Department of
Computer Science
&
Software Engineering

Student Manual

Volume B

2007 Edition

Acknowledgements

The following Staff have been the main contributors during the evolution of the Student Manual:

Sarah Ardu	Mark Bates	Brian Coogan	Jamie Curmi
Philip Dart	Gill Dobbie	Paul Dunn	Peter Eden
Wai-keong Foong	Rex Harris	Margaret Hassall	David Hornsby
Lorraine Johnston	Roy Johnston	Christopher Jones	Teresa Keddis
Kathleen Keogh	Michael Lawley	John Lenarcic	Rod Logan
Alistair Moffat	Ed Morris	Emma Norling	Virginia Norling
Michael Paddon	Heather Payne	Kathleen Keogh	Ian Richards
John Shepherd	Harald Søndergaard	Liz Sonenberg	James Thom
John Torriero	Jeroen van den Muyzenberg	Terence Wong	Justin Zobel

Thanks are also due to various people at the Universities of Sydney and New South Wales, and our local system gurus, Stewart Forster and Robert Elz.

Special thanks are also due to Thomas Mackey and Gray Watson, the original authors of the sections "RCS and Configuration Control" and "Concurrent Versions System (CVS)" respectively.

The 2002 Edition of Student Manual B was edited by Tim Gabrić. It was updated for the 2006 Edition by Joselito (Joey) Chua, with contributions from Mike Ciavarella and Zoltan Somogyi. It was slightly updated for the 2007 Edition by Lars Kulik.

If you have any suggestions for improving this manual, or find an error, send an e-mail to: `sm-editor@csse.unimelb.edu.au`.

For continuing updates and frequently-asked questions on this manual, visit: `http://www.csse.unimelb.edu.au/teaching/support/studentmanual/`

Contents

Principles of Responsible Student Behaviour

Students in the Department of Computer Science and Software Engineering have access to extensive computing facilities for the purpose of completing practical work associated with their course. These facilities include large and small systems, communications networks, and personal computers, as well as associated software, files, and data.

These facilities offer many opportunities for the sharing of information. With the ability to share facilities and information comes the responsibility to use the systems legally and to act in accordance with high standards of honesty and personal conduct. Every student in the Department of Computer Science and Software Engineering is expected to respect the rights of others and to use computing facilities appropriately.

Every student will sign a declaration form when applying for their computer account, and you should read this form carefully *before* signing it.

This document describes general principles of responsible student behaviour that have been adopted by the Department of Computer Science and Software Engineering. The University of Melbourne Computing Rules relating to the use of university computing and network facilities are listed in the Student Diary. Students found to be acting in breach of these rules and principles will be penalised.

Any questions concerning the interpretation of these principles should be directed to your lecturer or to the Head of the Department of Computer Science and Software Engineering.

Intended Use of Computing Facilities

Students are authorised to make use of computing facilities and resources provided by the University solely for academic purposes directly related to their course of study. As computing facilities and resources are limited, all students should be considerate of others.

- Before using any University computing facilities, e.g., computing equipment, printers, or software, it is the responsibility of each student to find out the conditions of use of the particular facilities they wish to use and to ensure that they have the required authorisation.
- In particular, students may use the Internet (ie news, mail, web browsers, etc), database, text processing, and other programs, provided it is for academic purposes as above and consistent with these principles.
- Computing facilities and resources provided for work in a particular subject should not in general be used for work in other subjects.

Privacy and Security

Students should take all reasonable precautions to protect their own accounts, files, disks, printouts, and other information from unauthorised use, and should also respect the individual privacy of other users.

- Students are responsible for their computer accounts, files, disks, and printouts. They should take all reasonable precautions to prevent unauthorised use of their accounts,

files, and disks by other users. Such precautions include care in the choice of passwords and in the setting of file access permissions. System-provided protection features can be used for this purpose.

- Students should not give authorisation to others to use their accounts, files, or disks unless specifically required to do so as part of their course.
- Students who believe their accounts, files, or disks have been compromised, i.e., accessed or able to be accessed by an unauthorised user, should notify the academic programmer or systems programmer immediately.
- Students should not attempt to access the account, files, or disk of another user without explicit authorisation from the other user, and then only if they are working on a joint project.
- Although students may sometimes be negligent or naive in revealing an account name or password to another person, this does not constitute authorisation for that other person to use the account.
- Normally, setting file access permissions to allow public or group access is considered authorisation for others to access those files. However, as many students do not understand these features, other students should not assume they have authorisation to access unprotected files, unless they have explicitly been given authorisation.
- Students should not use or distribute information not intended for use or distribution by its owner. This applies to passwords, files, listings, private notes, written assignments, and so on.
- Students are only permitted in University buildings during authorised opening hours.

Computer System Integrity

Students should not act intentionally to interfere with or to compromise the integrity of a computer system.

- Students should not attempt to restrict or deny access by authorised users to computing facilities. In particular, students should not tie up computer resources by game playing, listening to music or news streams, or other non-academic applications, by sending frivolous or excessive messages, or by using printing facilities inappropriately.
- Students should not use the account of another user, impersonate another user in communication, attempt to capture or crack passwords or encryption, destroy or alter data or programs belonging to other users, or subvert access restrictions such as quotas associated with users' accounts.

Intellectual Property Rights

Students must abide by any restrictions that are associated with the software and/or data that they use.

- Some software and data that reside on file systems that students may access are protected by copyright and other laws, and also by licenses and other contractual agreements. Students must not breach these restrictions.
- Generally, the purchaser of a computer program does not become the owner of the program, but is licensed to use it in accordance with the conditions of sale. Students must abide by any prohibitions that pertain to the use of any software that they are authorised to access. These may include restrictions against copying programs or data for use at another site, against use for non-educational purposes, and against disclosure of information (e.g., program source code).
- Students may not use University facilities for making or running unauthorised copies of software. There will be an automatic fine imposed by the Department for software piracy.
- Students undertake to keep confidential any disclosure to them by the University of software (including methods or concepts utilised therein) licensed to the University for use on its computers and they hereby indemnify and hold harmless the University against any claims of whatsoever nature arising out of any disclosure on their part to another of the said software in breach of this undertaking.

Duty of Care

The University and students have a responsibility to cooperate to provide a safe environment for teaching and learning. The University has a duty of care to students, and students have a duty of care towards each other and the academic and general staff to ensure everyone can learn in a safe environment. Action against students who act inappropriately will be taken in accordance with Statute 13 of the University on Student Discipline.

The Department of Computer Science and Software Engineering has a safety manual which can be viewed under "Safety" in the Department home page - see: `http://www.csse.unimelb.edu.au`

Any Safety questions or concerns about safety should be directed to the Department Officer and/or the Head of Department.

Collaboration and cheating

Students may discuss their course work with each other, and are encouraged to do so. However, students should only submit work that they have done themselves.

- Work submitted by a student should be the result of that student working substantially independently.
- Students should not use material (e.g., files, listings, hand-written notes) obtained from another student with or without their permission. Material obtained from textbooks or lecture notes may be used, provided appropriate attribution is given.
- Students should not provide another student with material that substantially assists that student in completing work he or she is expected to perform independently.

- Any student who is caught cheating will be penalised in accordance with the University discipline procedures.

Further information relating to the University's position with regard to plagiarism may be found at `http://www.services.unimelb.edu.au/plagiarism/`.

Disciplinary Action

University Penalties

The University is empowered to penalise students for improper behaviour, including withdrawal of access to University computing facilities, failure in an assignment or a subject, imposition of fines, and suspension or expulsion from the University.

The Department has decided that the minimum penalty imposed for software piracy will be a fine of $100.

Crimes (Computer) Act 1988

Under this Victorian Government legislation, Computer Trespass is a criminal offence and possible penalties include a term of imprisonment. Computer Trespass includes gaining access to, or entering, a computer system or part of a computer system without lawful authority to do so. Computer Trespass is an offence regardless of whether or not the trespasser gains, or intends to gain financial benefit, or alters or damages, or intends to alter or damage the files or programs.

Ramamohanarao Kotagiri
Head, Department of Computer Science and Software Engineering
February, 2003

Preface

This Manual is intended for use by students undertaking second year (or higher) courses held by the Department of Computer Science and Software Engineering. The reference material contained here should be considered a supplement to on-line references and text books.

Students new to the UNIX platform and/or undertaking first year course work should first consult **Student Manual A**.

Here are the sections of this Manual with brief descriptions:

- Unix Programming — Second Edition
  How to write programs which interface with the UNIX operating system.
- X Windows
  Basic information and hints to help you get more out of the X Windows GUI environment.
- Using GDB
  How to use the GNU Debugger to see what is going on "inside" another program while it executes — or what another program was doing at the moment it crashed.
- Make — A Program for Maintaining Computer Programs
  A description of the **make** command which allows the programmer to maintain, update, and regenerate groups of computer programs.
- RCS and Configuration Control
  An illustrated guide to using the Revision Control System (RCS) in conjunction with the **make** utility to manage a large software project.
- Concurrent Versions System (CVS)
  An introduction to another useful tool for managing software development projects.
- Introduction to Prolog
  Description of the PROLOG logic programming language.
- Lex — A Lexical Analyzer Generator
  Information on how to use the **lex** command to generate programs to be used in the simple lexical analysis of text.
- Yacc: Yet Another Compiler Compiler
  A description of the **yacc** command which converts a context-free grammar into a set of tables for a simple automaton that executes an LALR(1) parsing algorithm.

1 UNIX Programming — Second Edition

1.1 Introduction

This paper describes how to write programs that interface with the UNIX [1] operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document [1] collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [2] for Version 7 UNIX There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [3]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [4].

1.2 Basics

1.2.1 Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

```
main(argc, argv)       /* echo arguments */
int argc;
char *argv[];
{
        int i;

        for (i = 1; i < argc; i++)
              printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by `0` so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main` If you want to keep them around so other routines can get at them, you must copy them to external variables.

[1] UNIX is a trademark of A.T.&T. Bell Laboratories.

1.2.2 The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if `prog` uses `getchar`, then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog.`

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1` but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the "standard output," which is also by default the terminal. The output can be captured on a file by using `>`; if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog.`

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()  /* ccstrip: strip non-graphic characters */
```

```
{
        int c;
        while ((c = getchar()) != EOF)
            if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
                putchar(c);
        exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file */usr/include/stdio.h* of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

1.3 The Standard I/O Library

The "Standard I/O Library" is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

1.3.1 File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. (`FILE` is a type name, like `int`, not a structure tag).

The actual call to `fopen` in a program

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read (`"r"`, write (`"w"`), or append (`"a"`).

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which is defined as zero in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c`. `getc` and `putc` return `EOF` on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. `stdin`, `stdout` and `stderr` are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)        /* wc: count lines, words, chars */
int argc;
char *argv[];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc $>$ 2)
        printf("%7ld %7ld %7ld total", tlinect, twordct, tcharct);
    exit(0);
}
```

The function `fprintf` is identical to `printf`, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason

to call `fclose` on an output file — it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

1.3.2 Error Handling — Stderr and Exit

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. *wc* writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired.

1.3.3 Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with `putc`, etc., is buffered (except to `stderr`); to force it out immediately, use `fflush(fp)`.

`fscanf` is identical to `scanf`, except that its first argument is a file pointer (as with `fprintf`) that specifies the file from which the input comes; it returns `EOF` at end of file.

The functions `sscanf` and `sprintf` are identical to `fscanf` and `fprintf`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf` and into it for `sprintf`.

