 57, 243, 298, 306, 307, 696, 697, 902, 908, 910, 1208, 1221, 1235, 1236, 1244, 1245, 1246, 1247, 1248, 1256, 1258, 1259, 1261, 1262, 1263, 1274, 1287, 1291.
shtrap: 634, 640, 642.
si: 333, 334.
sig: 115, 659, 660, 701, 707, 708, 871, 1213, 1271, 1278.
SIG_BLOCK: 509, 689, 696, 707, 715, 716, 717, 725, 738, 739, 740.
SIG_DFL: 115, 634, 643, 646, 648, 653, 654, 655, 684, 686, 693, 730.
SIG_IGN: 634, 639, 643, 644, 646, 648, 653, 654, 682, 684, 685, 693.
SIG_SETMASK: 509, 681, 689, 691, 693, 696, 698, 699, 707, 715, 716, 717, 725, 738, 739, 740.
sig_t: 634, 638, 642, 648.
sigact: 642, 644.
Sigact_ign: 637, 639, 644.
Sigact_trap: 637, 639.
sigaction: 637, 639, 642, 644, 728, 730, 731.
sigaddset: 681.
SIGALRM: 642, 656, 659.
SIGCHLD: 509, 639, 642, 669, 678, 681, 689, 695, 701, 707, 708, 709, 718, 719, 720, 721,

722, 732, 1217.
SIGCONT: 688, 696, 707.
sigemptyset: 639, 642, 681, 730.
SIGERR_: 236, 463, 634, 635, 639, 642, 646.
SIGEXIT_: 236, 634, 635, 639, 642, 646.
SIGHUP: 639, 647, 688, 707.
SIGINT: 115, 232, 535, 639, 647, 649, 693, 733, 920, 1033.
sigjmp_buf: 107.
SIGKILL: 691.
siglongjmp: 236.
signal: 634, 639, 642, 643, 644, 646, 650, 651, 1271, 1277.
sigp: 717.
SIGPIPE: 654, 655, 733, 1237, 1242.
sigprocmask: 509, 681, 689, 691, 693, 696, 698, 699, 707, 715, 716, 717, 725, 738, 739, 740.
SIGQUIT: 115, 535, 639, 647, 693, 1033.
sigset_t: 509, 679, 680, 689, 696, 707, 715, 716, 717, 725, 738, 739, 740.
sigsetjmp: 229, 232, 378, 509, 515, 532, 631, 1170.
sigsuspend: 710.
SIGTERM: 115, 229, 639, 647, 707, 1271.
sigtraps: 115, 236, 535, 635, 636, 637, 639, 640, 641, 645, 650, 651, 652, 653, 654, 655, 659, 681, 682, 684, 685, 686, 693, 713, 733, 867, 1221, 1274, 1275, 1277.
SIGTSTP: 676, 682, 730, 731.
SIGTTIN: 676, 682, 686.
SIGTTOU: 676, 682.
SIGWINCH: 642, 867, 871.
simplify_path: 191, 742, 753, 1234.
sin: 1173.
sinh: 1173.
SINVALID: 330, 339.
size: 75, 76, 77, 88, 89, 90, 91, 92, 94, 96, 98, 101, 113, 133, 963.
SIZE_MAX: 75, 76, 77.
skip: 106.
skip_space: 1134, 1135, 1136.
skip_varname: 43, 104, 122, 144, 198, 199, 1281.
skip_wdvarname: 104, 199, 201, 202.
skiptabs: 373.
sleep: 688, 691.
slen: 165, 167, 572, 594, 596, 766, 768, 771, 772, 773, 791, 795, 796.
slenp: 594, 596.
SLETPAREN: 330, 339, 340, 367.
sletparen_info: 332.
slp: 739.
sm_default: 678, 679, 680, 681, 710.
sm_sigchld: 509, 678, 679, 680, 681, 689, 696, 707, 715, 716, 717, 725, 738, 739, 740.
SMALL: 103, 157, 823, 824, 839, 882, 967, 1021, 1045.
smark: 436.
smi: 1338, 1341.
sname: 450.
snprintf: 382, 386, 390, 403, 405, 406, 407, 411, 907.
snptreef: 457, 470, 488, 578, 600, 623, 626, 630, 689.
SOFT: 1285, 1290, 1291.
sold: 558, 600, 796, 797, 826, 855.
sortargs: 43, 58.
Source: 2, 4, 15, 227, 228, 230, 231, 313, 314, 315, 316, 317, 318, 319, 321, 325, 328, 368, 369, 370, 371, 412, 413, 421, 424, 558, 600, 631, 796, 812, 813, 815, 826, 840, 855, 1220.
source: 231, 232, 236, 305, 314, 315, 319, 320, 321, 323, 324, 325, 335, 339, 340, 369, 370, 371, 383, 405, 406, 420, 423, 424, 434, 444, 450, 451, 456, 558, 600, 631, 633, 797, 826, 855, 918, 947, 980, 1140, 1153, 1215, 1220, 1241.
sp: 101, 338, 367, 380, 381, 392, 394, 561, 562, 565, 567, 568, 570, 571, 572, 578, 579, 593, 594, 595, 597, 598, 599, 763, 764, 768, 781, 782, 785, 789, 790, 1144.
SPAT: 336, 357, 363, 486, 504, 505, 568.
SPATTERN: 330, 340, 341, 357, 363.
spec: 817.
SPEC_BI: 87, 301, 466, 537, 540, 546, 1247.
SPECIAL: 86, 113, 116, 119, 131, 139, 154, 156, 157, 158, 159, 163, 171, 172, 173, 178.
special: 103, 116, 119, 150, 154, 155, 160.
special_prompt_expand: 314, 377, 378.
specials: 103, 148, 150, 151.
split: 566, 584, 585, 597, 598, 600.
spp: 314, 379, 380.
sqrt: 76, 1173.
srand_deterministic: 159.
srchlen: 1022, 1143, 1145.
SRCHLEN: 1022, 1063, 1145.
srchpat: 1022, 1143, 1145.
SREREAD: 316, 317, 319, 323, 335, 370, 371.
srest: 615, 617, 618, 619.
SS_FORCE: 115, 642, 643, 653, 659, 681, 682, 684, 685, 686, 693.
SS_RESTORE_CURR: 115, 643, 648, 653, 654, 655.
SS_RESTORE_DFL: 643, 682, 685, 693.
SS_RESTORE_IGN: 643, 682, 684, 693.
SS_RESTORE_MASK: 642, 643.

SS_RESTORE_ORIG: 535, 639, 643, 659, 681, 684, 686, 867.
SS_SHTRAP: 642, 643, 659, 681, 867.
SS_USER: 642, 643, 648.
SSQUOTE: 330, 340, 342, 361.
SSTDIN: 29, 232, 235, 317, 318, 322, 325, 327, 368.
SSTRING: 27, 227, 317, 322, 423, 600, 633.
st: 560, 565, 567, 574, 579, 580, 581.
st_atime: 1192.
st_dev: 191, 1323.
st_gid: 1317.
st_head: 565, 567.
st_ino: 191, 1323.
st_mode: 31, 156, 255, 624, 765, 787, 788, 1192, 1194, 1197, 1317, 1318.
st_mtim: 823.
st_mtime: 1192, 1194, 1197, 1322.
st_size: 856, 1192, 1317.
st_uid: 839, 1317.
start: 27, 227, 232, 316, 319, 321, 322, 323, 324, 325, 328, 332, 346, 354, 368, 369, 370, 371, 558, 600, 606, 609, 610, 633, 753, 756, 797, 803, 804, 805, 806, 837, 838, 1006, 1007, 1059, 1118, 1119, 1147.
start_line: 424, 433, 434, 456.
start_token: 424, 433, 434, 456.
startlast: 600, 667, 715.
startp: 770, 803.
stat: 31, 156, 191, 255, 623, 624, 765, 786, 798, 812, 823, 839, 856, 1192, 1194, 1197, 1314, 1317, 1318, 1322, 1323.
stat_check: 786, 787, 788.
stat_done: 786.
statb: 156, 255, 623, 624, 765, 786, 787, 788, 798, 856, 1318.
state: 333, 334, 338, 339, 340, 342, 347, 351, 354, 361, 362, 367, 375, 376, 594, 597, 598, 599, 670, 672, 688, 690, 696, 697, 698, 701, 703, 704, 705, 707, 709, 710, 711, 712, 713, 714, 717, 720, 732, 733, 737, 739, 740, 817, 1022, 1028, 1029, 1063, 1065, 1066, 1067, 1068, 1070, 1071, 1072, 1073, 1074, 1075, 1145.
STATE_BSIZE: 313, 333, 334, 338, 339, 350.
State_info: 314, 332, 333, 334, 338.
state_info: 333, 334, 338, 339, 350.
statep: 333, 334, 338, 339, 346, 350, 353, 354, 358, 360, 361, 362.
staterr: 781, 783.
states: 338, 339.
status: 670, 672, 690, 697, 701, 704, 709, 712, 713, 714, 732, 733.
STBRACE: 330, 340, 347, 357.
stbuf: 1192, 1194, 1197.
stderr: 706.
STDIN_FILENO: 874.
stop: 61.
STOP_BRKCONT: 1223.
STOP_RETURN: 1222.
str: 27, 142, 227, 232, 308, 316, 319, 320, 321, 322, 324, 325, 328, 335, 368, 369, 370, 371, 377, 380, 382, 414, 417, 429, 439, 441, 450, 467, 468, 473, 474, 477, 499, 500, 506, 516, 517, 520, 521, 523, 535, 558, 562, 566, 578, 580, 581, 583, 584, 594, 595, 597, 598, 599, 600, 601, 602, 603, 604, 605, 633, 766, 771, 772, 773, 796, 797, 817, 834, 835, 836, 837, 838, 903, 928, 971, 987, 1161.
str_nsave: 10, 123, 142, 144, 213, 604, 605, 773, 786, 845, 992, 1173, 1174, 1260.
str_save: 10, 51, 156, 160, 164, 174, 212, 230, 378, 429, 439, 499, 501, 551, 558, 595, 613, 648, 775, 795, 815, 817, 821, 829, 847, 1194, 1195, 1224, 1264.
str_val: 65, 67, 104, 129, 135, 156, 157, 158, 171, 173, 182, 191, 194, 378, 403, 404, 465, 523, 529, 547, 549, 571, 580, 581, 595, 598, 599, 612, 623, 774, 795, 815, 881, 1162, 1170, 1186, 1231, 1232, 1234, 1258, 1337.
stravis: 824, 843.
strbuf: 174, 380, 382, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411.
strcasecmp: 641.
strchr: 37, 142, 324, 386, 388, 401, 530, 545, 570, 571, 572, 594, 595, 601, 606, 620, 621, 758, 763, 764, 767, 768, 773, 781, 785, 792, 806, 807, 826, 840, 846, 853, 881, 975, 1151, 1195, 1211, 1214, 1225, 1260, 1279.
strcmp: 24, 42, 50, 68, 90, 91, 96, 100, 124, 211, 305, 403, 404, 447, 614, 641, 778, 780, 817, 819, 821, 845, 907, 910, 911, 912, 913, 982, 1120, 1147, 1232, 1238, 1239, 1270, 1272, 1290, 1303, 1304, 1319, 1320, 1325, 1326, 1329, 1333.
strdup: 586, 903.
strerror: 28, 521, 523, 533, 535, 628, 630, 631, 632, 665, 685, 686, 687, 698, 699, 707, 711, 729, 731, 814, 818, 843, 854, 856, 1212, 1234, 1236, 1278, 1290.
strftime: 385, 386, 396, 397, 398, 399.
STRING: 275.
strip_nuls: 10, 328, 329, 856.
strlcat: 845.
strlcpy: 44, 139, 194, 212, 386, 391, 395, 400, 401, 402, 403, 404, 733, 768, 903, 981, 1145.