`fgets(buf,size, fp)` copies the next line from `fp`, up to and including a newline, into `buf`; at most `size-1` characters are copied; it returns `NULL` at end of file. `fputs(buf,fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c,fp)` "pushes back" the character `c` onto the input stream `fp`; a subsequent call to `getc`, `fscanf`, etc., will encounter `c`. Only one character of pushback per file is permitted.

1.4 Low-Level I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

1.4.1 File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of `READ(5,...)` and `WRITE(6,...)` in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with `<` and `>`, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

1.4.2 Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n`

bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define   BUFSIZE    512     /* best size for PDP-11 UNIX */

main()                       /* copy input to output      */
{
    char    buf[BUFSIZE];
    int     n;

    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
    exit(0);
}
```

If the file size is not a multiple of `BUFSIZE`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` which does unbuffered input.

```
#define    CMASK    0377    /* for making char's > 0 */

getchar()       /* unbuffered single character input */
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

`c` *must* be declared `char`, because `read` accepts a character pointer. The character being returned must be masked with `0377` to ensure that it is positive; otherwise sign extension may make it negative. (The constant `0377` is appropriate for the `PDP-11` but not necessarily for other machines.)

The second version of `getchar` does input in big chunks, and hands out the characters one at a time.

```
#define    CMASK    0377    /* for making char's > 0 */
#define    BUFSIZE  512

getchar()                   /* buffered version */
{
    static char    buf[BUFSIZE];
    static char    *bufp = buf;
    static int     n = 0;

    if (n == 0) {           /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

1.4.3 Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, `open` and `creat` [sic].

`open` is rather like the `fopen` discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`.

```
int fd;

fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to `open` a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and `-1` if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)                    /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
     int  f1, f2, n;
     char buf[BUFSIZE];

     if (argc != 3)
         error("Usage: cp from to", NULL);
     if ((f1 = open(argv[1], 0)) == -1)
         error("cp: can't open %s", argv[1]);
     if ((f2 = creat(argv[2], PMODE)) == -1)
         error("cp: can't create %s", argv[2]);

     while ((n = read(f1, buf, BUFSIZE)) > 0)
         if (write(f2, buf, n) != n)
             error("cp: write error", NULL);
     exit(0);
}


error(s1, s2)    /* print error message and die */
char *s1, *s2;
{
    printf(s1, s2);
    printf("");
    exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

1.4.4 Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary

order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long; fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0);    /* get to pos */
    return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP–11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

1.4.5 Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of –1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's*

Manual, so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `syserrno` is an array of character strings which can be indexed by `errno` and printed by your program.

1.4.6 Processes

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

1.4.7 The "System" Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
    system("date");
    /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember than `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

1.4.8 Low-Level Process Creation — Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `execl`. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with `execl`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like `<`, `>`, `*`, `?`, and `[]` in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
}
        execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, `/bin/sh`. Its argument `-c` says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in `commandline`.

1.4.9 Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl` or `execv`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork`:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the "process id." In one of these processes (the "child"), `proc_id` is zero. In the other (the "parent"), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);    /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execl` which does the `command` and then dies. In the parent, `fork` returns non-zero so it skips the `execl`. (If there is any error, `fork` returns `-1`).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait`:

```
int status;

if (fork() == 0)
    execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the `execl` or `fork`, or the possibility that there might be more than one child running simultaneously. (The `wait` returns the process id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in `status`). Still, these three lines are the heart of the standard library's `system` routine, which we'll show in a moment.

The `status` returned by `wait` encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork` nor the `exec` calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `execl`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

1.4.10 Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int fd[2];

stat = pipe(fd);
if (stat == -1)
    /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd,mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the the pipe with a `pipe` system call; it then `fork`s to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `execl`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define    READ    0
#define    WRITE   1
#define    tst(a, b) (mode == READ ? (b) : (a))
static     int     popen_pid;

popen(cmd, mode)
char       *cmd;
int        mode;
{
    int p[2];

    if (pipe(p) < 0)
        return(NULL);
```

```
        if ((popen_pid = fork()) == 0) {
            close(tst(p[WRITE], p[READ]));
            close(tst(0, 1));
            dup(tst(p[READ], p[WRITE]));
            close(tst(p[READ], p[WRITE]));
            execl("/bin/sh", "sh", "-c", cmd, 0);
            _exit(1);       /* disaster has occurred if we get here */
        }
        if (popen_pid == -1)
           return(NULL);
        close(tst(p[READ], p[WRITE]));
        return(tst(p[WRITE], p[READ]));
    }
```

The sequence of `close`s in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)     /* close pipe fd */
int fd;
{
    register r, (*hstat)(), (*istat)(), (*qstat)();
    int      status;
    extern int popen_pid;

    close(fd);
    istat = signal(SIGINT, SIG_IGN);
```

```
    qstat = signal(SIGQUIT, SIG_IGN);
    hstat = signal(SIGHUP, SIG_IGN);
    while ((r = wait(&status)) != popen_pid && r != -1);
    if (r == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    signal(SIGHUP, hstat);
    return(status);
}
```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

1.5 Signals — Interrupts and all that

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the `DEL` character is typed; *quit*, generated by the `FS` character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the

named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
    int onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process ... */

    exit(0);
}

onintr()
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to `signal`? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by `&`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>

jmp_buf        sjbuf;

main()
{
    int (*istat)(), onintr();
```

```
        istat = signal(SIGINT, SIG_IGN);       /* save original status */
        setjmp(sjbuf);  /* save current stack position */
        if (istat != SIG_IGN)
            signal(SIGINT, onintr);

        /* main processing loop */
    }

    onintr()
    {
        printf("Interrupt");
        longjmp(sjbuf); /* return to saved state */
    }
```

The include file `setjmp.h` declares the type `jmp_buf` an object in which the state can be saved. `sjbuf` is such an object; it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
    if (intflag)
        /* EOF caused by interrupt */
    else
        /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with

execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
    execl(...);

signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status);           /* until the child is done */
signal(SIGINT, onintr);  /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```
#include <signal.h>

system(s)       /* run command string s */
char *s;
{
    int status, pid, w;
    register int (*istat)(), (*qstat)();

    if ((pid = fork()) == 0) {
        execl("/bin/sh", "sh", "-c", s, 0);
        _exit(127);
    }

    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return(status);
}
```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1
```

References

[1] B. W. Kernighan and D. M. Ritchie, *UNIX Programming – Second Edition* Bell Laboratories, 1978.

[2] K. L. Thompson and D. M. Ritchie, *The* `UNIX` *Programmer's Manual,* Bell Laboratories, 1978.

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., 1978.

[4] B. W. Kernighan, "`UNIX` for Beginners — Second Edition." Bell Laboratories, 1978.

A The Standard I/O Library

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of `UNIX`.

A.1 General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

- **stdin**:
  The name of the standard input file
- **stdout**:
  The name of the standard output file
- **stderr**:
  The name of the standard error file

- **EOF**:
  is actually –1, and is the value returned by the read routines on end-of-file or error.
- **NULL**:
  is a notation for the null pointer, returned by pointer-valued functions to indicate an error
- **FILE**:
  expands to `struct _iob` and is a useful shorthand when declaring pointers to streams.
- **BUFSIZ**:
  is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.
- **getc, getchar, putc, putchar, feof, ferror, fileno**:
  are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

A.2 Calls

```
FILE *fopen(char *filename,char *type)
```

opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character). It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. The value returned is a file pointer. If it is `NULL` the attempt to open failed.

```
FILE *freopen(char *filename, char *type, FILE *ioptr)
```

The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen`. If the attempt to open fails, `NULL` is returned, otherwise `ioptr`, which will now refer to the new file. Often the reopened stream is `stdin` or `stdout`.

```
int getc(FILE *ioptr)
```

returns the next character from the stream named by `ioptr`, which is a pointer to a file such as returned by `fopen`, or the name `stdin`. The integer `EOF` is returned on end-of-file or when an error occurs. The null character `\0` is a legal character.

```
int fgetc(FILE *ioptr)
```

acts like `getc` but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

`putc(char c, FILE *ioptr)`

`putc` writes the character `c` on the output stream named by `ioptr`, which is a value returned from `fopen` or perhaps `stdout` or `stderr`. The character is returned as value, but `EOF` is returned on error.

`fputc(char c, FILE *ioptr)`

acts like `putc` but is a genuine function, not a macro.

`fclose(FILE *ioptr)`

The file corresponding to `ioptr` is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. `fclose` is automatic on normal termination of the program.

`fflush(FILE *ioptr)`

Any buffered information on the (output) stream named by `ioptr` is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, `stderr` always starts off unbuffered and remains so unless `setbuf` is used, or unless it is reopened.

`exit(errcode);`

terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls `fflush` for each output file. To terminate without flushing, use `_exit`.

`feof(FILE *ioptr)`

returns non-zero when end-of-file has occurred on the specified input stream.

`ferror(FILE *ioptr)`

returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

`getchar();`

is identical to `getc(stdin)`.

```
putchar(c);
```

is identical to `putc(c,`stdout).

```
char *fgets(char *s,FILE *ioptr)
```

reads up to `n-1` characters from the stream `ioptr` into the character pointer `s`. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. `fgets` returns the first argument, or `NULL` if error or end-of-file occurred.

```
fputs(char *s, FILE *ioptr)
```

writes the null-terminated string (character array) `s` on the stream `ioptr`. No newline is appended. No value is returned.

```
ungetc(char c, FILE *ioptr)
```

The argument character `c` is pushed back on the input stream named by `ioptr`. Only one character may be pushed back.

```
printf(char *format,a1, ...)
fprintf(FILE *ioptr, char *format, a1, ...)
sprintf(char *s, char *format, a1, ...)
```

`printf` writes on the standard output. `fprintf` writes on the named output stream. `sprintf` puts characters in the character array (string) named by `s`. The specifications are as described in section `printf`(3) of the `UNIX` *Programmer's Manual.*

```
scanf(char *format, a1, ...)
fscanf(FILE *ioptr, char *format, a1, ...)
sscanf(char *s, char *format, a1, ...)
```

`scanf` reads from the standard input. `fscanf` reads from the named input stream. `sscanf` reads from the character string supplied as `s`. `scanf` reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string `format`, and a set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored. `scanf` returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

`fread(ptr, sizeof( *ptr),nitems, FILE *ioptr)`

reads `nitems` of data beginning at `ptr` from file `ioptr`. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the `fopen` call.

`fwrite(ptr, sizeof( *ptr), nitems, FILE *ioptr)`

Like `fread`, but in the other direction.

`rewind(FILE *ioptr)`

rewinds the stream named by `ioptr`. It is not very useful except on input, since a rewound output file is still open only for output.

`system(char *string)`

The `string` is executed by the shell as if typed at the terminal.

`getw(FILE *ioptr)`

returns the next word from the input stream named by `ioptr`. `EOF` is returned on end-of-file or error, but since this a perfectly good integer `feof` and `ferror` should be used. A "word" is 16 bits on the `PDP-11.`

`putw(w, FILE *ioptr)`

writes the integer `w` on the named output stream.

`setbuf(FILE *ioptr, char *buf)`

`setbuf` may be used after a stream has been opened but before I/O has started. If `buf` is `NULL`, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

char buf[BUFSIZ];

`fileno(FILE *ioptr)`

returns the integer file descriptor associated with the file.

`fseek(ioptr,`offset,ptrname) FILE *ioptr; long offset;

The location of the next byte in the stream named by `ioptr` is adjusted. `offset` is a long integer. If `ptrname` is 0, the offset is measured from the beginning of the file; if `ptrname` is 1, the offset is measured from the current read or write pointer; if `ptrname` is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on `UNIX`non- systems, the offset must be a value returned from `ftell` and the ptrname must be 0).

```
long ftell(FILE *ioptr)
```

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On `UNIX`non- systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

```
getpw(uid, char *buf)
```

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

```
void *malloc(int num)
```

allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

```
void *calloc(int num)
```

allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available .

```
cfree(void *ptr)
```

Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including <`ctype.h`>.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable — a letter, digit, or punctuation character.

`iscntrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.

2 X Windows

2.1 Background

The X Window System runs on computers with bitmap displays (as opposed to 'character-based' displays, often referred to as 'dumb terminals').

It allows your programs to display output on your computer, whether your program is running on your computer (local) or on another computer (remotely).

2.2 X-Terminals

An X-Terminal is a specialised 'computer' that only runs one special program that allows you to connect directly to a remote machine. The remote machine is then completely responsible for 'managing' the X-Terminal. The X-Terminal itself is essentially 'invisible' and everything you do takes place on the remote machine.

2.3 First Login

When you first confront a Graphical Workstation or X-Terminal, you should see a box in the middle of the screen called the login prompt. To login, you need to type your Login name followed by your Password, at the prompt (the prompt is where the Cursor is blinking.)

Once you have logged in, you should see several windows. To type commands to a window, you first have to position the mouse pointer (by moving the mouse) inside that window. If the mouse pointer is not inside any windows, then it is said to be on the *root* or background window. The mouse pointer frequently changes shape (depending upon which window it is in), but it never disappears[2]. While on the root window, it looks like a bold cross.

2.4 Window Manager

The appearance and behaviour of the display is controlled by a program known as a *window manager*. The one configured for students use is called `fvwm2`.

The window manager gets the information it requires from a configuration file. If you wish to modify your setup, you should copy the default configuration file into your home directory with the following command:

```
cp /local/share/skel/common/fvwm2rc ~/.fvwm2rc
```

You can then modify it to your heart's content, preferably *after* reading the `fvwm2` manual pages. Be warned that incompatible changes will make things more difficult for the people trying to help you in lab classes.

2.5 Mouse Buttons and Menus

Pressing one of the mouse buttons while the mouse pointer is on the root window, gives access to different *pull-down* menus. When the cursor is on the root window, the mouse buttons

[2] Although there are programs that can make it disappear if you like

give you different pull–down menus.

- The menu on the right button gives you a listing of all the windows. If you highlight one of these and release the mouse button, the indicated window will be displayed in full (de-iconified if necessary and brought to the front) and your pointer will be positioned at the top-left corner of the window.
- The menu on the middle button provides some window manager functions. If the cursor changes shape but no other effect is noticeable, this means that the function operates on a window and you need to move the cursor to a window and press a mouse button before the operation will take effect. Try out some or all of these functions, but be careful of the *destroy* and *delete* operations!
- The menu on the left button allows you to create new windows. Choosing *new xterm* will create another xterm on the machine you are on. The choices named after machines create xterms on those machines; you will not be able to do this unless you have account on those machines.

  The *Quit* item in this menu is used to log-out of your X-session. This is arranged as a sub-menu just to make sure you wanted to log-out!

2.6 Xterm

An `xterm` is an application that emulates a dumb terminal. Many UNIX programs are character-based (ie. the UNIX shell) and typing commands into an xterm/shell is an effective way to run other programs.

In addition to simple character interaction, xterms provide other features above that of a simple dumb-terminal;

2.6.1 Cut & Paste

Xterm allows you to select text and copy it within the same or other windows.

The selection functions are invoked when the pointer buttons are used (with no modifiers - ie. no control or shift key)

Pointer button one (usually the left one) is used to highlight and save text. Move the cursor to the beginning of the text, and then hold the button down while moving the cursor to the end of the region and release the button. The selected text is highlighted and saved in the global cut-buffer. Double-clicking (ie. clicking the mouse button twice in quick succession) selects by words. Triple-clicking selects by lines.

Pointer button two (usually the middle one) 'types' (pastes) the text from the cut-buffer, inserting it as if it had been typed from the keyboard.

Pointer button three (usually right) extends the current selection. That is, you can change the bounds of a highlighted text selection by pressing and dragging the cursor while you re-position the start/end of the selection.

*warning - xterm will hang forever if you try to paste too much text at one time.

For more information, see the section: "POINTER USAGE", in the XTERM(1) manual page.

2.6.2 Menus

Xterm has three menus, named mainMenu, vtMenu and fontMenu. Each menu pops up under the correct combination of key and button presses.

The mainMenu pops up when the "control" key and pointer button one are pressed in a window. The mainMenu contains items that can be used to send *signals* to the shell.

The vtMenu sets various modes of the *emulation*, and is popped up when the "control" key and pointer button two are pressed in the window. Items here can be used to reset the terminal if you accidentally put it into, say, graphics mode (which can happen when you display a binary file)

The fontMenu can be used to set the font used in the window, and is accessed by pressing the "control" key and pointer button three.

2.7 X-Files

The following is a list of X-related files with a brief description of their purposes. For more information, you should try running "man" or "man -k" on the appropriate string.

$HOME/.fvwm2rc	user fvwm window manger configuration file
$HOME/.xrdb	X resource database file - a good place to put default settings for applications
$HOME/.xsession	commands to be run when logging on to a graphical display/workstation.
$HOME/.xsession-errors	guess!
$HOME/.xrsh.out	error output from xrsh(1) commands

There are also a number of general account configuration files in subdirectories of `/local/share/skel` which may also be of interest.

2.8 Copyright

3 Using GDB

3.1 Summary of GDB

The purpose of a debugger such as GDB [3] is to allow you to see what is going on "inside" another program while it executes—or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to debug programs written in C, C++, and Modula-2.

3.1.1 Free Software

GDB is *free software*, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms.

Fundamentally, the General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else.

For full details, see GNU GENERAL PUBLIC LICENSE in the Appendix.

3.2 Getting In and Out of GDB

Type **gdb** or **gdb** *program core* to start GDB and type **quit** or **C-d** (Control d) to exit.

3.2.1 Starting GDB

Start GDB with the shell command **gdb**. Once it's running, GDB reads commands from the terminal until you tell it to exit.

You can run **gdb** with no arguments or options; but the most usual way to start GDB is with one argument or two, specifying an executable program as the argument:

[3] Copyright ©1988, 1989, 1990, 1991, 1992 Free Software Foundation, Inc. see Appendix for conditions of copyright

gdb *program*

You can also start with both an executable program and a core file specified:

gdb *program core*

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

gdb *program* 1234

would attach GDB to process **1234** (unless you also have a file named '1234'; GDB does check for a core file first).

You can further control how GDB starts up by using command-line options. GDB itself can remind you of the options available.

Type

gdb -help

to display all available options and briefly describe their use ('**gdb -h**' is a shorter equivalent).

All options and command line arguments you give are processed in sequential order. The order makes a difference when the '**-x**' option is used.

3.2.2 Choosing Files

When GDB starts, it reads any arguments other than options as specifying an executable file and core file (or process ID). This is the same as if the arguments were specified by the '**-se**' and '**-c**' options respectively. (GDB reads the first argument that does not have an associated option flag as equivalent to the '**-se**' option followed by that argument; and the second argument that does not have an associated option flag, if any, as equivalent to the '**-c**' option followed by that argument.)

Many options have both long and short forms; both are shown in the following list. GDB also recognizes the long forms if you truncate them, so long as enough of the option is present to be unambiguous. (If you prefer, you can flag option arguments with '--' rather than '-', though we illustrate the more usual convention.)

-symbols=*file*
-s *file* Read symbol table from file *file*

-exec=*file*
-e *file* Use file *file* as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

-se = *file* Read symbol table from file *file* and use it as the executable file.

-core = *file*
-c *file* Use file *file* as a core dump to examine.

-command = *file*
-x *file* Execute GDB commands from file *file*.

-directory = *directory*
-d *directory* Add *directory* to the path to search for source files.

3.2.3 Choosing Modes

You can run GDB in various alternative modes—for example, in batch mode or quiet mode.

-nx
-n Do not execute commands from any '**.gdbinit**' initialization files. Normally, the commands in these files are executed after all the command options and arguments have been processed.

-quiet
-q "Quiet". Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

-batch Run in batch mode. Exit with status **0** after processing all the command files specified with '**-x**' (and '**.gdbinit**', if not inhibited). Exit with nonzero status if an error occurs in executing the GDB commands in the command files.
Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in order to make this more useful, the message

> Program exited normally.

(which is ordinarily issued whenever a program running under GDB control terminates) is not issued when running in batch mode.

-cd=*directory*
Run GDB using *directory* as its working directory, instead of the current directory.

-fullname
-f Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time your program stops). This recognizable format looks like two '`\032`' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two '`\032`' characters as a signal to display the source code for the frame.

-b *bps* Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

-tty=*device*
Run using *device* for your program's standard input and output.

3.2.4 Leaving GDB

quit
q To exit GDB, use the **quit** command (abbreviated **q**), or type an end-of-file character (usually **C-d**).

interrupt An interrupt (often **C-c**) will not exit from GDB, but rather will terminate the action of any GDB command that is in progress and return to GDB command level. It is safe to type the interrupt character at any time because GDB does not allow it to take effect until a time when it is safe.

If you have been using GDB to control an attached process or device, you can release it with the **detach** command.

3.2.5 Shell Commands

If you need to execute occasional shell commands during your debugging session, there is no need to leave or suspend GDB; you can just use the **shell** command.

shell ***command string***
Directs GDB to invoke an inferior shell to execute *command string*. If it exists, the environment variable **SHELL** is used for the name of the shell to run. Otherwise GDB uses `/bin/sh`.

The utility **make** is often needed in development environments. You do not have to use the **shell** command for this purpose in GDB:

make ***make-args***
Causes GDB to execute an inferior **make** program with the specified arguments. This is equivalent to '**shell make** ***make-args***'.

3.3 GDB Commands

You can abbreviate GDB command if that abbreviation is unambiguous; and you can repeat certain GDB commands by typing just **RET**.

3.3.1 Command Syntax

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command **step** accepts an argument which is the number of times to step, as in '**step 5**'. You can also use the **step** command with no arguments. Some command names do not allow any arguments.

GDB command names may always be truncated if that abbreviation is unambiguous. Other possible command abbreviations are listed in the documentation for individual commands. In some cases, even ambiguous abbreviations are allowed; for example, **s** is specially defined as equivalent to **step** even though there are other commands whose names start with **s**. You can test abbreviations by using them as arguments to the **help** command.

A blank line as input to GDB (typing just **RET**) means to repeat the previous command. Certain commands (for example, **run**) will not repeat this way; these are commands for which unintentional repetition might cause trouble and which you are unlikely to want to repeat.

The **list** and **x** commands, when you repeat them with **RET**, construct new arguments rather than repeating exactly as typed. This permits easy scanning of source or memory.

GDB can also use **RET** in another way: to partition lengthy output, in a way similar to the

common utility **more**. Since it is easy to press one **RET** too many in this situation, GDB disables command repetition after any command that generates this sort of display.

A line of input starting with # is a comment; it does nothing. This is useful mainly in command files.

3.3.2 Getting Help

You can always ask GDB itself for information on its commands, using the command **help**.

help
h

You can use **help** (abbreviated **h**) with no arguments to display a short list of named classes of commands:

```
(gdb) help
List of classes of commands:

running – Running the program
stack – Examining the stack
data – Examining data
breakpoints – Making program stop at certain points
files – Specifying and examining files
status – Status inquiries
support – Support facilities
user-defined – User-defined commands
aliases – Aliases of other commands
obscure – Obscure features

Type "help" followed by a class name for a list of
commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if
unambiguous.
(gdb)
```

help *class* Using one of the general help classes as an argument, you can get a list of the individual commands in that class. For example, here is the help display for the class **status**:

```
(gdb) help status
Status inquiries.
List of commands:
show – Generic command for showing things set with
    "set"
info – Generic command for printing status
Type "help" followed by command name for full docu-
    mentation.
Command name abbreviations are allowed if unambigu-
    ous.
(gdb)
```

help *command*
With a command name as **help** argument, GDB will display a short paragraph on how to use that command.