strlen: 24, 44, 55, 93, 132, 136, 164, 165, 175, 212, 272, 288, 291, 370, 382, 383, 403, 579, 581, 587, 595, 614, 724, 745, 746, 750, 752, 757, 758, 763, 764, 769, 779, 781, 783, 785, 789, 790, 795, 802, 829, 837, 840, 845, 899, 900, 903, 907, 926, 981, 986, 987, 989, 994, 1007, 1118, 1119, 1146, 1147, 1152, 1153, 1195, 1233, 1275, 1282, 1339, 1342.
strncasecmp: 641.
strncat: 213.
strncmp: 403, 621, 641, 724, 776, 795, 837, 987, 1147, 1175.
strp: 374.
strrchr: 43, 68, 821, 868.
strstr: 171, 377, 723, 829, 837, 868, 987, 1233.
strtok_r: 817.
strtol: 146, 215.
strtonum: 722, 1337.
strunvis: 840.
strv: 316, 322, 566, 584, 597, 598, 1220.
stype: 572, 573, 574, 579, 580, 594, 596, 597, 598, 599.
stypep: 594, 595, 596.
sub: 144, 622, 631, 1078, 1121, 1122, 1123, 1125, 1126, 1127, 1132.
submatch: 914, 920, 921, 923.
Subshell: 427, 448.
Subst: 340, 342, 352, 359.
subst_exstat: 5, 6, 465, 528, 530, 585, 1220, 1224.
substitute: 65, 144, 189, 330, 337, 378, 463, 555, 558, 571, 1198.
SubType: 565, 573, 574.
suffix: 311, 312.
`suspend`: 1228.
`SWORD`: 330, 339, 340, 367.
`SWORDS`: 317, 322, 1220.
`SWORDSEP`: 317, 322.
`SWSTR`: 317, 322, 558, 797.
symbol: 338, 412, 1332.
symbolic: 1207, 1208.
syncio: 445.
synio: 412, 445, 453, 454.
syniocf: 426, 428, 429, 444, 451, 452, 453.
syntax: 807.
syntaxerr: 412, 418, 419, 423, 428, 431, 432, 437, 438, 440, 443, 448, 455, 459, 1332.
sys_siglist: 639.
sys_signame: 639.
syscon: 677.
systime: 468, 672, 690, 701, 709.
s1: 76, 829, 1322.
s2: 76, 1322.
t: 84, 121, 210, 231, 380, 417, 418, 419, 420, 426, 431, 432, 436, 442, 443, 450, 460, 463, 467, 468, 471, 499, 500, 524, 542, 600, 614, 689, 753, 778, 853, 1077, 1078, 1260, 1265, 1271.
`T_ERR_EXIT`: 1302, 1325, 1326.
t_op: 1296, 1300, 1301, 1304.
table: 80, 81, 88, 89, 90, 91, 94, 96, 98, 101, 103, 105, 183, 184, 766, 775, 1260, 1265.
`TAILQ_ENTRY`: 897.
TAILQ_FOREACH: 899, 909, 910, 921, 982.
TAILQ_FOREACH_SAFE: 911, 912, 913.
`TAILQ_HEAD`: 898.
`TAILQ_HEAD_INITIALIZER`: 898.
`TAILQ_INIT`: 1016.
`TAILQ_INSERT_TAIL`: 903.
`TAILQ_REMOVE`: 906.
taliases: 87, 183, 184, 186, 548, 550, 553, 1260, 1265.
tan: 1173.
`TAND`: 414, 419, 472, 506, 511.
tanh: 1173.
`TASYNC`: 414, 420, 477, 506.
`TBANG`: 414, 428, 472, 506.
tbi: 545, 546.
tbl: 80, 84, 88, 90, 91, 93, 94, 95, 96, 97, 99, 100, 101, 103, 104, 113, 116, 117, 119, 121, 131, 132, 133, 136, 137, 140, 141, 148, 150, 151, 154, 155, 160, 163, 165, 171, 172, 173, 178, 182, 190, 191, 193, 194, 316, 369, 416, 461, 462, 463, 524, 533, 536, 539, 540, 541, 542, 545, 553, 554, 566, 571, 573, 575, 594, 613, 623, 775, 791, 872, 1149, 1158, 1159, 1168, 1169, 1170, 1176, 1178, 1186, 1187, 1188, 1192, 1198, 1214, 1225, 1230, 1243, 1250, 1255, 1257, 1260, 1262, 1265, 1280.
tblp: 94, 316, 324, 369.
tbls: 88, 89, 90, 91, 94, 96, 98, 101, 113, 133.
`TBRACE`: 414, 427, 477, 506.
tbuf: 614, 1161, 1162.
tc: 443.
`TCASE`: 414, 441, 474, 499, 506, 520.
tcgetattr: 665, 685, 711, 712.
tcgetpgrp: 686, 711.
tcol: 1050.
`TCOM`: 414, 426, 444, 450, 451, 463, 465, 468, 472, 506, 521, 600.
`TCOPROC`: 414, 420, 466, 477, 506.
tcopy: 414, 470, 499, 503, 505, 542.
`TCSADRAIN`: 698, 699, 712, 729, 869.
tcsetattr: 698, 699, 712, 729, 869.
tcsetpgrp: 687, 688, 692, 694, 698, 699, 711, 729, 731.
tcur: 1050.

tcursor: 1088, 1089, 1091.
TDBRACKET: 414, 426, 452, 472, 506.
te: 96, 452, 513, 775, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1317, 1319, 1321, 1324, 1325, 1326, 1329, 1330, 1331, 1332, 1333, 1334, 1335, 1336.
TEF_DBRACKET: 452, 513, 1302, 1308, 1319.
TEF_ERROR: 1302, 1305, 1306, 1307, 1308, 1309, 1317, 1321, 1324, 1326, 1332, 1336.
TELIF: 414, 432, 475, 506, 519.
Temp: 2, 3, 107, 243, 271, 272, 273, 631, 844.
temp: 545, 550, 551, 552, 852.
Temp_type: 243, 271, 272.
temp_type: 271.
temps: 107, 111, 196, 197, 632, 854.
tempvar: 1158, 1170, 1173, 1174, 1186, 1187.
TEOF: 233, 414, 423.
termios: 662, 663, 672, 869.
test: 621.
test: 1325.
test_aexpr: 1292, 1305, 1306.
test_eaccess: 1292, 1317, 1318.
Test_env: 412, 452, 461, 513, 1292, 1293, 1299, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1310, 1313, 1314, 1324, 1325, 1329, 1330, 1331, 1332, 1333, 1334, 1335, 1336.
test_env: 1299.
test_eval: 1293, 1313, 1314, 1335.
test_isop: 1293, 1303, 1304, 1308, 1329, 1333.
Test_meta: 412, 461, 1292, 1293, 1297, 1299, 1303, 1304, 1327, 1329, 1333.
test_nexpr: 1292, 1306, 1307.
test_oexpr: 1292, 1302, 1305, 1309, 1316.
Test_op: 412, 461, 1292, 1293, 1296, 1298, 1299, 1304, 1308, 1310, 1313, 1314, 1326, 1330, 1331, 1334, 1335.
test_parse: 452, 513, 1293, 1302, 1325.
test_primary: 1292, 1307, 1308.
texec: 524, 535.
TEXEC: 414, 472, 506, 521, 535.
texpand: 81, 89, 91, 92, 94.
tf: 468, 844, 854, 856.
TF_CHANGED: 634, 646, 648.
TF_DFL_INTR: 634, 639, 640, 645, 647, 648, 649, 650, 651.
TF_EXEC_DFL: 634, 643, 648, 653.
TF_EXEC_IGN: 634, 643, 648, 653.
TF_FATAL: 634, 639, 640, 645, 647, 648, 649, 650, 651, 710.
TF_NOARGS: 467, 468.
TF_NOREAL: 468.
TF_ORIG_DFL: 634, 643, 644, 654, 684.
TF_ORIG_IGN: 634, 642, 643, 644, 648, 654, 684.
TF_POSIX: 467, 468.
TF_SHELL_USES: 634, 639, 648, 659, 682, 867.
TF_TTY_INTR: 634, 639, 713.
TF_USER_SET: 634, 648, 652.
tfd: 665.
tflag: 1260, 1261, 1264.
TFOR: 414, 439, 473, 506, 514.
tfree: 470, 500, 532, 544.
TFUNC: 414.
TFUNCT: 414, 450, 477, 506.
then: 415, 431.
THEN: 415, 431.
thenpart: 412, 430, 431, 432.
thing: 1250, 1252, 1253, 1256, 1258.
TIF: 414, 430, 432, 475, 506, 519.
tilde: 554, 611, 612.
tilde_ok: 561, 565, 567, 569, 570, 571, 572, 577, 579, 586, 587, 591, 592, 593.
time: 385, 386, 396, 397, 398, 399.
time: 414, 415, 429, 506.
TIME: 415, 429.
timeradd: 468, 701.
timerclear: 468, 690.
timersub: 468, 701.
times: 1226.
timespec: 154, 311, 468, 1189, 1192, 1200.
timespeccmp: 823.
timespecsub: 154, 468, 1192.
timeval: 311, 312, 468, 669, 672, 1200.
timex: 468, 506, 1203.
timex_hook: 465, 467, 1203.
TIOCGWINSZ: 872.
tl: 418, 420, 443.
TLIST: 414, 420, 472, 506, 508.
tlist: 272.
tm: 380, 385, 386, 396, 397, 398, 399.
TM_AND: 1297, 1306, 1329.
TM_BINOP: 1297, 1303, 1308, 1326, 1329, 1333.
TM_CPAREN: 1297, 1309, 1329.
TM_END: 1297, 1302, 1303, 1326, 1329, 1333.
TM_NOT: 1297, 1307, 1326, 1329.
TM_OPAREN: 1297, 1308, 1329.
TM_OR: 1297, 1305, 1329.
TM_UNOP: 1297, 1303, 1304, 1308, 1329, 1333.
tmout_enum: 656, 657, 658.
TMOUT_EXECUTING: 325, 645, 656, 657.
TMOUT_LEAVING: 645, 656, 660.
TMOUT_READING: 325, 656, 660.
tmp: 141, 144, 289, 290, 292, 309, 371, 376, 378, 721, 739, 740, 818, 952, 953, 954, 996.
tmpbuf: 380, 386.

tmpdir: 81, 82, 156, 160, 272.
tnamecmp: 81, 100, 101.
to: 378.
TO_FILAXST: 1298, 1300, 1317.
TO_FILBDEV: 1298, 1300, 1317.
TO_FILCDEV: 1298, 1300, 1317.
TO_FILCDF: 1298, 1300, 1317.
TO_FILEQ: 1298, 1301, 1323.
TO_FILEX: 1298, 1300, 1317.
TO_FILEXST: 1298, 1300, 1317.
TO_FILFIFO: 1298, 1300, 1317.
TO_FILGID: 1298, 1300, 1317.
TO_FILGZ: 1298, 1300, 1317.
TO_FILID: 1298, 1300, 1317.
TO_FILNT: 1298, 1301, 1322.
TO_FILOT: 1298, 1301, 1322.
TO_FILRD: 1298, 1300, 1317.
TO_FILREG: 1298, 1300, 1317.
TO_FILSETG: 1298, 1300, 1317.
TO_FILSETU: 1298, 1300, 1317.
TO_FILSOCK: 1298, 1300, 1317.
TO_FILSTCK: 1298, 1300, 1317.
TO_FILSYM: 1298, 1300, 1317.
TO_FILTT: 1298, 1300, 1310, 1317.
TO_FILUID: 1298, 1300, 1317.
TO_FILWR: 1298, 1300, 1317.
TO_INTEQ: 1298, 1301, 1321.
TO_INTGE: 1298, 1301, 1321.
TO_INTGT: 1298, 1301, 1321.
TO_INTLE: 1298, 1301, 1321.
TO_INTLT: 1298, 1301, 1321.
TO_INTNE: 1298, 1301, 1321.
TO_NONOP: 1298, 1300, 1301, 1304, 1308, 1326.
TO_OPTION: 1298, 1300, 1316.
TO_STEQL: 1298, 1301, 1319, 1334.
TO_STGT: 1298, 1301, 1320.
TO_STLT: 1298, 1301, 1320.
TO_STNEQ: 1298, 1301, 1319, 1334.
TO_STNZE: 1298, 1300, 1308, 1315, 1326.
TO_STZER: 1298, 1300, 1315.
TODO: 2, 15, 58, 102, 105, 112, 115, 116, 121, 127, 132, 137, 140, 149, 178, 182, 189, 194, 229, 258, 293, 301, 325, 327, 332, 338, 339, 340, 344, 363, 364, 366, 367, 370, 371, 375, 377, 380, 414, 430, 431, 446, 448, 450, 463, 465, 470, 480, 506, 520, 523, 526, 532, 541, 545, 565, 566, 568, 573, 600, 625, 639, 643, 645, 648, 650, 665, 666, 684, 694, 703, 709, 712, 726, 769, 798, 807, 812, 821, 826, 854, 889, 892, 897, 931, 952, 970, 1005, 1024, 1052, 1114, 1147, 1173, 1177, 1183, 1185, 1212, 1235, 1326, 1334, 1342.
toglob: 771, 772, 773, 796, 797.
tok: 434, 817, 1162, 1169, 1170, 1172, 1173, 1174, 1175, 1176, 1177, 1184, 1188.
token: 338, 412, 418, 419, 420, 426, 432, 438, 440, 442, 443, 455, 459, 1158, 1163, 1166, 1169, 1175, 1176, 1178, 1188, 1332.
tokeninfo: 415, 416, 455.
tokens: 1303.
tokentab: 415, 416, 458.
tokp: 1162, 1169, 1170, 1172, 1173, 1174.
tolower: 170, 959, 960, 1156.
toolongseen: 840.
toplevel: 231, 235.
TOR: 414, 419, 472, 506.
totlen: 380, 381, 382, 383, 392, 394.
totncmp: 96.
toupper: 169, 959, 960, 1156.
tp: 3, 89, 90, 91, 92, 93, 94, 96, 98, 101, 148, 150, 151, 272, 273, 463, 464, 465, 466, 506, 523, 524, 526, 529, 530, 531, 532, 533, 534, 535, 536, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 611, 623, 630, 766, 775, 1243, 1244, 1245, 1246, 1247, 1248.
TPAREN: 414, 427, 472, 506.
TPAT: 414, 442, 443, 520.
tpeek: 338, 423, 442, 443, 444, 454, 1329, 1330, 1332.
TPIPE: 414, 418, 463, 466, 472, 506, 507.
tprintinfo: 96.
tputc: 471, 478, 479, 482, 483, 484, 485, 486, 489, 490, 491, 492, 494, 496, 498.
tputC: 469, 479, 481, 482, 483, 485.
tputS: 469, 480, 493.
tputs: 967, 1045.
TRACE: 85, 127, 532, 1246, 1252, 1253, 1259.
trap: 234, 236, 327, 463, 634, 635, 636, 640, 645, 646, 648, 651, 652, 874, 1221.
trap: 1221.
Trap: 105, 634, 636, 637, 638, 640, 641, 642, 643, 645, 646, 648, 650, 651, 652, 653, 654, 1221, 1271.
trap_pending: 638, 651, 710.
trapsig: 463, 638, 639, 640, 642, 648, 713, 920, 1033.
trapstr: 646.
tried_reset: 279.
trimsub: 554, 580, 601.
true: 15, 39, 40, 121, 144, 163, 202, 258, 299, 301, 302, 335, 338, 365, 370, 415, 418, 419, 420, 426, 430, 431, 432, 436, 437, 438, 439, 440, 442, 444, 449, 450, 455, 464, 509, 514, 528, 530, 532, 533, 535, 542, 571, 626, 627, 628, 630, 631, 632, 677, 693, 699, 701, 711, 716, 731, 773, 790, 850, 851,