In addition to **help**, you can use the GDB commands **info** and **show** to inquire about the state of your program, or the state of GDB itself. Each command supports many topics of inquiry; this manual introduces each of them in the appropriate context. The listings under **info** and under **show** in the Index point to all the sub-commands.

info This command (abbreviated **i**) is for describing the state of your program; for example, it can list the arguments given to your program (**info args**), the registers currently in use (**info registers**), or the breakpoints you have set (**info breakpoints**). You can get a complete list of the **info** sub-commands with **help info**.

show In contrast, **show** is for describing the state of GDB itself. You can change most of the things you can **show**, by using the related command **set**; for example, you can control what number system is used for displays with **set radix**, or simply inquire which is currently in use with **show radix**.
To display all the settable parameters and their current values, you can use **show** with no arguments; you may also use **info set**. Both commands produce the same display.

Here are three miscellaneous **show** subcommands, all of which are exceptional in lacking corresponding **set** commands:

show version
Show what version of GDB is running. You should include this information in GDB bug-reports. If multiple versions of GDB are in use at your site, you may occasionally want to make sure what version of GDB you are running; as GDB evolves, new commands are introduced, and old ones may wither away. The version number is also announced when you start GDB with no arguments.

show copying
Display information about permission for copying GDB.

show warranty
Display the GNU "NO WARRANTY" statement.

3.4 Running Programs Under GDB

When you run a program under GDB, you must first generate debugging information when you compile it. You may start it with its arguments, if any, in an environment of your choice. You may redirect your program's input and output, debug an already running process, or kill a child process.

3.4.1 Compiling for Debugging

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the '**-g**' option when you run the compiler.

Many C compilers are unable to handle the '**-g**' and '**-O**' options together. Using those compilers, you cannot generate optimized executables containing debugging information.

The GNU C compiler supports '**-g**' with or without '**-O**', making it possible to debug optimized code. We recommend that you *always* use '**-g**' whenever you compile a program. You may think your program is correct, but there is no sense in pushing your luck.

When you debug a program compiled with '-g -O', remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file! An extreme example: if you define a variable, but never use it, GDB never sees that variable–because the compiler optimizes it out of existence.

Some things do not work as well with '**-g -O**' as with just '**-g**', particularly on machines with instruction scheduling. If in doubt, recompile with '**-g**' alone, and if this fixes the problem, please report it as a bug (including a test case!).

Older versions of the GNU C compiler permitted a variant option '**-gg**' for debugging information. GDB no longer supports this format; if your GNU C compiler has this option, do not use it.

3.4.2 Starting your Program

run
r
Use the **run** command to start your program under GDB. You must first specify the program name (except on VxWorks) with an argument to GDB, or by using the **file** or **exec-file** command.

If you are running your program in an execution environment that supports processes, **run** creates an inferior process and makes that process run your program. (In environments without processes, **run** jumps to the start of your program.)

The execution of a program is affected by certain information it receives from its superior.

GDB provides ways to specify this information, which you must do *before* starting your program. (You can change it after starting your program, but such changes will only affect your program the next time you start it.) This information may be divided into four categories:

The *arguments.*
: Specify the arguments to give your program as the arguments of the **run** command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments. In Unix systems, you can control which shell is used with the **SHELL** environment variable.

The *environment.*
: Your program normally inherits its environment from GDB, but you can use the GDB commands **set environment** and **unset environment** to change parts of the environment that will be given to your program.

The *working directory.*
: Your program inherits its working directory from GDB. You can set GDB's working directory with the **cd** command in GDB.

The *standard input and output.*
: Your program normally uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the **run** command line, or you can use the **tty** command to set a different device for your program.
Warning: While input and output redirection work, you cannot use pipes to pass the output of the program you are debugging to another program; if you attempt this, GDB is likely to wind up debugging the wrong program.

When you issue the **run** command, your program begins to execute immediately. Once your program has been started by the **run** command (and then stopped), you may evaluate expressions that involve calls to functions in your program, using the **print** or **call** commands.

If the modification time of your symbol file has changed since the uast time GDB read its symbols, GDB will discard its symbol table and re-read it. When it does this, GDB tries to retain your current breakpoints.

3.4.3 Your Program's Arguments

The arguments to your program can be specified by the arguments of the **run** command. They are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to your program. GDB uses the shell indicated by your environment variable **SHELL** if it exists; otherwise, GDB uses **/bin/sh**.

run with no arguments uses the same arguments used by the previous **run**, or those set by the **set args** command.

set args
: Specify the arguments to be used the next time your program is run. If **set args** has no arguments, **run** will execute your program with no arguments. Once you have run your program with arguments, using **set args** before the next **run** is the only way to run it again without arguments.

show args Show the arguments to give your program when it is started.

3.4.4 Your Program's Environment

The *environment* consists of a set of environment variables and their values. Environment variables conventionally record such things as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they are inherited by all the other programs you run. When debugging, it can be useful to try running your program with a modified environment without having to start GDB over again.

path *directory*
Add *directory* to the front of the **PATH** environment variable (the search path for executables), for both GDB and your program. You may specify several directory names, separated by '**:**' or whitespace. If *directory* is already in the path, it is moved to the front, so it will be searched sooner.
You can use the string '**$cwd**' to refer to whatever is the current working directory at the time GDB searches the path. If you use '.' instead, it refers to the directory where you executed the **path** command. GDB fills in the current path where needed in the *directory* argument, before adding it to the search path.

show paths
Display the list of search paths for executables (the **PATH** environment variable).

show environment [*varname*]
Print the value of environment variable @varvarname to be given to your program when it starts. If you do not supply @varvarname, print the names and values of all environment variables to be given to your program. You can abbreviate **environment** as **env**.

set environment *varname* [=] *value*
Sets environment variable *varname* to *value*. The value changes for your program only, not for GDB itself. *value* may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. The *value* parameter is optional; if it is eliminated, the variable is set to a null value.
For example, this command:

set env USER = foo

tells a Unix program, when subsequently run, that its user is named '**foo**'. (The spaces around '**=**' are used for clarity here; they are not actually required.)

unset environment*varname*
Remove variable *varname* from the environment to be passed to your program. This is different from '**set env** *varname* ='; **unset environment** removes the variable from the environment, rather than assigning it an empty value.

3.4.5 Your Program's Working Directory

Each time you start your program with **run**, it inherits its working directory from the current working directory of GDB. The GDB working directory is initially whatever it inherited from its parent process (typically the shell), but you can specify a new working directory in GDB with the **cd** command.

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on.

cd *directory*
: Set GDB's working directory to *directory*.

pwd
: Print GDB's working directory.

3.4.6 Your Program's Input and Output

By default, the program you run under GDB does input and output to the same terminal that GDB uses. GDB switches the terminal to its own terminal modes to interact with you, but it records the terminal modes your program was using and switches back to them when you continue running your program.

info terminal
: Displays GDB's recorded information about the terminal modes your program is using.

You can redirect your program's input and/or output using shell redirection with the **run** command. For example,

```
run > outfile
```

starts your program, diverting its output to the file '**outfile**'.

Another way to specify where your program should do input and output is with the **tty** command. This command accepts a file name as argument, and causes this file to be the default for future **run** commands. It also resets the controlling terminal for the child process, for future **run** commands. For example,

```
tty /dev/ttyb
```

directs that processes started with subsequent **run** commands default to do input and output on the terminal **/dev/ttyb** and have that as their controlling terminal.

An explicit redirection in **run** overrides the **tty** command's effect on the input/output device, but not its effect on the controlling terminal.

When you use the **tty** command or redirect input in the **run** command, only the input *for your program* is affected. The input for GDB still comes from your terminal.

3.4.7 Debugging an Already-Running Process

attach *process-id*
: This command attaches to a running process—one that was started outside GDB. (**info files** will show your active targets.) The command takes as argument a process ID. The usual way to find out the process-id of a Unix process is with the **ps** utility, or with the '**jobs -l**' shell command.
attach will not repeat if you press **RET** a second time after executing the command.

To use **attach**, you must be debugging in an environment which supports processes; for example, 'attach' does not work for programs on bare-board targets that lack an operating system. You must also have permission to send the process a signal.

When using **attach**, you should first use the **file** command to specify the program running in the process and load its symbol table.

The first thing GDB does after arranging to debug the specified process is to stop it. You can examine and modify an attached process with all the GDB commands that are ordinarily available when you start processes with **run**. You can insert breakpoints; you can step and continue; you can modify storage. If you would rather the process continue running, you may use the **continue** command after attaching GDB to the process.

detach
: When you have finished debugging the attached process, you can use the **detach** command to release it from GDB's control. Detaching the process continues its execution. After the **detach** command, that process and GDB become completely independent once more, and you are ready to **attach** another process or start one with **run**. **detach** will not repeat if you press **RET** again after executing the command.

If you exit GDB or use the **run** command while you have an attached process, you kill that process. By default, you will be asked for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the **set confirm** command.

3.4.8 Killing the Child Process

kill
: Kill the child process in which your program is running under GDB.

This command is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump file while your program is running.

On some operating systems, a program cannot be executed outside GDB while you have breakpoints set on it inside GDB. You can use the **kill** command in this situation to permit running your program outside the debugger.

The **kill** command is also useful if you wish to recompile and relink your program, since on many systems it is impossible to modify an executable file while it is running in a process. In this case, when you next type **run**, GDB will notice that the file has changed, and will re-read the symbol table (while trying to preserve your current breakpoint settings).

3.5 Stopping and Continuing

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and find out why.

Inside GDB, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as **step**. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. Usually, the messages shown by GDB provide ample explanation of the status of your program—but you can also explicitly request this information at any time.

info program
: Display information about the status of your program: whether it is running or not, what process it is, and why it stopped.

3.5.1 Breakpoints, Watchpoints, and Exceptions

A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add various conditions to control in finer detail whether your program will stop. You can set breakpoints with the **break** command and its variants, to specify the place where your program should stop by line number, function name or exact address in the program. In languages with exception handling (such as GNU C++), you can also set breakpoints where an exception is raised

A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints, but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

Each breakpoint or watchpoint is assigned a number when it is created; these numbers are successive integers starting with one. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be *enabled* or *disabled*; if disabled, it has no effect on your program until you enable it again.

Breakpoints are set with the **break** command (abbreviated **b**).

You have several ways to say where the breakpoint should go.

break *function*
: Set a breakpoint at entry to function *function*. When using source languages that permit overloading of symbols, such as C++, *function* may refer to more than one possible place to break.

break +*offset*
break -*offset*
: Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected frame.

break *linenum*
: Set a breakpoint at line *linenum* in the current source file. That file is the last file whose source text was printed. This breakpoint will stop your program just before it executes any of the code on that line.

break *filename*:*linenum*
: Set a breakpoint at line *linenum* in source file *filename*.

break *filename*:*function*
: Set a breakpoint at entry to function *function* found in file *filename*. Specifying a file name as well as a function name is superfluous except when multiple files contain similarly named functions.

break **address*
: Set a breakpoint at address *address*. You can use this to set breakpoints in parts of your program which do not have debugging information or source files.

break
: When called without any arguments, **break** sets a breakpoint at the next instruction to be executed in the selected stack frame. In any selected frame but the innermost, this will cause your program to stop as soon as control returns to that frame. This is similar to the effect of a **finish** command in the frame inside the selected frame—except that **finish** does not leave an active breakpoint. If you use **break** without an argument in the innermost frame, GDB will stop the next time it reaches the current location; this may be useful inside loops.
GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when your program stopped.

break ... **if** *cond*
: Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond* evaluates as true. '...' stands for one of the possible arguments described above (or no argument) specifying where to break.

tbreak *args*
: Set a breakpoint enabled only for one stop. *args* are the same as for the **break** command, and the breakpoint is set in the same way, but the breakpoint is automatically disabled after the first time your program stops there.

rbreak *regex*
: Set breakpoints on all functions matching the regular expression *regex*. This command sets an unconditional breakpoint on all matches, printing a list of all breakpoints it set. Once these breakpoints are set, they are treated just like the breakpoints set with the **break** command. They can be deleted, disabled, made conditional, etc., in the standard ways.
When debugging C++ programs, **rbreak** is useful for setting breakpoints on overloaded functions that are not members of any special classes.

info breakpoints [*n*]
info break [***n***]
info watchpoints [***n***]
: Print a table of all breakpoints and watchpoints set and not deleted, with the following columns for each breakpoint:
Breakpoint Numbers
Type Breakpoint or watchpoint.
Disposition Whether the breakpoint is marked to be disabled or deleted when hit.
Enabled or Disabled Enabled breakpoints are marked with 'y'. 'n' marks breakpoints that are not enabled.
Address Where the breakpoint is in your program, as a memory address
What Where the breakpoint is in the source for your program, as a file and line number. If a breakpoint is conditional, *info break* shows the condition on the line following the affected breakpoint; breakpoint commands, if any, are listed after that.
info break with a breakpoint number N as argument lists only that breakpoint. The convenience variable *$_* and the default examining-address for the 'x' command are set to the address of the last breakpoint listed.

GDB allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful.

GDB itself sometimes sets breakpoints in your program for special purposes, such as proper handling of **longjmp** (in C programs). These internal breakpoints are assigned negative numbers, starting with '-1'; **info breakpoints** does not display them.

You can see these breakpoints with the GDB maintenance command **maint info breakpoints**.

maint info breakpoints
: Using the same format as **info breakpoints**, display both the breakpoints you've set explicitly, and those GDB is using for internal purposes. Internal breakpoints are shown with negative breakpoint numbers. The type column identifies what kind of breakpoint is shown:

breakpoint
: Normal, explicitly set breakpoint.

watchpoint
: Normal, explicitly set watchpoint.

longjmp
: Internal breakpoint, used to handle correctly stepping through **longjmp** calls.

longjmp resume
: Internal breakpoint at the target of a **longjmp**.

until
: Temporary internal breakpoint used by the GDB **until** command.

finish
: Temporary internal breakpoint used by the GDB **finish** command.

3.5.2 Setting Watchpoints

You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.

Watchpoints currently execute two orders of magnitude more slowly than other breakpoints, but this can well be worth it to catch errors where you have no clue what part of your program is the culprit.

watch *expr*
: Set a watchpoint for an expression.

info watchpoints
: This command prints a list of watchpoints; it is otherwise similar to **info break**.

3.5.3 Breakpoints and Exceptions

Some languages, such as GNU C++, implement exception handling. You can use GDB to examine what caused your program to raise an exception, and to list the exceptions your program is prepared to handle at a given point in time.

catch *exceptions*
: You can set breakpoints at active exception handlers by using the **catch** command. *exceptions* is a list of names of exceptions to catch.

You can use **info catch** to list active exception handlers.

There are currently some limitations to exception handling in GDB. These will be corrected in a future release.

- If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits.
- You cannot raise an exception interactively.
- You cannot interactively install an exception handler.

Sometimes **catch** is not the best way to debug exception handling: if you need to know exactly where an exception is raised, it is better to stop *before* the exception handler is called, since that way you can see the stack before any unwinding takes place. If you set a breakpoint in an exception handler instead, it may not be easy to find out where the exception was raised.

To stop just before an exception handler is called, you need some knowledge of the implementation. In the case of GNU C++, exceptions are raised by calling a library function named **__raises_exception** which has the following ANSI C interface:

```
/* addr is where the exception identifier is stored.
   ID is the exception identifier.  */
void __raise_exception (void **addr, void *id);
```

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on **__raise_exception**

With a conditional breakpoint that depends on the value of *id*, you can stop your program when a specific exception is raised. You can use multiple conditional breakpoints to stop your program when any of a number of exceptions are raised.

3.5.4 Deleting Breakpoints

It is often necessary to eliminate a breakpoint or watchpoint once it has done its job and you no longer want your program to stop there. This is called *deleting* the breakpoint. A breakpoint that has been deleted no longer exists; it is forgotten.

With the **clear** command you can delete breakpoints according to where they are in your program. With the **delete** command you can delete individual breakpoints or watchpoints by specifying their breakpoint numbers.

It is not necessary to delete a breakpoint to proceed past it. GDB automatically ignores breakpoints on the first instruction to be executed when you continue execution without changing the execution address.

clear
: Delete any breakpoints at the next instruction to be executed in the selected stack frame. When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

clear *function*
clear *filename*:*function*
: Delete any breakpoints set at entry to the function *function*.

clear *linenum*
clear *filename*:*linenum*
: Delete any breakpoints set at or within the code of the specified line.

delete [**breakpoints**] [*bnums*...]
: Delete the breakpoints or watchpoints of the numbers specified as arguments. If no argument is specified, delete all breakpoints (GDB asks confirmation, unless you have **set confirm off**). You can abbreviate this command as **d**.

3.5.5 Disabling Breakpoints

Rather than deleting a breakpoint or watchpoint, you might prefer to *disable* it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can *enable* it again later.

You disable and enable breakpoints and watchpoints with the **enable** and **disable** commands, optionally specifying one or more breakpoint numbers as arguments. Use **info break** or **info watch** to print a list of breakpoints or watchpoints if you do not know which numbers to use.

A breakpoint or watchpoint can have any of four different states of enablement:

- Enabled. The breakpoint will stop your program. A breakpoint set with the **break** command starts out in this state.
- Disabled. The breakpoint has no effect on your program.
- Enabled once. The breakpoint will stop your program, but when it does so it will become disabled. A breakpoint set with the **tbreak** command starts out in this state.
- Enabled for deletion. The breakpoint will stop your program, but immediately after it does so it will be deleted permanently.

You can use the following commands to enable or disable breakpoints and watchpoints:

disable [**breakpoints**] [*bnums...*]
: Disable the specified breakpoints—or all breakpoints, if none are listed. A disabled breakpoint has no effect but is not forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later. You may abbreviate **disable** as **dis**.

enable [**breakpoints**] ***[bnums ...]***
: Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

enable [**breakpoints**] **once** ***bnums...***
: Enable the specified breakpoints temporarily. Each will be disabled again the next time it stops your program.

enable [**breakpoints**] **delete** *bnums...*
: Enable the specified breakpoints to work once and then die. Each of the breakpoints will be deleted the next time it stops your program.

Save for a breakpoint set with **tbreak** breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command **until** can set and delete a breakpoint of its own, but it will not change the state of your other breakpoints.

3.5.6 Break Conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Boolean expression in your programming language. A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is *true*.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated—that is, when the condition is false. In C, if you want to test an assertion expressed by the condition *assert*, you should set the condition '! *assert*' on the appropriate breakpoint.

Conditions are also accepted for watchpoints; you may not need them, since a watchpoint is inspecting the value of an expression anyhow—but it might be simpler, say, to just set a

watchpoint on a variable name, and specify a condition that tests whether the new value is an interesting one.

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached.

Break conditions can be specified when a breakpoint is set, by using 'if' in the arguments to the **break** command. They can also be changed at any time with the **condition** command. The **watch** command does not recognize the **if** keyword; **condition** is the only way to impose a further condition on a watchpoint.

condition *bnum expression*
: Specify *expression* as the break condition for breakpoint or watchpoint number *bnum*. From now on, this breakpoint will stop your program only if the value of *expression* is true (nonzero, in C). When you use **condition**, GDB checks *expression* immediately for syntactic correctness, and to determine whether symbols in it have referents in the context of your breakpoint. GDB does not actually evaluate *expression* at the time the **condition** command is given, however.

condition *bnum*
: Remove the condition from breakpoint number *bnum*. It becomes an ordinary unconditional breakpoint.

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. This is so useful that there is a special way to do it, using the *ignore count* of the breakpoint. Every breakpoint has an ignore count, which is an integer. Most of the time, the ignore count is zero, and therefore has no effect. But if your program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues. As a result, if the ignore count value is n, the breakpoint will not stop the next n times it is reached.

ignore *bnum count*
: Set the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, your program's execution will not stop; other than to decrement the ignore count, GDB takes no action.
To make the breakpoint stop the next time it is reached, specify a count of zero.

continue *count*
c *count*
fg *count*

Continue execution of your program, setting the ignore count of the breakpoint where your program stopped to *count* minus one. Thus, your program will not stop at this breakpoint until the *count*'th time it is reached.
An argument to this command is meaningful only when your program stopped due to a breakpoint. At other times, the argument to **continue** is ignored.
The synonym **fg** is provided purely for convenience, and has exactly the same behavior as other forms of the command.

If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, the condition will be checked.

You could achieve the effect of the ignore count with a condition such as '$foo-- <= 0' using a debugger convenience variable that is decremented each time.

3.5.7 Breakpoint Command Lists

You can give any breakpoint (or watchpoint) a series of commands to execute when your program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.

commands [***bnum***]
... ***command-list*** ...
end

Specify a list of commands for breakpoint number *bnum*. The commands themselves appear on the following lines. Type a line containing just **end** to terminate the commands.
To remove all commands from a breakpoint, type **commands** and follow it immediately with **end**; that is, give no commands.
With no *bnum* argument, **commands** refers to the last breakpoint or watchpoint set (not to the breakpoint most recently encountered).

Pressing **RET** as a means of repeating the last GDB command is disabled within a *command-list*.

You can use breakpoint commands to start your program up again. Simply use the **continue** command, or **step**, or any other command that resumes execution. Subsequent commands in the command list are ignored.

If the first command specified is **silent**, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If the remaining commands too print nothing, you will see no sign that the breakpoint was reached at all. **silent** is meaningful only at the beginning of a breakpoint command list.

The commands **echo** and **output** that allow you to print precisely controlled output are often useful in silent breakpoints.

For example, here is how you could use breakpoint commands to print the value of **x** at entry to **foo** whenever **x** is positive.

```
break foo if x>0
commands
silent
printf "x is %d\n",x
cont
end
```

One application for breakpoint commands is to compensate for one bug so you can test for another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the **continue** command so that your program does not stop, and start with the **silent** command so that no output is produced. Here is an example:

```
break 403
commands
silent
set x = y + 4
cont
end
```

3.5.8 Breakpoint Menus

Some programming languages (notably C++) permit a single function name to be defined several times, for application in different contexts. This is called *overloading*. When a function name is overloaded, '**break** *function*' is not enough to tell GDB where you want a breakpoint. GDB offers you a menu of numbered choices for different possible breakpoints, and waits for your selection with the prompt >. The first two options are always '**[0] cancel**' and '**[1] all**'. Typing **1** sets a breakpoint at each definition of *function*, and typing **0** aborts the **break** command without setting any new breakpoints.