873, 874, 887, 915, 934, 935, 950, 961, 991, 997, 1018, 1161, 1162, 1177, 1221, 1222, 1223, 1255, 1268, 1272, 1273, 1281, 1332.
true: 1205.
truncate: 966, 968.
try: 1230, 1234.
ts: 96, 98, 99, 311, 553, 611, 775, 1265.
TSELECT: 414, 439, 473, 506.
tsize: 89.
tstate: 80, 96, 97, 98, 99, 553, 775, 1265.
ts0: 468.
ts1: 468.
ts2: 468.
tt: 416, 455, 458.
TT_HEREDOC_EXP: 271, 632.
TT_HIST_EDIT: 271, 854.
tt_sigs: 676, 682, 684, 685, 693.
TTIME: 414, 429, 477, 506.
tty_close: 661, 664, 665, 684, 693.
tty_devtty: 662, 663, 665, 685.
tty_fd: 63, 662, 663, 664, 665, 685, 686, 687, 688, 692, 694, 698, 699, 709, 711, 712, 729, 731, 869, 872.
tty_init: 661, 665, 676, 677, 685, 731.
tty_state: 662, 663, 665, 685, 698, 699, 712, 729, 869.
ttyname: 391.
ttypgrp: 686.
ttypgrp_ok: 669, 684, 685, 686, 687, 692, 693, 698, 699, 700, 709, 729, 731.
ttystate: 671, 672, 698, 711.
TUNTIL: 414, 437, 476, 506, 518.
tv: 312.
tv_nsec: 311.
tv_sec: 154, 159, 311, 312, 1192.
tv_usec: 312.
tvar: 121, 123, 124, 125, 1172, 1173, 1174.
tvp: 1257.
tw: 499.
twb1: 1053, 1054, 1055, 1056.
twb2: 1053, 1056.
TWHILE: 414, 437, 476, 506, 518.
type: 84, 93, 103, 107, 111, 115, 117, 118, 121, 126, 127, 129, 130, 132, 139, 145, 148, 150, 164, 171, 172, 173, 175, 178, 185, 232, 233, 235, 236, 271, 272, 313, 316, 317, 318, 321, 322, 325, 327, 328, 335, 368, 369, 414, 416, 417, 421, 423, 426, 436, 460, 463, 465, 466, 468, 471, 472, 473, 475, 476, 478, 499, 507, 508, 509, 511, 514, 515, 518, 520, 524, 529, 532, 535, 540, 541, 545, 550, 552, 560, 561, 565, 566, 567, 570, 571, 572, 579, 580, 581, 582, 583, 584, 585, 586, 593, 600, 630, 1006, 1161, 1168, 1174, 1187, 1222, 1223, 1244, 1248, 1260.
type: 1243, 1249.
type_flags: 524, 529.
typeset: 1250.
typeset: 86, 104, 121, 123, 189, 525, 529, 535, 567, 872, 1215, 1255, 1280.
t0: 468.
t1: 460, 468, 471, 474.
t2: 460.
u: 84, 316, 414, 566, 623.
U_: 1026, 1027.
u_char: 792, 793.
u_ops: 1300, 1304.
u_sasparen: 332.
u_sbquote: 332.
u_scsparen: 332.
u_sletparen: 332.
uc: 1134, 1135, 1136.
UCASEV_AL: 86, 117, 127, 128, 164, 165, 174, 1252, 1253, 1254, 1255, 1259.
UCHAR_MAX: 205, 210, 601.
Uflag: 1260.
ugbuf: 316, 319, 323, 324.
uid_t: 5, 6.
uint32_t: 314, 315, 812, 818.
uint64_t: 10, 174, 175, 214, 595.
ulimit: 1283.
ulimit: 1285.
umask: 1207, 1208, 1210.
umask: 1207.
unalias: 1265.
UNCTRL: 882, 900, 972.
undo: 1022, 1031, 1077, 1089, 1154.
undobuf: 1022, 1031.
undocbuf: 1022.
UNESCAPE: 337, 344, 797.
unget_char: 1008, 1009, 1010.
ungetsc: 314, 316, 319, 320, 339, 341, 345, 346, 347, 348, 354, 360, 365, 366, 367, 370, 373, 375.
unit: 364, 365, 478, 623, 626, 629, 630.
unlink: 273.
unset: 1225.
unset: 104, 131, 140, 141, 1198, 1225, 1280.
unset_var: 1225.
unsetspec: 103, 113, 131, 160.
unspecial: 103, 151, 160.
until: 414, 415, 437, 506, 518.
until: 338.
UNTIL: 415, 437.
unwind: 4, 107, 108, 229, 232, 235, 236, 299, 300, 301, 304, 463, 509, 515, 524, 532, 645, 647, 693,

920, 1033, 1161, 1171, 1222, 1223.
uoptind: 32, 36, 154, 1280.
uqword: 1329, 1333.
usage: 1226.
use_cdpath: 747, 748.
use_tty: 685.
user_lineno: 154, 159.
user_opt: 33, 34, 36, 113, 154, 534, 1280, 1282.
USERATTRIB: 86, 117, 1258.
username: 5, 6, 194, 400.
usrtime: 468, 672, 690, 701, 709.
UTF-8: 1011, 1061.
u2: 84, 93, 117, 127, 139, 166, 167, 168, 533, 535, 547, 549, 1259.
u64ton: 10, 214, 491, 494, 595.
v: 148, 571, 791, 1168, 1170, 1290.
V_COLUMNS: 148, 149, 157.
V_EDITOR: 148, 149, 157.
v_evaluate: 163, 571, 1159, 1168, 1170, 1186.
V_HISTCONTROL: 148, 149, 157, 160.
V_HISTFILE: 148, 149, 157.
V_HISTSIZE: 148, 149, 154, 157.
V_IFS: 148, 149, 156, 160.
V_LINENO: 148, 149, 154, 159, 160.
V_MAIL: 148, 149, 158, 160.
V_MAILCHECK: 148, 149, 158, 160.
V_MAILPATH: 148, 149, 158, 160.
V_NONE: 148, 150.
V_OPTIND: 148, 149, 154, 156.
V_PATH: 148, 149, 156, 160.
V_POSIXLY_CORRECT: 148, 149, 156.
V_RANDOM: 148, 149, 154, 159, 160.
V_SECONDS: 148, 149, 154, 159, 160.
V_TERM: 148, 149, 157.
V_TMOUT: 148, 149, 159, 160.
V_TMPDIR: 148, 149, 156, 160.
V_VISUAL: 148, 149, 157.
va: 297, 298, 299, 300, 301, 302, 303, 304, 335, 487, 488, 489, 490, 491, 492, 493, 494, 495, 497.
va_arg: 290, 291, 490, 491, 492, 493, 494, 495, 497, 905.
va_end: 294, 295, 296, 297, 298, 299, 300, 301, 303, 304, 335, 487, 488, 905.
va_start: 294, 295, 296, 297, 298, 299, 300, 301, 303, 304, 335, 487, 488, 905.
val: 36, 84, 85, 86, 103, 116, 119, 120, 121, 122, 123, 125, 126, 127, 128, 129, 130, 131, 132, 133, 135, 136, 137, 139, 141, 147, 159, 164, 171, 172, 173, 174, 175, 178, 324, 369, 415, 416, 458, 524, 532, 535, 539, 540, 541, 542, 544, 548, 551, 553, 613, 630, 1152, 1162, 1168, 1169, 1170, 1173, 1174, 1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1187, 1206, 1245, 1248, 1255, 1256, 1260, 1262, 1263, 1264, 1266, 1267, 1268, 1290, 1291.
vallen: 132.
valp: 144.
vals: 140.
var: 121, 122, 123, 140, 562, 565, 566, 573, 574, 577, 579, 580, 581, 594, 597, 599, 1280, 1281.
VAR: 1162, 1163, 1164, 1173, 1177.
var?TODO: 1177.
VARASN: 337, 340, 426, 432, 444.
varcpy: 554, 574, 575.
VARG1: 1063, 1066.
VARG2: 1063, 1071.
varname: 562, 572, 578, 594.
vars: 105, 112, 113, 116, 117, 119, 120, 133, 134, 185, 414, 417, 426, 439, 442, 446, 448, 450, 460, 470, 472, 473, 474, 499, 500, 514, 520, 529, 600, 1257.
varsub: 554, 562, 572, 594.
vasn: 1176, 1178, 1179, 1188.
VCMD: 1063, 1064, 1070, 1072, 1145.
VDISCARD: 869.
VEOF: 869.
VERASE: 869.
verbose: 52, 54.
version_param: 24, 61, 192.
very_start: 753, 756.
VEXTCMD: 1063, 1064.
VFAIL: 1063, 1064, 1070.
Vflag: 1243.
vflag: 1243, 1244, 1245, 1246, 1247, 1248.
vfptreef: 469, 487, 488, 489.
vi: 61.
VI: 11, 17, 21, 23, 48, 325, 868, 873, 1021.
vi/*: 1117.
vi/+: 1141.
vi/,: 1123.
vi/-: 1142.
vi//: 1143.
vi/;: 1123.
vi/=: 1113.
vi/?: 1143.
vi/@: 1149.
vi/#: 1157.
vi/$: 1131.
vi/%: 1132.
vi/\: 1115.
vi/^[: 1114.
vi/^: 1129.
vi/^E: 1113.
vi/^F: 1115.

vi/^H: 1125.
vi/^I: 1116.
vi/^L: 1044.
vi/^N: 1141.
vi/^P: 1142.
vi/^R: 1046.
vi/^X: 1117.
vi/~: 1156.
vi/_: 1144.
vi/|: 1130.
vi/␣: 1126.
vi/a: 1079.
vi/A: 1080.
vi/b: 1121.
vi/B: 1121.
vi/c: 1102.
vi/C: 1104.
vi/d: 1102.
vi/D: 1105.
vi/e: 1122.
vi/E: 1122.
vi/f: 1123.
vi/F: 1123.
vi/g: 1140.
vi/G: 1140.
vi/h: 1125.
vi/i: 1081.
vi/I: 1082.
vi/j: 1141.
vi/k: 1142.
vi/l: 1126.
vi/n: 1143.
vi/N: 1143.
vi/p: 1108.
vi/P: 1109.
vi/r: 1083.
vi/R: 1084.
vi/s: 1085.
vi/S: 1086.
vi/t: 1123.
vi/T: 1123.
vi/u: 1154.
vi/U: 1155.
vi/v: 1153.
vi/w: 1127.
vi/W: 1127.
vi/x: 1106.
vi/X: 1107.
vi/y: 1102.
vi/Y: 1102.
vi/0: 1128.
vi/⟨*escape*⟩: 1114.
vi_cmd: 1023, 1074, 1075, 1077.
vi_error: 1023, 1035, 1059, 1065, 1067, 1073, 1074, 1075, 1083, 1089, 1118, 1119, 1145.
vi_hook: 1023, 1028, 1063, 1074.
vi_insert: 1023, 1065, 1088.
vi_macro_reset: 1023, 1029, 1030, 1032, 1035.
vi_pprompt: 1023, 1028, 1047, 1058, 1060.
vi_reset: 1023, 1028, 1029.
VINTR: 869.
VIS_NL: 824, 843.
VIS_SAFE: 824, 843.
VKILL: 869.
vl: 1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184.
VLIT: 1028, 1063, 1065.
VLNEXT: 869.
VMIN: 869.
VNORMAL: 1029, 1063, 1064, 1065, 1067, 1072, 1073, 1074, 1075, 1145.
void: 10.
voptarg: 1280.
vp: 113, 116, 117, 118, 119, 120, 121, 127, 128, 131, 132, 133, 134, 135, 136, 137, 139, 140, 141, 145, 146, 147, 154, 155, 156, 157, 158, 159, 160, 165, 166, 167, 168, 171, 172, 173, 174, 175, 178, 179, 182, 190, 193, 194, 575, 594, 595, 598, 791, 795, 872, 1169, 1170, 1186, 1187, 1192, 1198, 1214, 1215, 1225, 1250, 1256, 1257, 1258, 1259.
vp__: 226.
vpbase: 121, 125, 126, 127.
vpp: 113, 133.
vp2: 133, 134.
vq: 113, 117, 140, 163, 164, 171, 172, 1186, 1280.
VQUIT: 869.
vr: 1176, 1179, 1180, 1181, 1182, 1183, 1184, 1185.
VREDO: 1063, 1064.
VSEARCH: 1063, 1064, 1066.
vtemp: 103, 118, 145, 171.
VTIME: 869.
VWERASE: 869.
VXCH: 1063, 1064.
v1: 1321.
v2: 1321.
w: 471, 500, 556, 557, 559, 771, 796, 1275.
W_OK: 156, 261, 626, 1239, 1317.
wait: 1213.
waitfor: 667, 717, 1213.
waitlast: 585, 600, 667, 715, 716.
waitpid: 670, 701.
want: 791, 792, 793, 795.
want_letnum: 1134, 1135, 1136.
wantlen: 791, 795.

warningf: 39, 40, 65, 163, 243, 300, 528, 530, 533, 535, 600, 626, 627, 628, 630, 631, 632, 645, 665, 685, 686, 687, 699, 701, 711, 716, 729, 731, 818, 1161, 1162, 1222, 1223.
was_set: 542, 544.
wastty: 231, 232, 235.
wb: 1058.
wbsize: 237, 247, 249, 251, 276, 277, 287.
wbuf: 1031, 1037, 1047, 1048.
wbuf_len: 1031, 1037, 1047.
wb1: 1053, 1054, 1056, 1057.
wb2: 1053, 1056.
`WCOREDUMP`: 733.
wdcopy: 451, 470, 499, 501, 503, 1329.
wdleft: 274.
wdscan: 470, 503, 504, 572, 578.
wdstrip: 450, 470, 505.
werase: 859, 867, 869, 870, 1017, 1088, 1145.
`WEXITSTATUS`: 704, 714, 733.
what: 20, 43, 46, 58, 455, 456.
`whence`: 1243.
where: 709, 720, 721.
which: 737.
`WHILE`: 415, 437.
`while`: 414, 415, 437, 506, 518.
`whitespace`: 339.
`who`: 61.
width: 311, 312.
`WIFSIGNALED`: 701, 713.
`WIFSTOPPED`: 701.
win: 1031, 1037, 1047, 1048.
winleft: 1031, 1036, 1040, 1041, 1049, 1050, 1054, 1057, 1077.
winsize: 872.
winwidth: 1031, 1037, 1049, 1050, 1054, 1055, 1056, 1057.
wlen: 881.
wnleft: 237, 244, 245, 247, 249, 251, 253, 254, 276, 277, 278, 286, 287.
`WNOHANG`: 701.
word: 561, 563, 565, 567, 569, 570, 571, 572, 580, 581, 582, 584, 586, 593, 594, 596, 776, 777, 778.
wordlist: 412, 439, 440.
words: 770, 776, 780, 781, 783, 791, 795, 796, 797, 798, 799, 801, 802, 811, 1006, 1007, 1059, 1118, 1119, 1120.
wordsp: 770, 771, 796.
wp: 15, 62, 189, 237, 244, 245, 247, 249, 251, 276, 277, 278, 286, 287, 331, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 360, 361, 362, 363, 365, 366, 367, 371, 374, 375, 376, 460, 467, 480, 481, 482, 483, 485, 486, 503, 504, 505, 513, 536, 539, 561, 565, 586, 587, 606, 609, 610, 766, 769, 775, 781, 783, 784, 785, 786, 789, 790, 844, 846, 847, 848, 1200, 1205, 1206, 1207, 1212, 1213, 1214, 1215, 1216, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229, 1230, 1232, 1233, 1236, 1237, 1238, 1239, 1240, 1243, 1249, 1250, 1251, 1252, 1253, 1254, 1255, 1260, 1261, 1265, 1266, 1268, 1269, 1270, 1271, 1272, 1273, 1274, 1279, 1280, 1281, 1282, 1285, 1289, 1299, 1303, 1308, 1310, 1324, 1325, 1326, 1333, 1334.
wp_end: 1299, 1303, 1308, 1310, 1324, 1325.
write: 245, 262, 265, 268, 269, 270, 277, 278, 286, 287, 509, 705, 1242.
ws: 338, 339, 340, 344, 346, 347, 348, 349, 354, 365, 366, 367, 371, 374, 872.
ws_col: 872.
ws_row: 872.
wsp: 375, 376.
`WSTOPSIG`: 733.
`WTERMSIG`: 704, 713, 714, 733.
`WUNTRACED`: 701.
wv: 598.
x: 565.
`X_`: 1026, 1027.
x_abort: 885, 889, 891, 1017, 1018.
x_adj_done: 886, 916, 926, 971, 973, 1014.
x_adj_ok: 886, 916, 926, 931, 966, 1013.
x_adjust: 884, 886, 926, 938, 973, 1006, 1007, 1013.
x_arg: 886, 914, 919, 920, 924, 925, 928, 932, 933, 936, 937, 942, 943, 958, 974, 975, 978, 979, 980, 990, 991.
x_arg_defaulted: 886, 919, 925, 949, 980, 988, 991.
x_arg_set: 887, 924, 925.
x_basename: 776, 779, 799, 863, 1120.
x_beg_hist: 885, 889, 891, 976.
x_bind: 883, 907, 1279.
x_bind_quiet: 887, 903, 1017.
x_bs: 884, 926, 931, 938, 939, 953, 954, 961, 966, 973.
x_bword: 884, 934, 936, 940.
x_cf_glob: 770, 863, 1006, 1007, 1059, 1118, 1119.
X_chars: 859, 862, 863, 865, 869, 1017.
x_clear_screen: 885, 889, 892, 965.
x_col: 886, 916, 917, 931, 968, 969, 970, 1013, 1015.
x_cols: 157, 309, 310, 860, 861, 872, 886, 916, 1031, 1037.
x_command_glob: 770, 771, 862.
x_comment: 884, 885, 889, 891, 951.
x_comp_comm: 885, 889, 891, 998.
x_comp_file: 885, 889, 891, 1002.
x_comp_list: 885, 889, 891, 1004.

x_complete: 885, 889, 891, 1000.
x_cur_mode: 869.
x_debug_info: 885, 889, 1015.
x_del_back: 885, 889, 891, 932, 983, 1017.
x_del_bword: 885, 889, 891, 934, 1017.
x_del_char: 885, 889, 891, 933, 949.
x_del_fword: 885, 889, 891, 935.
x_del_line: 885, 889, 950, 1017.
x_delete: 884, 931, 932, 933, 934, 935, 973, 991, 994, 997, 1006, 1007.
x_displen: 886, 916, 917, 938, 961, 969, 973, 1015.
x_do_comment: 863, 878, 951, 1157.
x_do_ins: 926, 927, 1006, 1007.
x_draw_line: 885, 889, 964.
x_e_getc: 884, 925, 974, 975, 983, 1010, 1012.
x_e_getu8: 884, 919, 1012.
x_e_putc: 884, 917, 925, 927, 928, 931, 932, 933, 936, 937, 939, 942, 943, 946, 951, 952, 953, 958, 966, 967, 970, 971, 972, 974, 975, 981, 985, 986, 988, 996, 997, 1006, 1007, 1013, 1014, 1019.
x_e_puts: 884, 972, 983, 993, 994, 1014.
x_e_ungetc: 884, 925, 983, 1009, 1012.
x_emacs: 863, 873, 914.
x_emacs_keys: 863, 870, 887, 1017.
x_end_hist: 885, 889, 891, 977.
x_end_of_text: 885, 889, 892, 948, 949.
x_enumerate: 885, 889, 892, 1001.
x_eot_del: 885, 889, 892, 949.
x_error: 885, 889, 923, 1019.
x_escape: 863, 881, 1006, 1007, 1101, 1118, 1119.
x_expand: 885, 889, 892, 1007.
`X_EXTRA`: 220, 221.
x_file_glob: 770, 796, 862.
x_flush: 863, 877, 919, 946, 948, 966, 973, 983, 1015, 1028, 1033, 1035.
x_fold_capitalize: 885, 889, 891, 957.
x_fold_case: 884, 955, 956, 957, 958.
x_fold_lower: 885, 889, 891, 956.
x_fold_upper: 885, 889, 892, 955.
x_free_words: 796, 798, 801, 811, 863, 1006, 1059, 1119.
x_ftab: 888, 889, 897, 903, 904, 907, 908, 913.
x_fword: 884, 935, 937, 941.
x_getc: 863, 874, 1010, 1028, 1032.
x_goto: 884, 931, 932, 936, 938, 941, 942, 943, 944, 945, 954, 958, 974, 975, 981, 985, 986, 991, 994, 996, 997, 1006, 1007.
x_goto_hist: 885, 889, 892, 980.
x_histp: 887, 915, 947, 978, 979, 981, 986.
x_init: 63, 864, 867.
x_init_emacs: 863, 867, 1016.
x_ins: 884, 923, 926, 928, 973, 989, 990, 993, 994, 1006, 1007.
x_ins_string: 885, 889, 912, 922, 929.
x_insert: 885, 889, 923, 928, 929, 982, 983.
x_kill: 885, 889, 892, 991.
x_kill_region: 885, 889, 997.
x_last_command: 887, 914, 920, 994.
x_lastcp: 884, 926, 931, 961, 969, 971, 981, 1015.
x_list_comm: 885, 889, 892, 999.
x_list_file: 885, 889, 892, 1003.
x_literal: 885, 889, 892, 930.
x_literal_set: 887, 914, 920, 930.
x_load_hist: 884, 918, 976, 977, 978, 979, 980, 981, 986.
x_locate_word: 770, 803, 862.
x_longest_prefix: 799, 802, 863, 1006, 1119.
x_match: 884, 985, 986, 987.
x_meta_yank: 885, 889, 892, 994.
x_mode: 863, 869, 873, 874, 887, 920, 1033.
x_mv_back: 885, 889, 891, 893, 942.
x_mv_begin: 885, 889, 891, 894, 896, 944.
x_mv_bword: 885, 889, 891, 896, 940.
x_mv_end: 885, 889, 891, 894, 896, 945.
x_mv_forw: 885, 889, 892, 893, 943.
x_mv_fword: 885, 889, 892, 896, 941.
x_newline: 885, 889, 892, 946, 947, 951.
x_next_com: 885, 889, 891, 893, 979.
x_nextcmd: 887, 918, 947, 1016.
x_nl_next_com: 885, 889, 892, 947.
x_noop: 885, 889, 1017, 1020.
`X_OK`: 156, 261, 523, 548, 549, 626, 763, 765, 783, 1264, 1317, 1318.
x_prev_com: 885, 889, 892, 893, 978.
x_prev_histword: 885, 889, 892, 988.
x_print_expansions: 799, 863, 1006, 1059, 1119.
x_push: 884, 931, 950, 992.
x_putc: 799, 801, 863, 875, 948, 967, 1013, 1028, 1035, 1045, 1047, 1056, 1057, 1058, 1062, 1089.
x_puts: 863, 876, 1062.
x_read: 326, 864, 873, 914.
x_redraw: 884, 950, 951, 964, 965, 966, 973, 981, 983, 993, 994, 1006, 1015.
x_search: 884, 983, 984, 985, 986.
x_search_char_back: 885, 889, 892, 974.
x_search_char_forw: 885, 889, 892, 975.
x_search_hist: 885, 889, 892, 983.
x_set_arg: 885, 889, 895, 925.
x_set_mark: 885, 889, 892, 995.
x_sigwinch: 862, 867, 871.
x_size: 884, 931, 939, 961, 962, 963.
x_size_str: 884, 950, 951, 963, 981.
x_transpose: 885, 889, 892, 952.