For example, the following session excerpt shows an attempt to set a breakpoint at the overloaded symbol **String::after**. We choose three particular definitions of that function name:

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
```

Multiple breakpoints were set.
Use the "delete" command to delete unwanted breakpoints.
(gdb)

3.5.9 "Cannot Insert Breakpoints"

Under some operating systems, breakpoints cannot be used in a program if any other process is running that program. in this situation, attempting to run or continue a program with a breakpoint causes GDB to stop the other process.

When this happens, you have three ways to proceed:

1. Remove or disable the breakpoints, then continue.

2. Suspend GDB, and copy the file containing your program to a new name. Resume GDB and use the **exec-file** command to specify that GDB should run your program under that name. Then start your program again.

3. Relink your program so that the text segment is nonsharable, using the linker option **-N**. The operating system limitation may not apply to nonsharable executables.

3.6 Continuing and Stepping

Continuing means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or to a signal. (If due to a signal, you may want to use **handle**, or use '**signal 0**' to resume execution.

continue [***ignore-count***]
c [***ignore-count***]
fg [***ignore-count***]

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *ignore-count* allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of **ignore**.

The argument *ignore-count* is meaningful only when your program stopped due to a breakpoint. At other times, the argument to **continue** is ignored.

The synonyms 'c' and 'fg' are provided purely for convenience, and have exactly the same behavior as 'continue'.

To resume execution at a different place, you can use **return** to go back to the calling function; or **jump** to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

step Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated **s**.

> *Warning:* If you use the **step** command while control is within a function that was compiled without debugging information, execution will proceed until control reaches another function.

step *count*

Continue running as in **step**, but do so *count* times. If a breakpoint is reached or a signal not related to stepping occurs before *count* steps, stepping stops right away.

next [***count***]

Continue to the next source line in the current (innermost) stack frame. Similar to **step**, but any function calls appearing within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the stack level which was executing when the **next** command was given. This command is abbreviated **n**.
An argument *count* is a repeat count, as for **step**.
next within a function that lacks debugging information acts like **step**, but any function calls appearing within the code of the function are executed without stopping.

finish Continue running until just after function in the selected stack frame returns. Print the returned value (if any).
Contrast this with the **return** command.

until
u Continue running until a source line past the current line, in the current stack frame, is reached. This command is used to avoid single stepping through a loop more than once. It is like the **next** command, except that when **until** encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump.
This means that when you reach the end of a loop after single stepping though it, **until** will cause your program to continue execution until the loop is exited. In contrast, a **next** command at the end of a loop will simply step back to the beginning of the loop, which would force you to step through the next iteration.
until always stops your program if it attempts to exit the current stack frame.

until may produce somewhat counterintuitive results if the order of machine code does not match the order of the source lines. For example, in the following excerpt from a debugging session, the **f** (**frame**) command shows that execution is stopped at line **206**; yet when we use **until**, we get to line **195**:

```
(gdb) f
#0 main (argc=4, argv=0xf7fffae8) at m4.c:206
206 expand_input();
(gdb) until
195 for ( ; argc > 0; NEXTARG) {
```

This happened because, for execution efficiency, the compiler had generated code for the loop closure test at the end, rather than the start, of the loop—even though the test in a C **for**-loop is written before the body of the loop. The **until** command appeared to step back to the beginning of the loop when it advanced to this expression; however, it has not really gone to an earlier statement—not in terms of the actual machine code.
until with no argument works by means of single instruction stepping, and hence is slower than **until** with an argument.

until *location*
u ***location*** Continue running your program until either the specified location is reached, or the current stack frame returns. *location* is any of the forms of argument acceptable to **break**. This form of the command uses breakpoints, and hence is quicker than **until** without an argument.

stepi
si Execute one machine instruction, then stop and return to the debugger.
It is often useful to do '**display/i $pc**' when stepping by machine instructions. This will cause the next instruction to be executed to be displayed automatically at each stop.
An argument is a repeat count, as in **step**.

nexti
ni Execute one machine instruction, but if it is a function call, proceed until the function returns.
An argument is a repeat count, as in **next**.

3.7 Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, in Unix **SIGINT** is the signal a program gets when you type an interrupt (often **C-c**); **SIGSEGV** is the signal a program gets from referencing a place in memory far away from all the areas in use; **SIGALRM** occurs when the alarm clock timer goes off (which happens only if your program has requested an alarm).

Some signals, including **SIGALRM**, are a normal part of the functioning of your program. Others, such as **SIGSEGV**, indicate errors; these signals are *fatal* (kill your program immediately) if the program has not specified in advance some other way to handle the signal. **SIGINT** does not indicate an error in your program, but it is normally fatal so it can carry out the purpose of the interrupt: to kill the program.

GDB has the ability to detect any occurrence of a signal in your program. You can tell GDB

in advance what to do for each kind of signal.

Normally, GDB is set up to ignore non-erroneous signals like **SIGALRM** (so as not to interfere with their role in the functioning of your program) but to stop your program immediately whenever an error signal happens. You can change these settings with the **handle** command.

info signals
: Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals.

handle *signal keywords* ...
: Change the way GDB handles signal *signal*. *signal* can be the number of a signal or its name (with or without the '**SIG**' at the beginning). The *keywords* say what change to make.

The keywords allowed by the **handle** command can be abbreviated. Their full names are:

nostop
: GDB should not stop your program when this signal happens. It may still print a message telling you that the signal has come in.

stop
: GDB should stop your program when this signal happens. This implies the **print** keyword as well.

print
: GDB should print a message when this signal happens.

noprint
: GDB should not mention the occurrence of the signal at all. This implies the **nostop** keyword as well.

pass
: GDB should allow your program to see this signal; your program will be able to handle the signal, or may be terminated if the signal is fatal and not handled.

nopass
: GDB should not allow your program to see this signal.

When a signal has been set to stop your program, your program cannot see the signal until you continue. It will see the signal then, if **pass** is in effect for the signal in question *at that time*. In other words, after GDB reports a signal, you can use the **handle** command with **pass** or **nopass** to control whether that signal will be seen by your program when you later continue it.

You can also use the **signal** command to prevent your program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with '**signal 0**'.

3.8 Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, the information about where in your program the call was made from is saved in a block of data called a *stack frame*. The frame also

contains the arguments of the call and the local variables of the function that was called. All the stack frames are allocated in a region of memory called the *call stack*.

When your program stops, the GDB commands for examining the stack allow you to see all of this information.

One of the stack frames is *selected* by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. There are special GDB commands to select whichever frame you are interested in.

When your program stops, GDB automatically selects the currently executing frame and describes it briefly as the **frame** command does

3.8.1 Stack Frames

The call stack is divided up into contiguous pieces called *stack frames*, or *frames* for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function **main**. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing one of those bytes whose address serves as the address of the frame. Usually this address is kept in a register called the *frame pointer register* while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by GDB to give you a way of designating stack frames in GDB commands.

Some compilers allow functions to be compiled so that they operate without stack frames. (For example, the **gcc** option '**-fomit-frame-pointer**' will generate functions without a frame.) This is occasionally done with heavily used library functions to save the frame setup time. GDB has limited facilities for dealing with these function invocations. If the innermost function invocation has no stack frame, GDB will nevertheless regard it as though it had a separate frame, which is numbered zero as usual, allowing correct tracing of the function call chain. However, GDB has no provision for frameless functions elsewhere in the stack.

3.8.2 Backtraces

A backtrace is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

backtrace
bt Print a backtrace of the entire stack: one line per frame for all frames in the stack.
You can stop the backtrace at any time by typing the system interrupt character, normally **C-c**.

backtrace *n*
bt *n* Similar, but print only the innermost *n* frames.

backtrace -*n*
bt -*n* Similar, but print only the outermost *n* frames.

The names **where** and **info stack** (abbreviated **info s**) are additional aliases for **backtrace**.

Each line in the backtrace shows the frame number and the function name. The program counter value is also shown—unless you use **set print address off**. The backtrace also shows the source file name and line number, as well as the arguments to the function. The program counter value is omitted if it is at the beginning of the code for that line number.

Here is an example of a backtrace. It was made with the command '**bt 3**', so it shows the innermost three frames.

```
#0  m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
    at builtin.c:993
#1  0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2  0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
    at macro.c:71
(More stack frames follow...)
```

The display for frame zero does not begin with a program counter value, indicating that your program has stopped at the beginning of the code for line **993** of **builtin.c**.

3.8.3 Selecting a Frame

Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

backtrace
frame *n*
f *n* Select frame number *n*. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest-numbered frame is **main**'s frame.

frame *addr*
f *addr* Select the frame at address *addr*. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign numbers properly to all frames. In addition, this can be useful when your program has multiple stacks and switches between them.
On the SPARC architecture, **frame** needs two addresses to select an arbitrary frame: a frame pointer and a stack pointer.

up *n* Move *n* frames up the stack. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to one.

down *n* Move *n* frames down the stack. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to one. You may abbreviate **down** as **do**.

All of these commands end by printing two lines of output describing the frame. The first line shows the frame number, the function name, the arguments, and the source file and line number of execution in that frame. The second line shows the text of that source line. For example:

```
(gdb) up
#1  0x22f0 in main (argc=1, argv=0xf7fffbf4, env=0xf7fffbfc)
    at env.c:10
10              read_input_file (argv[i]);
```

After such a printout, the **list** command with no arguments will print ten lines centered on the point of execution in the frame.

up-silently *n*
down-silently *n*
These two commands are variants of **up** and **down**, respectively; they differ in that they do their work silently, without causing display of the new frame. They are intended primarily for use in GDB command scripts, where the output might be unnecessary and distracting.

3.8.4 Information About a Frame

There are several other commands to print information about the selected stack frame.

frame
f When used without any argument, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated **f**. With an argument, this command is used to select a stack frame.

info f This command prints a verbose description of the selected stack frame, including the address of the frame, the addresses of the next frame down (called by this frame) and the next frame up (caller of this frame), the language that the source code corresponding to this frame was written in, the address of the frame's arguments, the program counter saved in it (the address of execution in the caller frame), and which registers were saved in the frame. The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

info frame *addr*
info f *addr* Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command.

info args Print the arguments of the selected frame, each on a separate line.

info locals

Print the local variables of the selected frame, each on a separate line. These are all variables declared static or automatic within all program blocks that execution in this frame is currently inside of.

info catch

Print a list of all the exception handlers that are active in the current stack frame at the current point of execution. To see other exception handlers, visit the associated frame (using the **up**, **down**, or **frame** commands); then type **info catch**.

3.9 Examining Source Files

GDB can print parts of your program's source, since the debugging information recorded in your program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame, GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use GDB through its GNU Emacs interface, you may prefer to use Emacs facilities to view source;

3.9.1 Printing Source Lines

To print lines from a source file, use the **list** command (abbreviated **l**). There are several ways to specify what part of the file you want to print.

Here are the forms of the **list** command most commonly used:

list ***linenum***

Print lines centered around line number *linenum* in the current source file.

list ***function***

Print lines centered around the beginning of function *function*.

list Print more lines. If the last lines printed were printed with a **list** command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame, this prints lines centered around that line.

list - Print lines just before the lines last printed.

By default, GDB prints ten source lines with any of these forms of the **list** command. You can change this using **set listsize**:

set listsize ***count***

Make the **list** command display *count* source lines (unless the **list** argument explicitly specifies some other number).

show listsize

Display the number of lines that **list** will currently display by default.

Repeating a **list** command with **RET** discards the argument, so it is equivalent to typing just **list**. This is more useful than listing the same lines again. An exception is made for an argument of 'tt bf -'; that argument is preserved in repetition so that each repetition moves up in the source file.

In general, the **list** command expects you to supply zero, one or two *linespecs*. Linespecs specify source lines; there are several ways of writing them but the effect is always to specify some source line. Here is a complete description of the possible arguments for **list**:

list *linespec*
: Print lines centered around the line specified by *linespec*.

list *first*,*last*
: Print lines from *first* to *last*. Both arguments are linespecs.

list ,*last* Print lines ending with *last*.

list *first*,
: Print lines starting with *first*.

list + Print lines just after the lines last printed.

list - Print lines just before the lines last printed.

list
: As described in the preceding table.

Here are the ways of specifying a single source line—all the kinds of linespec.

number Specifies line *number* of the current source file. When a **list** command has two linespecs, this refers to the same source file as the first linespec.

+*offset* Specifies the line *offset* lines after the last line printed. When used as the second linespec in a **list** command that has two, this specifies the line *offset* lines down from the first linespec.

-*offset*
: Specifies the line *offset* lines before the last line printed.

filename*:*number
: Specifies line *number* in the source file *filename*.

function Specifies the line of the open-brace that begins the body of the function *function*.

filename:*function*
: Specifies the line of the open-brace that begins the body of the function *function* in the file *filename*. You only need the file name with a function name to avoid ambiguity when there are identically named functions in different source files.

address Specifies the line containing the program address *address*. *address* may be any expression.

3.9.2 Searching Source Files

There are two commands for searching through the current source file for a regular expression.

forward-search ***regexp***
search ***regexp***
: The command '**forward-search** ***regexp***' checks each line, starting with the one following the last line listed, for a match for *regexp*. It lists the line that is found. You can use synonym 'tt bf search *regexp*' or abbreviate the command name as **fo**.

reverse-search ***regexp***
: The command '**reverse-search** ***regexp***' checks each line, starting with the one before the last line listed and going backward, for a match for *regexp*. It lists the line that is found. You can abbreviate this command as '**rev**'.

3.9.3 Specifying Source Directories

Executable programs sometimes do not record the directories of the source files from which they were compiled, just the names. Even when they do, the directories could be moved between the compilation and your debugging session. GDB has a list of directories to search for source files; this is called the source path. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name. Note that the executable search path is @emphnot used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

If GDB cannot find a source file in the source path, and the object program records a directory, GDB tries that directory too. If the source path is empty, and there is no record of the compilation directory, GDB will, as a last resort, look in the current directory.

Whenever you reset or rearrange the source path, GDB will clear out any information it has cached about where source files are found, where each line is in the file, etc.

When you start GDB, its source path is empty. To add other directories, use the **directory** command.

directory ***dirname ...***
: Add directory *dirname* to the front of the source path. Several directory names may be given to this command, separated by 'tt bf :' or whitespace. You may specify a directory that is already in the source path; this moves it forward, so it will be searched sooner.
You can use the string '**$cdir**' to refer to the compilation directory (if one is recorded), and 'tt bf $cwd' to refer to the current working directory. '**$cwd**' is not the same as '**.**'—the former tracks the current working directory as it changes during your GDB session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

directory
: Reset the source path to empty again. This requires confirmation.

show directories
: Print the source path: show which directories it contains.

If your source path is cluttered with directories that are no longer of interest, GDB may sometimes cause confusion by finding the wrong versions of source. You can correct the situation as follows:

1. Use **directory** with no argument to reset the source path to empty.

2. Use **directory** with suitable arguments to reinstall the directories you want in the source path. You can add all the directories in one command.

3.9.4 Source and Machine Code

You can use the command **info line** to map source lines to program addresses (and viceversa), and the command **disassemble** to display a range of addresses as machine instructions.

info line ***linespec***
: Print the starting and ending addresses of the compiled code for source line *linespec*. You can specify source lines in any of the ways understood by the **list** command.

For example, we can use **info line** to discover the location of the object code for the first line of function **m4_changequote**:

```
(gdb) info line m4_changecom
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

We can also inquire (using ****addr*** as the form for *linespec*) what source line covers a particular address:

```
(gdb) info line *0x63ff
Line 926 of "builtin.c" starts at pc 0x63e4 and ends at 0x6404.
```

After **info line**, the default address for the **x** command is changed to the starting address of the line, so that 'tt bf x/i' is sufficient to begin examining the machine code. Also, this address is saved as the value of the convenience variable **$_**.

disassemble
: This specialized command is provided to dump a range of memory as machine instructions. The default memory range is the function surrounding the program counter of the selected frame. A single argument to this command is a program counter value; the function surrounding this value will be dumped. Two arguments (separated by one or more spaces) specify a range of addresses (first inclusive, second exclusive) to be dumped.

We can use **disassemble** to inspect the object code range shown in the last **info line** example:

```
(gdb) disas 0x63e4 0x6404
Dump of assembler code from 0x63e4 to 0x6404:
0x63e4 builtin_init+5340:     ble 0x63f8 builtin_init+5360
0x63e8 builtin_init+5344:     sethi %hi(0x4c00), %o0
0x63ec builtin_init+5348:     ld [%i1+4], %o0
0x63f0 builtin_init+5352:     b 0x63fc builtin_init+5364
0x63f4 builtin_init+5356:     ld [%o0+4], %o0
0x63f8 builtin_init+5360:     or %o0, 0x1a4, %o0
0x63fc builtin_init+5364:     call 0x9288 path_search
```

```
0x6400 builtin_init+5368:     nop
End of assembler dump.
(gdb)
```

3.10 Examining Data

The usual way to examine data in your program is with the **print** command (abbreviated **p**), or its synonym **inspect**. It evaluates and prints the value of an expression of the language your program is written in.

print ***exp***
print ***/f exp***
: *exp* is an expression (in the source language). By default the value of *exp* is printed in a format appropriate to its data type; you can choose a different format by specifying '***/f***', where *f* is a letter specifying the format.

print
print ***/f***
: If you omit *exp*, GDB displays the last value again (from the *value history*. This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the **x** command. It examines data in memory at a specified address and prints it in a specified format.

If you are interested in information about types, or about how the fields of a struct or class are declared, use the **ptype** ***exp*** command rather than **print**.

3.10.1 Expressions

print and many other GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you are using is legal in an expression in GDB. This includes conditional expressions, function calls, casts and string constants. It unfortunately does not include symbols defined by preprocessor **#define** commands.

Because C is so widespread, most of the expressions shown in examples in this manual are in C. for information on how to use expressions in other languages.

In this section, we discuss operators that you can use in GDB expressions regardless of your programming language.

Casts are supported in all languages, not just in C, because it is so useful to cast a number into a pointer so as to examine a structure at that address in memory.

GDB supports these operators in addition to those of programming languages:

@
: '**@@**' is a binary operator for treating parts of memory as arrays.

::
: '**::**' allows you to specify a variable in terms of the file or function where it is defined.

{type} addr

Refers to an object of type *type* stored at address *addr* in memory. *addr* may be any expression whose value is an integer or pointer (but parentheses are required around binary operators, just as in a cast). This construct is allowed regardless of what kind of data is normally supposed to reside at *addr*.

3.10.2 Program Variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame; they must either be global (or static) or be visible according to the scope rules of the programming language from the point of execution in that frame. This means that in the function

```
foo (a)
     int a;
{
  bar (a);
  {
    int b = test ();
    bar (b);
  }
}
```

the variable **a** is usable whenever your program is executing within the function **foo**, but the variable **b** is visible only while your program is executing inside the block in which **b** is declared.

There is an exception: you can refer to a variable or function whose scope is a single source file even if the current execution point is not in this file. But it is possible to have more than one such variable or function with the same name (in different source files). If that happens, referring to that name has unpredictable effects. If you wish, you can specify a variable in a particular file, using the colon-colon notation:

file::*variable* *function*::*variable*

Here *file* or *function* is the name of the context for the static *variable*. In the case of file names, you can use quotes to make sure GDB parses the file name as a single word–for example, to print a global value of 'x' defined in 'f2.c':

This use of '::' is very rarely in conflict with the very similar use of the same notation in C++. GDB also supports use of the C++ scope resolution operator in GDB expressions.

Warning: Occasionally, a local variable may appear to have the wrong value at certain points in a function—just after entry to the function, and just before exit. You may see this problem when you are stepping by machine instructions. This is because on most machines, it takes more than one instruction to set up a stack frame (including local variable definitions); if you are stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On function exit, it usually also takes more than one machine

instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

3.10.3 Artificial Arrays

It is often useful to print out several successive objects of the same type in memory; a section of an array, or an array of dynamically determined size for which only a pointer exists in the program.

You can do this by referring to a contiguous span of memory as an *artificial array* with the binary operator '**@**'. The left operand of '**@**' should be the first element of the desired array, as an individual object. The right operand should be the desired length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. Here is an example. If a program says

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of **array** with

```
p *array@len
```

The left operand of '**@**' must reside in memory. Array values made with '**@**' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history, after printing one out.)

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable as a counter in an expression that prints the first interesting value, and then repeat that expression via **RET**. For instance, suppose you have an array **dtab** of pointers to structures, and you are interested in the values of a field **fv** in each structure. Here is an example of what you might type:

```
set $i = 0
p dtab[$i++]->fv
RET
RET
. . .
```

3.10.4 Output formats

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the **print** command with a slash and a format letter. The format letters supported are:

x Regard the bits of the value as an integer, and print the integer in hexadecimal.

d Print as integer in signed decimal.

u Print as integer in unsigned decimal.

o Print as integer in octal.

t Print as integer in binary. The letter 't' stands for "two".

a Print as an address, both absolute in hex and as an offset from the nearest preceding symbol. This format can be used to discover where (in what function) an unknown address is located:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

c Regard as an integer and print it as a character constant.

f Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex, type

```
p/x $pc
```

Note that no space is required before the slash; this is because command names in GDB cannot contain a slash.

To reprint the last value in the value history with a different format, you can use the **print** command with just a format and no expression. For example, '**p/x**' reprints the last value in hex.

3.10.5 Examining Memory

You can use the command **x** (for "examine") to examine memory in any of several formats, independently of your program's data types.

x/*nfu addr*
x *addr*
x Use the command **x** to examine memory.

n, *f*, and *u* are all optional parameters that specify how much memory to display and how to format it; em addr is an expression giving the address where you want to start displaying memory. If you use defaults for *nfu*, you need not type the slash '/'. Several commands set convenient defaults for *addr*.

***n*, the repeat count**
The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units *u*) to display.

***f*, the display format**

The display format is one of the formats used by **print**, or '**s**' (null-terminated string) or '**i**' (machine instruction). The default is '**x**' (hexadecimal) initially, or the format from the last time you used either **x** or **print**.

***u*, the unit size**

The unit size is any of

b	Bytes.
h	Halfwords (two bytes).
w	Words (four bytes). This is the initial default.
g	Giant words (eight bytes).

Each time you specify a unit size with **x**, that size becomes the default unit the next time you use **x**. (For the '**s**' and '**i**' formats, the unit size is ignored and is normally not written.)

***addr*, starting display address**

addr is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. The default for *addr* is usually just after the last address examined—but several other commands also set the default address: **info breakpoints** (to the address of the last breakpoint listed), **info line** (to the starting address of a line), and **print** (if you use it to display a value from memory).

For example, '**x/3uh 0x54320**' is a request to display three halfwords (**h**) of memory, formatted as unsigned decimal integers ('**u**'), starting at address **0x54320**. '**x/4xw $sp**' prints the four words ('**w**') of memory above the stack pointer (here, '**$sp**'; in hexadecimal ('x').

Since the letters indicating unit sizes are all distinct from the letters specifying output formats, you do not have to remember whether unit size or format comes first; either order will work. The output specifications '**4xw**' and '**4wx**' mean exactly the same thing. (However, the count *n* must come first; '**wx4**' will not work.)

Even though the unit size *u* is ignored for the formats '**s**' and '**i**', you might still want to use a count *n*; for example, '**3i**' specifies that you want to see three machine instructions, including any operands. The command **disassemble** gives an alternative way of inspecting machine instructions.