x_try_array: 770, 791.
x_tty: 887, 907, 1016.
x_vi: 863, 873, 1028.
x_vi_putbuf: 1023, 1101, 1118, 1119.
x_vi_zotc: 1023, 1033, 1034, 1062.
x_xchg_point_mark: 885, 889, 892, 996.
x_yank: 885, 889, 892, 993, 994.
x_zotc: 884, 938, 948, 953, 954, 971, 972, 1018, 1062.
x_zots: 884, 926, 931, 969, 971.
XARG: 565, 584, 594, 597, 598.
XARGSEP: 565, 584.
XBASE: 561, 562, 565, 567, 568, 569, 572, 579, 582, 583, 584, 585, 593, 594, 600.
XBGND: 463, 507, 509, 510, 535, 689, 690, 692, 693, 694.
xbp: 886, 915, 938, 950, 951, 961, 966, 969, 970, 973, 981, 1015, 1018.
xbuf: 886, 915, 920, 932, 936, 942, 944, 949, 950, 951, 952, 953, 966, 970, 971, 973, 974, 975, 981, 985, 986, 991, 1006, 1007, 1015, 1018.
XCCLOSE: 463, 507, 509, 689.
XCF_COMMAND: 766, 770, 998, 999.
XCF_COMMAND_FILE: 766, 1000, 1001, 1004, 1059, 1118, 1119.
XCF_FILE: 766, 1002, 1003, 1007.
XCF_FULLPATH: 766, 771, 783, 1059, 1118, 1119.
Xcheck: 217, 220, 223, 338, 340, 344, 348, 349, 371, 373, 374, 375, 376, 565, 570, 571, 611, 758, 785, 1216, 1240.
*Xcheck_grow_*: 217, 219, 223.
XcheckN: 223, 325, 327, 750, 751, 752, 760, 764, 781, 782, 789, 790, 829, 856.
Xclose: 217, 224, 367, 373, 374, 587, 757, 764, 829.
XCOM: 565, 570, 585, 600.
XCOPROC: 463, 509, 689, 693, 694.
xcp: 886, 915, 926, 927, 931, 932, 933, 936, 937, 938, 941, 942, 943, 950, 951, 952, 953, 954, 958, 961, 966, 973, 974, 975, 981, 991, 992, 993, 994, 995, 996, 997, 1006, 1007, 1015, 1018.
xend: 886, 915, 927, 951, 981.
xep: 886, 915, 920, 927, 931, 933, 937, 943, 945, 946, 949, 950, 951, 952, 958, 961, 966, 970, 974, 975, 981, 991, 1006, 1007, 1015, 1018.
XERROK: 463, 506, 508, 511, 512, 516, 517, 518, 519, 520, 532, 689, 693.
xerrok: 463, 468, 506, 507, 508, 510, 511, 512, 516, 517, 518, 519, 520, 524, 532, 535, 689.
XEXEC: 463, 464, 506, 507, 509, 510, 524, 527, 535, 689, 693.
xf: 903, 904.
XF_ARG: 888, 889.
xf_flags: 888, 902, 908.
xf_func: 888, 904, 913, 920, 922, 982.
xf_name: 888, 902, 907, 908.
XF_NOBIND: 888, 889, 902, 908.
XF_PREFIX: 888.
xflag: 1260, 1262.
XFORK: 463, 464, 506, 507, 509, 510, 600, 689.
Xfree: 221, 365, 366, 611, 763, 781, 784, 1215, 1241.
Xinit: 217, 220, 221, 339, 368, 373, 374, 567, 587, 611, 757, 763, 781, 784, 829, 856, 1215, 1234, 1237.
Xlength: 225, 327, 328, 338, 365, 586, 587, 747, 757, 786, 790, 856, 1215, 1216, 1242.
xlp: 886, 915, 926, 950, 961, 966, 970, 971, 1015, 1018.
xlp_valid: 887, 915, 926, 931, 950, 961, 969, 973, 981, 1018.
xmp: 887, 915, 931, 950, 993, 995, 996, 997.
Xnleft: 225, 327, 789, 856.
XNULLSUB: 565, 572, 580, 582, 594, 597, 598.
xp: 132, 217, 218, 219, 221, 222, 223, 224, 225, 325, 326, 327, 328, 373, 594, 595, 597, 598, 599, 600, 747, 750, 751, 752, 757, 758, 759, 760, 761, 762, 763, 764, 781, 782, 784, 785, 786, 787, 788, 789, 790, 807, 829, 856, 1214, 1215, 1216, 1230, 1234, 1235, 1237, 1240, 1241, 1242.
XPclose: 133, 226, 426, 440, 442, 557, 771, 797.
XPCLOSE: 463, 464, 507, 689.
xpe: 807.
XPfree: 226, 426, 556, 559, 598, 771, 801.
XPinit: 133, 226, 426, 440, 442, 556, 557, 559, 598, 771, 797, 801.
XPIPE: 463.
XPIPEI: 463, 507, 689, 690, 693.
XPIPEO: 463, 507, 600, 689.
xpos: 373.
xpp: 785, 789, 790.
XPptrv: 226, 448, 556, 559, 598, 769, 776, 780, 783, 801, 1332.
XPput: 133, 226, 426, 440, 442, 446, 451, 557, 586, 587, 598, 609, 769, 771, 775, 786, 797, 801, 1329, 1330.
XPsize: 226, 444, 446, 448, 556, 559, 598, 769, 771, 781, 784, 1332.
XPtrV: 133, 226, 426, 440, 442, 554, 555, 556, 557, 559, 565, 598, 606, 766, 769, 771, 775, 781, 784, 785, 796, 801, 1299.
Xput: 217, 222, 373, 750, 751, 757, 760, 1215, 1216, 1218, 1237, 1240, 1241.
xr: 997.
Xrestpos: 225, 354, 373, 579, 580, 581, 762.

xs: 217, 218, 221, 222, 223, 224, 225, 316, 325, 327, 328, 368, 373, 757, 763, 764, 781, 782, 784, 785, 786, 787, 788, 789, 790, 829, 856, 1214, 1215, 1216, 1230, 1234, 1235, 1237, 1240, 1241, 1242.
Xsavepos: 225, 346, 373, 574, 760.
Xsize: 225.
xsp: 217, 219, 747, 750, 751, 752, 758, 759, 760, 761, 762.
xstrcmp: 10, 43, 211, 769, 780.
Xstring: 218, 325, 328, 338, 371, 572, 580, 582, 611, 747, 759, 761, 762, 764, 781, 785, 786, 787, 788, 790, 856, 1215, 1234, 1235, 1241, 1242.
XString: 217, 218, 219, 220, 223, 224, 225, 314, 316, 338, 373, 374, 375, 554, 565, 611, 741, 742, 747, 757, 758, 763, 781, 784, 785, 829, 856, 1214, 1230, 1237.
XStringP: 218.
`XSUB`: 562, 564, 565, 580, 581, 583, 586, 594, 597, 598, 599.
`XSUBMID`: 562, 565, 583, 586.
`XTIME`: 463, 465, 468.
xx_cols: 886, 916, 917, 931, 966, 968, 969, 970, 973, 1013, 1015.
`XXCOM`: 463, 507, 600, 689, 690, 693, 694.
yank_range: 1023, 1102, 1105, 1106, 1107, 1110.
yanklen: 1022, 1108, 1109, 1110.
ybuf: 1022, 1108, 1109, 1110.
`YYERRCODE`: 366, 415.
yyerror: 313, 335, 338, 373, 376, 439, 445, 450, 453, 454, 455, 456.
yylex: 220, 313, 314, 331, 336, 337, 338, 361, 412, 417, 420, 427, 558, 633, 797.
yylval: 314, 315, 331, 365, 367, 369, 439, 440, 441, 442, 446, 449, 451, 454, 457, 558, 633, 797, 1329, 1330, 1332.
yyparse: 314, 412, 418, 423, 424.
YYSTYPE: 314, 315, 331.
`Z_`: 1026, 1027.
zero_ok: 594, 595, 597.
`ZEROFIL`: 86, 117, 127, 128, 167, 168, 179, 1252, 1254, 1259.
zflag: 739.

⟨ (Re-)start argument parsing 38 ⟩ Used in section 37.
⟨ (**if** (*flags* & FC_SPECBI)) Search for a special built-in 546 ⟩ Used in section 545.
⟨ Abort getopts if the state is invalid 1282 ⟩ Used in section 1280.
⟨ Add a character to the Emacs mode history search 985 ⟩ Used in section 983.
⟨ Append an entry to the history file (and unlock) 824 ⟩ Used in section 821.
⟨ Append end-pattern 580 ⟩ Used in section 579.
⟨ Append the extraction from *cdpathp* 751 ⟩ Used in section 747.
⟨ Append the formatted string 293 ⟩ Used in section 289.
⟨ Append the path component at *xp* 760 ⟩ Used in section 758.
⟨ Append *file* 752 ⟩ Used in section 747.
⟨ Apply an assignment substitution 581 ⟩ Used in section 579.
⟨ Apply the MAGIC glob 790 ⟩ Used in section 785.
⟨ Attach tty to job 698 ⟩ Used in section 696.
⟨ Attempt to SIGCONT a job 699 ⟩ Used in section 696.
⟨ Back from suspend—reset signals, pgrp & tty 731 ⟩ Used in section 728.
⟨ Back out the last history entry 828 ⟩ Cited in section 826. Used in section 827.
⟨ Bourne commands 311, 312, 467, 468, 1205, 1206, 1207, 1212, 1213, 1214, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229 ⟩ Used in section 1200.
⟨ Build option strings 44 ⟩ Used in section 43.
⟨ Cache the function pathname found 552 ⟩ Used in section 549.
⟨ Cache the pathname found 551 ⟩ Used in section 549.
⟨ Calculate fc's default *hfirst* & *hlast* 850 ⟩ Used in section 844.
⟨ Calculate the range to copy from the buffer 1103 ⟩ Used in section 1102.
⟨ Cancel the current line and **return** 1090 ⟩ Used in section 1088.
⟨ Carry out a command 531, 532, 535 ⟩ Used in section 524.
⟨ Case-fold the first character 959 ⟩ Used in section 958.
⟨ Case-fold the remaining characters 960 ⟩ Used in section 958.
⟨ Change to the new directory 1234 ⟩ Used in section 1230.
⟨ Check for ./ or ../ 748 ⟩ Used in section 747.
⟨ Check for a valid subshell or function 448 ⟩ Used in section 444.
⟨ Check for end of word or $IFS separation 586 ⟩ Cited in sections 561 and 569. Used in section 565.
⟨ Check for pattern replacement argument 847 ⟩ Used in section 846.
⟨ Check for qualifiers in *word* part 596 ⟩ Used in section 594.
⟨ Check for search prefix 848 ⟩ Used in section 846.
⟨ Check for traps &c. 234 ⟩ Used in section 233.
⟨ Check if the globbed file exists or for the empty string 798 ⟩ Used in section 796.
⟨ Check that the variable name is valid 122 ⟩ Used in section 121.
⟨ Clean up and *exec* in subprocess 693 ⟩ Used in section 689.
⟨ Clear a key binding and **return** 911 ⟩ Used in section 907.
⟨ Clear the screen 967 ⟩ Used in section 966.
⟨ Close a broken or old style history file and **return** 820 ⟩ Used in section 815.
⟨ Close freshly duplicated file-descriptors 630 ⟩ Used in section 623.
⟨ Collect non-special or quoted characters to form word 340 ⟩ Used in section 338.
⟨ Common headers 8 ⟩ Used in section 7.
⟨ Common **Source** sources 322 ⟩ Used in section 321.
⟨ Compile and interpret an expression 233 ⟩ Cited in section 463. Used in section 231.
⟨ Convert an empty path to "." and **break** 754 ⟩ Used in section 753.
⟨ Convert an integer to a string 135 ⟩ Used in section 133.
⟨ Convert the search string into a pattern 772 ⟩ Used in section 771.
⟨ Convert unset tty characters to internal 'unset' value 870 ⟩ Used in section 869.
⟨ Copy a variable into the local **Block** 117 ⟩ Used in section 116.
⟨ Copy and expand characters to print 1240 ⟩ Used in section 1237.

⟨ Copy the alias to the macro buffer 1152 ⟩ Used in section 1149.
⟨ Copy the next entry from *path* into *xp* 782 ⟩ Used in section 781.
⟨ Copy the next prompt character 381 ⟩ Used in section 380.
⟨ Copy the previous command's last argument 989 ⟩ Used in section 988.
⟨ Copy the previous command's x_arg^{th} argument 990 ⟩ Used in section 988.
⟨ Create a copy of *argv* 59 ⟩ Used in section 15.
⟨ Create a new **tbl** entry in *p* 93 ⟩ Used in section 91.
⟨ Create subprocess 691 ⟩ Used in section 689.
⟨ Create temp file to hold heredoc content 632 ⟩ Used in section 631.
⟨ Define built-in commands 187 ⟩ Used in section 15.
⟨ Define `MAGIC` 767 ⟩ Used in section 7.
⟨ Deny localising `$PATH`, `$ENV` or `$SHELL` if `FRESTRICTED` 124 ⟩ Used in section 121.
⟨ Detect `$(`-expansion 346 ⟩ Used in section 345.
⟨ Detect `${`-expansion 347 ⟩ Used in section 345.
⟨ Detect `$`-expansion 345 ⟩ Used in section 343.
⟨ Detect `$`*variable*-expansion 348 ⟩ Cited in section 562. Used in section 345.
⟨ Detect `\`-escapes 344 ⟩ Used in section 343.
⟨ Detect assignment to an array 371 ⟩ Used in section 340.
⟨ Detect backtick-expansion 350 ⟩ Used in section 343.
⟨ Detect compilation of a debugging build 12 ⟩ Used in section 7.
⟨ Detect escapes and substitutions 343 ⟩ Cited in sections 342, 350, 352, and 359. Used in section 340.
⟨ Detect matching a pattern and **switch to** `SPATTERN` 341 ⟩ Cited in sections 355, 357, and 363. Used in section 340.
⟨ Detect privilege early 25 ⟩ Used in section 15.
⟨ Detect punctuation and `$`*[0-9]*-expansion 349 ⟩ Used in section 345.
⟨ Detected "`[`" in *has_globbing* 808 ⟩ Used in section 807.
⟨ Detected "`]`" in *has_globbing* 809 ⟩ Used in section 807.
⟨ Detected pattern in *has_globbing* 810 ⟩ Used in section 807.
⟨ Determine a source expression's inline base 179 ⟩ Used in section 178.
⟨ Determine the post-formatted string's length 166 ⟩ Used in section 165.
⟨ Differentiate between `sh` and `ksh` 24 ⟩ Used in section 23.
⟨ Display the current *umask* 1208 ⟩ Used in section 1207.
⟨ Display the provenance of a function 1246 ⟩ Used in section 1244.
⟨ Display the provenance of a name 1244 ⟩ Used in section 1243.
⟨ Display the provenance of a shell built-in 1247 ⟩ Used in section 1244.
⟨ Display the provenance of an alias 1245 ⟩ Used in section 1244.
⟨ Display the provenance of an executable 1248 ⟩ Used in section 1244.
⟨ Do some kind of history thing in Emacs mode 918 ⟩ Used in section 914.
⟨ Do substitutions on the content of the heredoc 633 ⟩ Used in section 631.
⟨ Down-case a string 170 ⟩ Used in section 165.
⟨ Draw a continuation character at the end of the tty 970 ⟩ Used in section 966.
⟨ Draw on the tty from the input buffer 969 ⟩ Used in section 966.
⟨ Edit the temporarily-saved history 855 ⟩ Used in section 844.
⟨ Emacs mode keybindings 891, 892, 893, 894, 895, 896 ⟩ Used in section 1016.
⟨ Emacs mode source 890, 897, 899, 900, 901, 902, 903, 905, 906, 907, 914, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 955, 956, 957, 958, 961, 962, 963, 964, 965, 966, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 986, 987, 988, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1006, 1007, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020 ⟩ Used in section 882.
⟨ Emacs mode static functions 884, 885 ⟩ Used in section 882.
⟨ Emacs mode static variables 886, 887, 889, 898, 1008 ⟩ Used in section 882.
⟨ Emacs or VI flags 21 ⟩ Used in section 20.

⟨ Emit a complete word 587 ⟩ Cited in section 584. Used in section 586.
⟨ Enable and/or re-enable FRESTRICTED & FERREXIT 67 ⟩ Used in section 65.
⟨ End of source path and **return** 786 ⟩ Used in section 785.
⟨ Ensure a vi macro is not called recursively 1151 ⟩ Used in section 1149.
⟨ Erase the current word and **return** 1091 ⟩ Used in section 1088.
⟨ Establish a return point for break/continue 515 ⟩ Used in sections 514 and 518.
⟨ Establish an error handler (*Include*) 229 ⟩ Used in section 228.
⟨ Establish an error handler (*shell*) 232 ⟩ Used in section 231.
⟨ Evaluate and **return** a test operator 1315, 1316, 1317, 1319, 1320, 1321, 1322, 1323 ⟩ Used in section 1314.
⟨ Evaluate command arguments 464, 465 ⟩ Used in section 463.
⟨ Evaluate high-precedence expressions and **return** it 1177 ⟩ Used in section 1176.
⟨ Evaluate the included file's contents 230 ⟩ Used in section 228.
⟨ Execute a syntax tree node and **break** or **Break** 506 ⟩ Used in section 463.
⟨ Execute a TASYNC node and **break** 510 ⟩ Used in section 506.
⟨ Execute a TBANG node and **break** 512 ⟩ Used in section 506.
⟨ Execute a TCASE node and **break** 520 ⟩ Used in section 506.
⟨ Execute a TCOPROC node and **break** 509 ⟩ Used in section 506.
⟨ Execute a TDBRACKET node and **break** 513 ⟩ Used in section 506.
⟨ Execute a TEXEC node and **break** 521 ⟩ Used in section 506.
⟨ Execute a TFOR node 516 ⟩ Used in section 514.
⟨ Execute a TFOR/TSELECT node and **break** 514 ⟩ Used in section 506.
⟨ Execute a TIF/TELIF node and **break** 519 ⟩ Used in section 506.
⟨ Execute a TLIST node and **break** 508 ⟩ Used in section 506.
⟨ Execute a TOR/TAND node and **break** 511 ⟩ Used in section 506.
⟨ Execute a TPIPE node and **break** 507 ⟩ Used in section 506.
⟨ Execute a TSELECT node 517 ⟩ Used in section 514.
⟨ Execute a TWHILE/TUNTIL node and **break** 518 ⟩ Used in section 506.
⟨ Execute initialisation statements 62 ⟩ Used in section 15.
⟨ Expand $@/$* 597 ⟩ Used in section 594.
⟨ Expand a hash table if necessary 92 ⟩ Used in section 91.
⟨ Expand a prompt meta-character 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411 ⟩ Used in section 382.
⟨ Expand a variable 599 ⟩ Used in section 594.
⟨ Expand a COMSUB character 570 ⟩ Used in section 568.
⟨ Expand a CSUBST character 579 ⟩ Used in section 568.
⟨ Expand alternate expressions 610 ⟩ Used in section 606.
⟨ Expand an array 598 ⟩ Used in section 594.
⟨ Expand an EXPRSUB character 571 ⟩ Used in section 568.
⟨ Expand an OQUOTE character 569 ⟩ Used in section 568.
⟨ Expand an OSUBST character 572 ⟩ Cited in section 594. Used in section 568.
⟨ Expand the nth word 1120 ⟩ Used in section 1119.
⟨ Expand to the length of a string or array 595 ⟩ Used in section 594.
⟨ Expand **switch** (*type*) ≡ (XARGSEP ∨ XARG) 584 ⟩ Used in section 565.
⟨ Expand **switch** (*type*) ≡ (XSUB ∨ XSUBMID) 583 ⟩ Used in section 565.
⟨ Expand **switch** (*type*) ≡ XBASE 568 ⟩ Used in section 565.
⟨ Expand **switch** (*type*) ≡ XCOM 585 ⟩ Used in section 565.
⟨ Expand **switch** (*type*) ≡ XNULLSUB 582 ⟩ Used in section 565.
⟨ Expect a command in vi mode 1070 ⟩ Used in sections 1069 and 1071.
⟨ Externally-linked variables 6, 14, 19, 34, 49, 73, 82, 110, 153, 162, 184, 206, 242, 264, 315, 538, 636, 658, 662, 675, 680, 744, 861, 865, 1202, 1328 ⟩ Used in section 7.
⟨ Extract number's base from string 180 ⟩ Used in section 178.
⟨ Figure out $PWD 191 ⟩ Used in section 15.

⟨ Figure out how many rows & columns 309 ⟩ Used in section 308.
⟨ Figure out if this is a command 806 ⟩ Used in section 803.
⟨ Figure out the column width 310 ⟩ Used in section 308.
⟨ Figure out where the variable-name part ends 376 ⟩ Used in section 375.
⟨ Fill in an (unset & undefined) array entry 139 ⟩ Used in section 136.
⟨ Fill in the (maybe allocated) next/nested substitution 574 ⟩ Used in section 572.
⟨ Fill the current display buffer 1054 ⟩ Used in section 1053.
⟨ Filter the globbed list for executables 783 ⟩ Used in section 781.
⟨ Find a key-binding definition 921 ⟩ Cited in section 899. Used in section 920.
⟨ Find a matching close brace if any 607 ⟩ Used in section 606.
⟨ Find and remove a named alias 1266 ⟩ Used in section 1265.
⟨ Find job *j* and process *p* structures for this *pid* 702 ⟩ Used in section 701.
⟨ Find *func* in **x_ftab** 904 ⟩ Used in section 903.
⟨ Find $n \leftarrow |vp\rightarrow val.i|$ and *base* 175 ⟩ Used in section 174.
⟨ Finish an `ALIAS` or `KEYWORD` 369 ⟩ Used in section 367.
⟨ Finish the token (may **return** or **restart**) 367 ⟩ Cited in section 338. Used in section 338.
⟨ Flag is special or invalid 39 ⟩ Used in section 37.
⟨ Flush a file's read/write buffers 277 ⟩ Used in section 275.
⟨ Flush a string's read/write buffers 276 ⟩ Used in section 275.
⟨ Format a tree part 490, 491, 492, 493, 494, 495, 496, 497, 498 ⟩ Used in section 489.
⟨ Format an integer as a string 174 ⟩ Used in section 173.
⟨ Found `\` in a prompt, determine what it is 382 ⟩ Used in section 381.
⟨ Free any old allocation and undefine a user function 544 ⟩ Used in section 542.
⟨ Free up an entire array 141 ⟩ Used in section 131.
⟨ Get a job's status and print its command 697 ⟩ Used in section 696.
⟨ Get dimensions of the `select` list 1339 ⟩ Used in section 1338.
⟨ Get the history item matching a number 835 ⟩ Used in section 834.
⟨ Get the history item matching a string 836 ⟩ Used in section 834.
⟨ Glob "*str*"`*` to an array of words 797 ⟩ Used in section 796.
⟨ Global variables 5, 18, 33, 61, 72, 109, 152, 161, 183, 188, 241, 263, 537, 635, 657, 674, 679, 743, 860 ⟩ Used in section 3.
⟨ Globbing 766, 770, 771, 773, 775, 778, 779, 781, 791, 796, 799, 802, 803, 811 ⟩ Used in section 866.
⟨ Handle input in Emacs mode 920 ⟩ Used in section 914.
⟨ Handle `EOF` 235 ⟩ Used in section 233.
⟨ Ignore blank lines 422 ⟩ Used in section 420.
⟨ Import the calling process' environment 189 ⟩ Used in section 15.
⟨ Increase *num* by the value of the next digit 181 ⟩ Used in section 178.
⟨ Initialise (non-)interactivity 66 ⟩ Used in section 15.
⟨ Initialise interactive editing 63 ⟩ Used in section 15.
⟨ Initialise interactivity 30 ⟩ Used in section 29.
⟨ Initialise job control 673 ⟩ Used in section 15.
⟨ Initialise signal masks & handlers 681 ⟩ Used in section 677.
⟨ Initialise the lexing state machine 339 ⟩ Cited in section 359. Used in section 338.
⟨ Initialise to read from a file 28 ⟩ Used in section 26.
⟨ Initialise to read from standard input 29 ⟩ Cited in section 815. Used in section 26.
⟨ Initialise to run a command string 27 ⟩ Used in section 26.
⟨ Initialise *expand* state 567 ⟩ Used in section 565.
⟨ Inject the current line number 383 ⟩ Used in section 381.
⟨ Insert a character in vi mode 1099 ⟩ Used in section 1088.
⟨ Insert a new entry into an array 138 ⟩ Used in section 136.
⟨ Is the variable being assigned to? 128 ⟩ Used in section 127.
⟨ KSH commands 36, 1230, 1236, 1237, 1243, 1249, 1250, 1260, 1265, 1268, 1269, 1270, 1271, 1277, 1279, 1280 ⟩ Used in section 1201.

⟨ Leave vi insert mode and **return** 1092 ⟩ Used in section 1088.
⟨ Left justify a string 168 ⟩ Used in section 165.
⟨ Lexing SASPAREN state ($((...))) 354 ⟩ Cited in section 346. Used in section 340.
⟨ Lexing SBASE handle csh-style history 370 ⟩ Used in section 340.
⟨ Lexing SBQUOTE state (` ... `) 358 ⟩ Cited in section 350. Used in section 340.
⟨ Lexing SBRACEQ state ("${⟨ *var* ⟩}") 356 ⟩ Cited in section 347. Used in section 340.
⟨ Lexing SBRACE state (${⟨ *var* ⟩}) 355 ⟩ Cited in section 347. Used in section 340.
⟨ Lexing SCSPAREN state ($(...)) 353 ⟩ Cited in sections 346 and 354. Used in section 340.
⟨ Lexing SDQUOTE state ("...") 352 ⟩ Cited in section 342. Used in section 340.
⟨ Lexing SHEREDELIM state (<</<<- delimiter) 361 ⟩ Cited in sections 339 and 362. Used in section 340.
⟨ Lexing SHEREDQUOTE state (" in <</<<- delimiter) 362 ⟩ Cited in section 361. Used in section 340.
⟨ Lexing SLETPAREN state (((...))) 360 ⟩ Cited in section 339. Used in section 340.
⟨ Lexing SPATTERN state ([*+?@!](...|...)) 363 ⟩ Cited in section 341. Used in section 340.
⟨ Lexing SSQUOTE state ('...') 351 ⟩ Cited in sections 342 and 361. Used in section 340.
⟨ Lexing STBRACE state (${*var*[#%]...}) 357 ⟩ Cited in section 347. Used in section 340.
⟨ Lexing SWORD state (*single word*) 359 ⟩ Cited in section 339. Used in section 340.
⟨ Link a new process into jobs list 690 ⟩ Used in section 689.
⟨ List a named alias and **continue** 1263 ⟩ Used in section 1260.
⟨ List a single variable or array 1258 ⟩ Used in section 1257.
⟨ List all aliases 1262 ⟩ Used in section 1260.
⟨ List all possible signals 1275 ⟩ Used in section 1274.
⟨ List all user functions 1256 ⟩ Used in section 1250.
⟨ List all variables 1257 ⟩ Used in section 1250.
⟨ List history 853 ⟩ Used in section 844.
⟨ List of all test operators 1298 ⟩ Used in section 1296.
⟨ List signals and **return** 1274 ⟩ Used in section 1271.
⟨ Load a function that's in $FPATH or **break** 533 ⟩ Used in section 532.
⟨ Locate a word's beginning in *start* 804 ⟩ Used in section 803.
⟨ Locate or create an array entry 137 ⟩ Used in section 136.
⟨ Locate the command's details 530 ⟩ Used in section 524.
⟨ Locate the end of a word in *end* 805 ⟩ Used in section 803.
⟨ Locate the value in an assignment 123 ⟩ Used in section 121.
⟨ Look for a numeric argument 42 ⟩ Used in section 37.
⟨ Look for a possibly optional argument 40 ⟩ Used in section 37.
⟨ Look for a process matching %?⟨ *string* ⟩ 723 ⟩ Used in section 722.
⟨ Look for a process matching %⟨ *string* ⟩ 724 ⟩ Used in section 722.
⟨ Look for an attached argument 41 ⟩ Used in section 37.
⟨ Look for the next macro character 1032 ⟩ Used in section 1028.
⟨ Look for the real *stderr* 706 ⟩ Used in section 703.
⟨ Look up macro letter in alias list 1150 ⟩ Used in section 1149.
⟨ Maintain the history file's size 822 ⟩ Used in section 821.
⟨ Make directories look like directories 788 ⟩ Used in section 786.
⟨ Make the *hlast hfirst* and the *hfirst hlast* 852 ⟩ Used in section 844.
⟨ Mark an in-use function to be deleted later 543 ⟩ Used in section 542.
⟨ Mark brace expansion characters 590 ⟩ Used in section 586.
⟨ Mark character class characters 588 ⟩ Used in section 586.
⟨ Mark first unquoted : for ~ 592 ⟩ Used in section 586.
⟨ Mark first unquoted = for ~ 591 ⟩ Used in section 586.
⟨ Mark glob characters 589 ⟩ Used in section 586.
⟨ Mark the job as started 694 ⟩ Used in section 689.
⟨ Match 0/1 ("*"/"+") or more times and **return** 617 ⟩ Used in section 615.
⟨ Match a globbing "*" and **return** 616 ⟩ Used in section 615.

⟨ Match none of the patterns and **return** 619 ⟩ Used in section 615.
⟨ Match once or not ("?") or one of many patterns ("@") and **return** 618 ⟩ Used in section 615.
⟨ Mathematical Operators 1167 ⟩ Used in section 1158.
⟨ Move the file descriptor to FDBASE or above 246 ⟩ Used in section 245.
⟨ Names of special variables 149 ⟩ Used in section 148.
⟨ No valid alternation expansion 609 ⟩ Used in section 606.
⟨ Note when co-process dies 705 ⟩ Used in section 703.
⟨ Obtain process status and job state 704 ⟩ Used in section 703.
⟨ Open a new file for redirection 627 ⟩ Used in section 623.
⟨ POSIXise standard input 31 ⟩ Used in section 15.
⟨ Pad the current display buffer to the right margin 1055 ⟩ Used in section 1053.
⟨ Parse a command based on its first token 427, 428, 429, 430, 437, 439, 441, 449, 451, 452 ⟩ Used in section 426.
⟨ Parse a numeric *umask* 1209 ⟩ Used in section 1207.
⟨ Parse a regular command word 444 ⟩ Used in section 427.
⟨ Parse a symbolic *umask* 1210 ⟩ Used in section 1207.
⟨ Parse an expression beginning with a letter 1173 ⟩ Used in section 1172.
⟨ Parse an expression beginning with a number 1174 ⟩ Used in section 1172.
⟨ Parse an expression beginning with a symbol 1175 ⟩ Used in section 1172.
⟨ Parse and act on a limit 1288 ⟩ Used in section 1285.
⟨ Parse brief options to kill 1272 ⟩ Used in section 1271.
⟨ Parse command line arguments 26 ⟩ Used in section 15.
⟨ Parse full options to kill 1273 ⟩ Used in section 1271.
⟨ Parse options to typeset &c. 1251, 1252, 1253, 1254 ⟩ Used in section 1250.
⟨ Parse symbol umask characters 1211 ⟩ Used in section 1210.
⟨ Parse the arguments to getopts 1281 ⟩ Used in section 1280.
⟨ Path is relative—append *cwd* 750 ⟩ Used in section 747.
⟨ Perform a mathematical operation 1180, 1181, 1182, 1183, 1184, 1185 ⟩ Used in section 1179.
⟨ Perform a vi mode command 1044, 1046, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1102, 1104, 1105, 1106, 1107, 1108, 1109, 1113, 1114, 1115, 1116, 1117, 1140, 1141, 1142, 1143, 1144, 1149, 1153, 1154, 1155, 1156, 1157 ⟩ Used in section 1077.
⟨ Perform a vi movement command 1121, 1122, 1123, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132 ⟩ Used in section 1078.
⟨ Perform the default action and **unwind** or **return** 647 ⟩ Used in section 646.
⟨ Perform the lower-precision operations 1179 ⟩ Used in section 1176.
⟨ Permit files to look like directories 787 ⟩ Used in section 786.
⟨ Possibly re-attempt an interrupted flush, or **return** 278 ⟩ Used in section 277.
⟨ Prepare Emacs mode buffers 915 ⟩ Used in section 914.
⟨ Prepare Emacs mode tty 916 ⟩ Used in section 914.
⟨ Prepare I/O redirection for execution 466 ⟩ Used in section 463.
⟨ Prepare a new command object 550 ⟩ Used in section 549.
⟨ Prepare a variable for exporting 126 ⟩ Used in section 121.
⟨ Prepare for interaction 682 ⟩ Used in section 677.
⟨ Prepare redirection token and **return** 365 ⟩ Used in section 338.
⟨ Prepare to read another line and **continue** 1219 ⟩ Used in section 1216.
⟨ Prepend # and *base* 177 ⟩ Used in section 174.
⟨ Prepend $n \% base$ and reduce until $n \equiv 0$ 176 ⟩ Used in section 174.
⟨ Pretend we got an interrupt 1033 ⟩ Used in section 1028.
⟨ Print a job ID and status 734 ⟩ Used in section 732.
⟨ Print a job's first line 735 ⟩ Used in section 732.
⟨ Print a job's remaining pipeline 736 ⟩ Used in section 732.
⟨ Print a key binding and **return** 910 ⟩ Used in section 907.
⟨ Print a syntax tree node 472, 473, 474, 475, 476, 477 ⟩ Used in section 471.

⟨ Print a variable's flags 1259 ⟩ Used in section 1258.
⟨ Print all ulimit limits 1287 ⟩ Used in section 1285.
⟨ Print as-is if quotation is unnecessary 307 ⟩ Used in section 306.
⟨ Print options quietly 57 ⟩ Used in section 54.
⟨ Print options verbosely 55 ⟩ Cited in section 53. Used in section 54.
⟨ Print the prompt in Emacs mode 917 ⟩ Used in section 914.
⟨ Print to the terminal 1242 ⟩ Used in section 1237.
⟨ Process I/O redirection for irregular commands 453 ⟩ Used in section 426.
⟨ Process $0 and function arguments 534 ⟩ Used in section 532.
⟨ Process a complete line 328 ⟩ Used in section 325.
⟨ Process a flag option and **break** 58 ⟩ Used in section 45.
⟨ Process a long option and **break** 46 ⟩ Used in section 45.
⟨ Process a matched key-binding in Emacs mode 922 ⟩ Used in section 920.
⟨ Process an fc option 845 ⟩ Used in section 844.
⟨ Process an unmatched key-binding in Emacs mode 923 ⟩ Used in section 920.
⟨ Process no token and **return** or **restart** 366 ⟩ Used in section 338.
⟨ Put a lexicographic string 481, 482, 483, 484, 485, 486 ⟩ Used in section 480.
⟨ Quit the primary environment and *exit* 115 ⟩ Used in section 114.
⟨ Re-reading source code 323 ⟩ Used in section 321.
⟨ Read a line interactively using the editor 326 ⟩ Used in section 325.
⟨ Read a line simply 327 ⟩ Used in section 325.
⟨ Read a string into the next variable 1216 ⟩ Used in section 1215.
⟨ Read another line and **continue** 1218 ⟩ Used in section 1216.
⟨ Read the next character 1217 ⟩ Used in section 1216.
⟨ Read the next variable into a string 291 ⟩ Used in section 289.
⟨ Read the profile file(s) 65 ⟩ Used in section 15.
⟨ Read variables from the file descriptor 1215 ⟩ Used in section 1214.
⟨ Reading an alias 324 ⟩ Used in section 321.
⟨ Receive a literal 'EOF' (^D) character 1034 ⟩ Used in section 1028.
⟨ Redraw the last line of the prompt 968 ⟩ Used in section 966.
⟨ Reload the history file if it's changed (after locking) 823 ⟩ Used in section 821.
⟨ Remove a "../" component and **continue** 756 ⟩ Used in section 753.
⟨ Remove a history search pattern character in Emacs mode 984 ⟩ Used in section 983.
⟨ Remove a path component at *xp* and **continue** 759 ⟩ Used in section 758.
⟨ Remove an alias 1267 ⟩ Used in section 1265.
⟨ Remove the oldest zombie from the job list 727 ⟩ Used in section 726.
⟨ Remove MAGIC and *globit* again 789 ⟩ Used in section 785.
⟨ Replace a non-macro key binding and **return** 913 ⟩ Used in section 907.
⟨ Report (details of) a syntax error 456, 457, 458 ⟩ Used in section 455.
⟨ Report a bad substitution 578 ⟩ Used in section 572.
⟨ Report an error attempting redirection and **return** 628 ⟩ Used in section 623.
⟨ Reset Emacs mode meta sequence 924 ⟩ Used in section 914.
⟨ Reset all tracked aliases and **return** 1261 ⟩ Used in section 1260.
⟨ Reset *tilde_ok* 593 ⟩ Used in section 586.
⟨ Reset *xp* if the symlink is absolute 762 ⟩ Used in section 758.
⟨ Resolve a symlink in the path so far or **continue** 761 ⟩ Used in section 758.
⟨ Restore tty & pgrp 729 ⟩ Used in section 728.
⟨ Restore tty settings if they are unchanged 712 ⟩ Used in section 709.
⟨ Restrict potential system calls 16 ⟩ Used in section 15.
⟨ Right justify a string 167 ⟩ Used in section 165.
⟨ Run the fixed history commands 856 ⟩ Used in section 844.
⟨ Save a file-descriptor prior to redirection 629 ⟩ Used in section 623.

⟨ Save a redirection in *iops* 445 ⟩ Used in section 444.
⟨ Save a string value in a string variable 164 ⟩ Used in section 163.
⟨ Save a token in *args* or *vars* 446 ⟩ Used in section 444.
⟨ Save flags the first time a signal action is set 644 ⟩ Used in section 642.
⟨ Save selected history lines to a temporary file 854 ⟩ Used in section 844.
⟨ Save the `tty`'s current process group 711 ⟩ Used in section 709.
⟨ Save the local buffer in the fresh variable 130 ⟩ Used in section 127.
⟨ Save the new directory 1235 ⟩ Used in section 1230.
⟨ Save the value in a local buffer 129 ⟩ Used in section 127.
⟨ Scan `echo` arguments 1238, 1239 ⟩ Used in section 1237.
⟨ Scan arguments 45 ⟩ Used in section 43.
⟨ Scan the format string for flags 290 ⟩ Used in section 289.
⟨ Search `$PATH` or `$FPATH` 549 ⟩ Used in section 545.
⟨ Search for a tracked alias 548 ⟩ Used in section 545.
⟨ Search for a user-defined function 547 ⟩ Used in section 545.
⟨ Search for an historic command 849 ⟩ Used in section 846.
⟨ Search for *name* in each entry in *path* and **return** it 764 ⟩ Used in section 763.
⟨ Search tables for an internal command or a function 774 ⟩ Used in section 771.
⟨ Set `$USER` and the main prompt 190 ⟩ Used in section 15.
⟨ Set `$_` to the last command 525 ⟩ Used in section 524.
⟨ Set a macro key binding and **return** 912 ⟩ Used in section 907.
⟨ Set a single un-named `ulimit` limit 1289 ⟩ Used in section 1285.
⟨ Set an alias' value and/or flags 1264 ⟩ Used in section 1260.
⟨ Set default flag values 23 ⟩ Used in section 15.
⟨ Set process group ID 687 ⟩ Used in section 685.
⟨ Set special I/O, getopt & POSIX variables 156 ⟩ Used in section 155.
⟨ Set special history & interaction variables 157 ⟩ Used in section 155.
⟨ Set special variables that need no introduction 159 ⟩ Used in section 155.
⟨ Set special variables to do with mail 158 ⟩ Used in section 155.
⟨ Set the global `$PATH` variable 193 ⟩ Used in section 15.
⟨ Set the type of new variables 529 ⟩ Used in section 524.
⟨ Set up I/O duplication and **break** 626 ⟩ Used in section 623.
⟨ Set up job control 692 ⟩ Used in section 689.
⟨ Set up output redirection and **break** 624 ⟩ Used in section 623.
⟨ Set up redirection from a heredoc and **break** 625 ⟩ Used in section 623.
⟨ Set up the base environment 185 ⟩ Used in section 15.
⟨ Set up variable and command dictionaries 186 ⟩ Used in section 15.
⟨ Set variables & attributes and **return** 1255 ⟩ Used in section 1250.
⟨ Set *rv* from a job started under `JF_PIPEFAIL` 714 ⟩ Used in section 709.
⟨ Set/clear variable flags 127 ⟩ Cited in section 121. Used in section 121.
⟨ Shared function declarations 4, 10, 70, 104, 243, 413, 462, 555, 638, 667, 742, 813, 864, 883, 1159, 1190, 1203, 1204, 1294 ⟩ Used in section 7.
⟨ Show all Emacs mode function names and **return** 908 ⟩ Used in section 907.
⟨ Show all key bindings and **return** 909 ⟩ Used in section 907.
⟨ Signal a process 1278 ⟩ Used in section 1271.
⟨ Simulate `^C` to loops &c. 713 ⟩ Used in section 709.
⟨ Skip "`./`" and **continue** 755 ⟩ Used in section 753.
⟨ Skip possible array de-reference 200 ⟩ Used in section 199.
⟨ Sort and remove duplicate entries 780 ⟩ Used in section 771.
⟨ Sort by basename, then path order 776 ⟩ Used in section 771.
⟨ Special treatment for built-in `builtin` 526 ⟩ Used in section 524.
⟨ Special treatment for built-in `command` 528 ⟩ Used in section 524.

⟨ Special treatment for built-in `exec` 527 ⟩ Used in section 524.
⟨ Store initial `$PPID` & `$(K)SH_VERSION` 192 ⟩ Used in section 15.
⟨ Stringify a number 292 ⟩ Used in section 291.
⟨ Stringify process state into *buf* 733 ⟩ Used in section 732.
⟨ Strip identical directory names and **return** 801 ⟩ Used in section 799.
⟨ Substitute and execute command 846 ⟩ Used in section 844.
⟨ Substitute substitutions 577 ⟩ Used in section 572.
⟨ Substitute trims 576 ⟩ Used in section 572.
⟨ Suspend the shell 730 ⟩ Used in section 728.
⟨ Take a stab at argument count from here 793 ⟩ Used in section 791.
⟨ Temporarily disable `-re` 64 ⟩ Used in section 65.
⟨ Temporarily enable `nohup` by default for three decades 22 ⟩ Used in section 23.
⟨ Test and **return** a binary expression 1312 ⟩ Used in section 1308.
⟨ Test and **return** a parenthetical expression 1309 ⟩ Used in section 1308.
⟨ Test and **return** a unary expression 1311 ⟩ Used in section 1308.
⟨ Transpose GNU style 954 ⟩ Used in section 952.
⟨ Transpose Gosling style 953 ⟩ Used in section 952.
⟨ Trim the longest match at the beginning (`##`) and **break** 603 ⟩ Used in section 601.
⟨ Trim the longest match at the end (`%%`) and **break** 605 ⟩ Used in section 601.
⟨ Trim the shortest match at the beginning (`#`) and **break** 602 ⟩ Used in section 601.
⟨ Trim the shortest match at the end (`%`) and **break** 604 ⟩ Used in section 601.
⟨ Try to comment out the input and **return** 880 ⟩ Used in section 878.
⟨ Try to find the array 794 ⟩ Used in section 791.
⟨ Turn job control off 684 ⟩ Used in section 683.
⟨ Turn job control on 685 ⟩ Used in section 683.
⟨ Type definitions 17, 32, 47, 71, 84, 88, 97, 105, 106, 107, 218, 226, 237, 262, 271, 316, 331, 332, 364, 414, 433, 566, 573, 634, 656, 670, 671, 777, 859, 888, 1005, 1024, 1036, 1111, 1160, 1163, 1164, 1165, 1169, 1191, 1199, 1276, 1284, 1296, 1297, 1299, 1340 ⟩ Cited in section 897. Used in section 7.
⟨ Uncomment the input and **return** 0 879 ⟩ Used in section 878.
⟨ Unescape `\` or detect quotes and **switch state** 342 ⟩ Cited in sections 356 and 360. Used in section 340.
⟨ Unexpected evaluation; **break** 1162 ⟩ Used in section 1161.
⟨ Unexport any redefined instances 134 ⟩ Used in section 133.
⟨ Up-case a string 169 ⟩ Used in section 165.
⟨ Update the "more character(s)" 1057 ⟩ Used in section 1053.
⟨ Update the terminal from *wb1* 1056 ⟩ Used in section 1053.
⟨ Use *fcntl* to figure out the read/write flags 248 ⟩ Used in sections 247 and 249.
⟨ Validate `fc`'s supplied *hfirst* & *hlast* 851 ⟩ Used in section 844.
⟨ Validate arguments/flags if the variable is read-only 125 ⟩ Used in section 121.
⟨ Wait for a foregrounded job to finish 700 ⟩ Used in section 696.
⟨ Wait for input in Emacs mode 919 ⟩ Used in section 914.
⟨ Wait to be given tty 686 ⟩ Used in section 685.
⟨ Wait while the job's running 710 ⟩ Used in section 709.
⟨ Walk back to find start of command 792 ⟩ Used in section 791.
⟨ Walk the array and build words list 795 ⟩ Used in section 791.
⟨ `cd` with one argument—go there 1232 ⟩ Used in section 1230.
⟨ `cd` with two arguments—substitute in `$PWD` 1233 ⟩ Used in section 1230.
⟨ `cd` with zero arguments—go `$HOME` 1231 ⟩ Used in section 1230.
⟨ `jobs.c` **static** functions 668 ⟩ Used in section 666.
⟨ `jobs.c` **static** variables 669, 676 ⟩ Used in section 666.
⟨ `misc.c` declarations 52, 207 ⟩ Used in section 9.
⟨ `misc.c` variables 48, 205 ⟩ Used in section 9.
⟨ `test` special cases from POSIX 1326 ⟩ Used in section 1325.

⟨ ulimit limits 1286 ⟩ Used in section 1285.
⟨ "Extract" the first element of *cdpathp* 749 ⟩ Used in section 747.
⟨ "Insert" a backspace character and **return** 1089 ⟩ Used in section 1088.
⟨ "Print" to the history 1241 ⟩ Used in section 1237.
⟨ alloc.c 69, 74, 75, 76, 77, 78, 79 ⟩
⟨ c_ksh.c 1201 ⟩
⟨ c_sh.c 1200 ⟩
⟨ c_test.c 1292, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1310, 1313, 1314, 1318, 1324, 1325 ⟩
⟨ c_test.h 1293 ⟩
⟨ c_ulimit.c 1283, 1285, 1290, 1291 ⟩
⟨ charclass.h 204 ⟩
⟨ config.h 11 ⟩
⟨ edit.c 862, 866, 867, 868, 869, 871, 872, 873, 874, 875, 876, 877, 878, 881 ⟩
⟨ edit.h 863 ⟩
⟨ emacs.c 882 ⟩
⟨ eval.c 554, 556, 557, 558, 559, 565, 575, 594, 600, 601, 606, 611, 612, 613, 768, 769, 784, 785 ⟩
⟨ exec.c 461, 463, 523, 524, 536, 539, 540, 541, 542, 545, 553, 623, 631, 763, 765, 1333, 1334, 1335, 1336, 1337, 1338, 1341, 1342, 1343 ⟩
⟨ expand.h 217 ⟩
⟨ expr.c 1158, 1161, 1168, 1170, 1172, 1176, 1178, 1186, 1187, 1188 ⟩
⟨ history.c 812, 814, 815, 816, 817, 818, 819, 821, 825, 826, 827, 829, 830, 831, 832, 833, 834, 837, 838, 839, 840, 841, 842, 843, 844, 857 ⟩
⟨ io.c 240, 250, 255, 256, 257, 258, 259, 260, 261, 265, 266, 267, 268, 269, 270, 272, 297, 298, 299, 300, 301, 302, 303, 304, 305 ⟩
⟨ jobs.c 666, 672, 677, 683, 688, 689, 695, 696, 701, 703, 707, 708, 709, 715, 716, 717, 718, 719, 720, 721, 722, 725, 726, 728, 732, 737, 738, 739, 740 ⟩
⟨ lex.c 314, 319, 320, 321, 325, 333, 334, 335, 338, 368, 372, 373, 374, 375, 377, 378, 379, 380 ⟩
⟨ lex.h 313 ⟩
⟨ mail.c 1189, 1192, 1193, 1194, 1195, 1196, 1197, 1198 ⟩
⟨ misc.c 9, 20, 35, 37, 43, 50, 51, 53, 54, 56, 102, 209, 210, 211, 212, 213, 214, 215, 216, 219, 279, 280, 306, 308, 329, 614, 615, 620, 621, 622, 746, 807 ⟩
⟨ path.c 741, 745, 747, 753, 757, 758 ⟩
⟨ sh.h 7 ⟩
⟨ shf.c 239, 245, 247, 249, 251, 252, 253, 254, 274, 275, 281, 282, 283, 284, 285, 286, 287, 288, 289, 294, 295, 296 ⟩
⟨ shf.h 238 ⟩
⟨ syn.c 412, 415, 416, 417, 418, 419, 420, 421, 423, 424, 426, 431, 432, 434, 435, 436, 438, 440, 442, 443, 447, 450, 454, 455, 459, 460, 1327, 1329, 1330, 1331, 1332 ⟩
⟨ table.c 81, 83, 89, 90, 91, 94, 95, 96, 98, 99, 100, 101 ⟩
⟨ table.h 80 ⟩
⟨ trap.c 637, 639, 640, 641, 642, 643, 645, 646, 648, 649, 650, 651, 652, 653, 654, 655, 659, 660 ⟩
⟨ tree.c 469, 471, 478, 479, 480, 487, 488, 489, 499, 500, 501, 502, 503, 504, 505 ⟩
⟨ tree.h 470 ⟩
⟨ tty.c 663, 664, 665 ⟩
⟨ tty.h 661 ⟩
⟨ var.c 103, 112, 113, 116, 119, 121, 131, 132, 133, 136, 140, 142, 143, 144, 148, 150, 151, 154, 155, 160, 163, 165, 171, 172, 173, 178, 182, 198, 199, 201, 202, 203 ⟩
⟨ version.c 13 ⟩
⟨ vi insert mode non-standard commands 1093, 1094, 1095, 1096, 1097, 1098 ⟩ Used in section 1088.
⟨ vi mode source 1028, 1029, 1030, 1031, 1035, 1038, 1039, 1040, 1041, 1042, 1043, 1045, 1047, 1048, 1049, 1050, 1051, 1052, 1053, 1058, 1059, 1060, 1061, 1062, 1063, 1064, 1076, 1077, 1078, 1088, 1100, 1101, 1110, 1118, 1119, 1124, 1133, 1134, 1135, 1136, 1137, 1138, 1139, 1146, 1147 ⟩ Used in section 1021.
⟨ vi mode static functions 1023 ⟩ Used in section 1021.

⟨ vi mode static variables 1022, 1025, 1026, 1037, 1112 ⟩ Used in section 1021.
⟨ vi mode VARG1 state 1068 ⟩ Used in section 1063.
⟨ vi mode VARG2 state 1069 ⟩ Used in section 1063.
⟨ vi mode VCMD state 1074 ⟩ Used in section 1063.
⟨ vi mode VEXTCMD state 1071 ⟩ Used in section 1063.
⟨ vi mode VFAIL state 1073 ⟩ Used in section 1063.
⟨ vi mode VLIT state 1067 ⟩ Used in section 1063.
⟨ vi mode VREDO state 1075 ⟩ Used in section 1063.
⟨ vi mode VSEARCH state 1145 ⟩ Used in section 1063.
⟨ vi mode VXCH state 1072 ⟩ Used in section 1063.
⟨ vi VNORMAL command state 1066 ⟩ Used in section 1063.
⟨ vi VNORMAL insert or replace state 1065 ⟩ Used in section 1063.
⟨ vi.c 1021 ⟩
⟨ **return** a temporary read-only variable 118 ⟩ Used in section 116.
⟨ **return** a variable already in the environment 120 ⟩ Used in section 119.
⟨ **return** the value of non-alphabetic variables 145 ⟩ Used in section 119.
⟨ **return** the value of positional parameters 146 ⟩ Used in section 145.
⟨ **return** the value of punctuation variables 147 ⟩ Used in section 145.
⟨ **unwind** after an evaluation error 1171 ⟩ Used in section 1170.

(Mostly) Public Domain `ksh`

	Section	Page
(Mostly) Public Domain Korn Shell	1	1
Changes From OpenBSD 7.0	2	2
main.c	3	3
Shared Header & Utilities	7	5
Conditional Compilation	11	7
Internal Debugging	12	8
Version	13	9
Global Initialisation	15	10
Option Parsing	32	17
Initial Shell Environment	61	28
Memory	69	31
Hash Table	80	34
Iteration	97	40
Sorting	100	41
Variables & The Environment	103	42
Exporting variables	132	54
Arrays	136	56
Special Variables	145	59
String & Number Formatting	161	67
Initialisation	183	73
Miscellanea	195	76
Character Data (Strings)	204	79
Expandable String Macros	217	83
Scripts & Core Loop	227	86
Handling Errors	236	90
Input & Output	237	91
File Descriptors	255	99
Co-Process File Descriptors	262	101
Temporary Files	271	103
Buffers	274	104
Reading & Writing	282	108
Formatted Output	289	114
Error & Warning Messages	299	121
Display Routines	306	124
Source Input	313	127
Lexical Analysis	330	137
Lexing-Supporting Functions	368	157
Interactive Prompt	377	163
Compilation	412	172
Pipelines	417	176
I/O redirection	453	190
Supporting Functions	455	191

Execution ... 461 193
Timing Commands ... 467 197
Syntax Tree ... 469 200
External Commands ... 524 220
Functions & Built-in Commands ... 536 227
Expansion/Substitution ... 554 233
Alternation ... 606 257
Tilde Expansion ... 611 259
Pattern Matching ... 614 261
Subprocess I/O ... 623 266
Operating System ... 634 270
Signal/Trap Utilities ... 649 278
Alarm ... 656 280
Terminal (`tty`) Devices ... 661 281
Job Control ... 666 283
Subprocesses ... 689 291
`SIGCHLD` Handler ... 701 297
Signalling ... 707 300
Job List ... 718 306
Asynchronous Jobs ... 725 310
Job Control Utilities ... 728 312
Filesystem ... 741 318
Path Names ... 747 320
Globbing ... 766 327
Globbing Utilities ... 799 341
History ... 812 345
History Navigation ... 830 353
History File I/O ... 839 356
`fc`/`history` Built-in Command ... 844 358
Interactive Editor ... 858 363
Shared Routines ... 878 369
Emacs ... 882 371
Emacs Mode Key Bindings ... 897 381
Emacs Mode Functions ... 914 386
Insertion & Deletion ... 926 390
Navigation ... 936 393
Termination ... 946 396
Mutation ... 951 397
Drawing ... 961 400
Searching/History ... 974 404
Kill Ring & Region ... 991 409
File/command name completion routines ... 998 412
I/O ... 1008 415
Bits ... 1016 418
vi ... 1021 419
vi Edit Buffer ... 1036 426
vi Display ... 1044 428
vi Modes ... 1063 435
vi Commands ... 1077 440
vi Insert Mode ... 1079 441
vi Yank Buffer ... 1102 447
vi Word Expansion ... 1111 450

vi Cursor Navigation 1121 454
vi History Navigation 1140 460
Other vi Commands 1148 466
Mathematical Expressions 1158 469
Mail 1189 481
Built-in Commands 1199 485
Double-Bracket Tests (`[[ ... ]]`) 1327 548
Menu Selection 1337 552
Index 1344 554