All the defaults for the arguments to **x** are designed to make it easy to continue scanning memory with minimal specifications each time you use **x**. For example, after you have inspected three machine instructions with '**x/3i *addr***', you can inspect the next seven with just '**x/7**'. If you use **RET** to repeat the **x** command, the repeat count *n* is used again; the other arguments default as for successive uses of **x**.

The addresses and contents printed by the **x** command are not saved in the value history because there is often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables **$_** and **$__**. After an **x** command, the last address examined is available for use in expressions in the convenience variable **$_**. The contents of that address, as examined, are available in the convenience variable **$__**.

If the **x** command has a repeat count, the address and contents saved are from the last memory unit printed; this is not the same as the last address printed if several units were

printed on the last line of output.

3.10.6 Automatic Display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that GDB will print its value each time your program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

This display shows item numbers, expressions and their current values. As with displays you request manually using **x** or **print**, you can specify the output format you prefer; in fact, **display** decides whether to use **print** or **x** depending on how elaborate your format specification is—it uses **x** if you specify a unit size, or one of the two formats (‘**i**’ and ‘**s**’) that are only supported by **x**; otherwise it uses **print**.

display ***exp***
: Add the expression *exp* to the list of expressions to display each time your program stops.
display will not repeat if you press **RET** again after using it.

display/*fmt exp*
: For *fmt* specifying only a display format and not a size or count, add the expression *exp* to the auto-display list but arranges to display it each time in the specified format *fmt*.

display/*fmt addr*
: For *fmt* ‘**i**’ or ‘**s**’, or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time your program stops. Examining means in effect doing ‘**x/*fmt addr***’.

For example, ‘**display/i $pc**’ can be helpful, to see the machine instruction about to be executed each time execution stops (‘**$pc**’ is a common name for the program counter.

undisplay ***dnums ...***
delete display ***dnums ...***
: Remove item numbers *dnums* from the list of expressions to display.
undisplay will not repeat if you press **RET** after using it. (Otherwise you would just get the error ‘**No display number ...**’

disable display ***dnums...***
: Disable the display of item numbers *dnums*. A disabled display item is not printed automatically, but is not forgotten. It may be enabled again later.

enable display ***dnums...***
: Enable display of item numbers *dnums*. It becomes effective once again in auto display of its expression, until you specify otherwise.

display
: Display the current values of the expressions on the list, just as is done when your program stops.

info displays

Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions which would not be displayed right now because they refer to automatic variables not currently available.

If a display expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is disabled when execution enters a context where one of its variables is not defined. For example, if you give the command **display last_char** while inside a function with an argument **last_char**, then this argument will be displayed while your program continues to stop inside that function. When it stops elsewhere—where there is no variable **last_char**—display is disabled. The next time your program stops where **last_char** is meaningful, you can enable the display expression once again.

3.10.7 Print Settings

GDB provides the following ways to control how arrays, structures, and symbols are printed.

These settings are useful for debugging programs in any language:

set print address
set print address on

GDB will print memory addresses showing the location of stack traces, structure values, pointer values, breakpoints, and so forth, even when it also displays the contents of those addresses. The default is on. For example, this is what a stack frame display looks like, with **set print address on**:

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530         if (lquote != def_lquote)
```

set print address off

Do not print addresses when displaying their contents. For example, this is the same stack frame displayed with **set print address off**:

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
530         if (lquote != def_lquote)
```

show print address

Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest earlier symbol plus an offset. If that symbol does not uniquely identify the address (for example, it is a name whose scope is a single source file), you may need to disambiguate. One way to do this is with ‘info line’, for example ‘info line *0x4537’.Alternately, you can set GDB to print the source file and

line number when it prints a symbolic address:

set print symbol-filename on
: Tell GDB to print the source file name and line number of a symbol in the symbolic form of an address.

set print symbol-filename off
: Do not print source file name and line number of a symbol. This is the default.

show print symbol-filename
: Show whether or not GDB will print the source file name and line number of a symbol in the symbolic form of an address.

Another situation where it is helpful to show symbol filenames and line numbers is when disassembling code; GDB shows you the line number and source file that corresponds to each instruction.

Also, you may wish to see the symbolic form only if the address being printed is reasonably close to the closest earlier symbol:

set print max-symbolic-offset *max-offset*
: Tell GDB to only display the symbolic form of an address if the offset between the closest earlier symbol and the address is less than *max-offset*. The default is 0, which means to always print the symbolic form of an address, if any symbol precedes it.

show print max-symbolic-offset
: Ask how large the maximum offset is that GDB prints in a symbolic address.

Sometimes GDB can tell you more about an address if it does an extensive search of its symbol information. The default is to provide a quick symbolic display that is usually correct, but which may not give the most useful answer when working in some object file formats. If you are not getting the information you need, try:

set print fast-symbolic-addr off
: Search all symbol information when displaying an address symbolically. This setting may display more information about static variables, for example, but also takes longer.

set print fast-symbolic-addr
set print fast-symbolic-addr on
: Search only the "minimal symbol information" when displaying symbolic information about an address. This is the default.

show print fast-symbolic-addr
: Ask whether GDB is using a fast or slow method of printing symbolic address.

If you have a pointer and you are not sure where it points, try **set print symbol-filename on** and **set print fast-symbolic-addr off**. Then you can determine the name and source file location of the variable where it points, using *pi$/a POINTER*. This interprets the address in symbolic form. For example, here GDB shows that a variable 'ptt' points at another variable 't', defined in 'hi2.c':

```
(gdb) set print fast-symbolic-addr off
```

```
(gdb) set print symbol-filename on
(gdb) p/a ptt
$4 = 0xe008 <t in hi2.c>
```

Other settings control how different kinds of objects are printed:

set print array
set print array on
: GDB will pretty print arrays. This format is more convenient to read, but uses more space. The default is off.

set print array off
: Return to compressed format for arrays.

show print array
: Show whether compressed or pretty format is selected for displaying arrays.

set print elements ***number-of-elements***
: If GDB is printing a large array, it stops printing after it has printed the number of elements set by the **set print elements** command. This limit also applies to the display of strings.Setting the number of elements to zero means that the printing is unlimited.

show print elements
: Display the number of elements of a large array that GDB will print before losing patience.

set print pretty on
: Cause GDB to print structures in an indented format with one member per line, like this:

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
 }
```

set print pretty off
: Cause GDB to print structures in a compact format, like this:

```
$1 = @next = 0x0, flags = @{sweet = 1, sour = 1@}, meat \
= 0x54 "Pork"@}
```

This is the default format.

show print pretty
: Show which format GDB will use to print structures.

set print sevenbit-strings on
: Print using only seven-bit characters; if this option is set, GDB will display any eight-bit characters (in strings or character values) using the notation *nnn*. This setting is best if you are working in English (ASCII) and you use the high-order bit of characters as a marker or "meta" bit.

set print sevenbit-strings off
: Print full eight-bit characters. This allows the use of more international character sets, and is the default.

show print sevenbit-strings
: Show whether or not GDB will print only seven-bit characters.

set print union on
: Tell GDB to print unions which are contained in structures. This is the default setting.

set print union off
: Tell GDB not to print unions which are contained in structures.

show print union
: Ask GDB whether or not it will print unions which are contained in structures.
For example, given the declarations

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
              Bug_forms;

struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};

struct thing foo = {Tree, {Acorn}};
```

with **set print union on** in effect '**p foo**' would print

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

and with **set print union off** in effect it would print

```
$1 = {it = Tree, form = {...}}
```

These settings are of interest when debugging C++ programs:

set print demangle
set print demangle on
Print C++ names in their source form rather than in the encoded ("mangled") form passed to the assembler and linker for type-safe linkage. The default is 'on'.

show print demangle
Show whether C++ names will be printed in mangled or demangled form.

set print asm-demangle
set print asm-demangle on
Print C++ names in their source form rather than their mangled form, even in assembler code printouts such as instruction disassemblies. The default is off.

show print asm-demangle
Show whether C++ names in assembly listings will be printed in mangled or demangled form.

set demangle-style *style*
Choose among several encoding schemes used by different compilers to represent C++ names. The choices for *style* are currently:

auto
Allow GDB to choose a decoding style by inspecting your program.

gnu
Decode based on the GNU C++ compiler ('g++') encoding algorithm.

lucid
Decode based on the Lucid C++ compiler ('lcc') encoding algorithm.

arm
Decode using the algorithm in the 'C++ Annotated Reference Manual'. *Warning:* this setting alone is not sufficient to allow debugging 'cfront'-generated executables. GDB would require further enhancement to permit that.

show demangle-style
Display the encoding style currently in use for decoding C++ symbols.

set print object
set print object on
When displaying a pointer to an object, identify the *actual* (derived) type of the object rather than the *declared* type, using the virtual function table.

set print object off
Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

show print object
Show whether actual, or declared, object types will be displayed.

set print vtbl
set print vtbl on
Pretty print C++ virtual function tables. The default is off.

set print vtbl off
Do not pretty print C++ virtual function tables.

show print vtbl
Show whether C++ virtual function tables are pretty printed, or not.

3.10.8 Value History

Values printed by the **print** command are saved in GDB's *value history* so that you can refer to them in other expressions. Values are kept until the symbol table is re-read or discarded (for example with the **file** or **symbol-file** commands). When the symbol table changes, the value history is discarded, since the values may contain pointers back to the types defined in the symbol table.

The values printed are given *history numbers* for you to refer to them by. These are successive integers starting with one. **print** shows you the history number assigned to a value by printing '**$*num* =** ' before the value; here *num* is the history number.

To refer to any previous value, use '**$**' followed by the value's history number. The way **print** labels its output is designed to remind you of this. Just **$** refers to the most recent value in the history, and **$$** refers to the value before that. **$$*n*** refers to the *n*th value from the end; **$$2** is the value just prior to **$$**, **$$1** is equivalent to **$$**, and **$$0** is equivalent to **$**.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices to type

```
p *$
```

If you have a chain of structures where the component **next** points to the next one, you can print the contents of the next one with this:

```
p *$.next
```

You can print successive links in the chain by repeating this command—which you can do by just typing **RET**.

Note that the history records values, not expressions. If the value of **x** is 4 and you type these commands:

```
print x
set x=5
```

then the value recorded in the value history by the **print** command remains 4 even though the value of **x** has changed.

show values
: Print the last ten values in the value history, with their item numbers. This is like '**p $$9**' repeated ten times, except that **show values** does not change the history.

show values ***n***
: Print ten history values centered on history item number *n*.

show values +
: Print ten history values just after the values last printed. If no more values are available, produces no display.

Pressing **RET** to repeat **show values** ***n*** has exactly the same effect as '**show values +**'.

3.10.9 Convenience Variables

GDB provides *convenience variables* that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no direct effect on further execution of your program. That is why you can use them freely.

Convenience variables are prefixed with '**$**'. Any name preceded by '**$**' can be used for a convenience variable, unless it is one of the predefined machine-specific register names. (Value history references, in contrast, are *numbers* preceded by '**$**').

You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. Example:

```
set $foo = *object_ptr
```

would save in **$foo** the value contained in the object pointed to by **object_ptr**.

Using a convenience variable for the first time creates it; but its value is **void** until you assign a new value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign a convenience variable any type of value, including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

show convenience
: Print a list of convenience variables used so far, and their values. Abbreviated **show con**.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example, to print a field from successive elements of an array of structures:

```
set $i = 0
print bar[$i++]->contents
```

... repeat that command by typing **RET**.

Some convenience variables are created automatically by GDB and given values likely to be useful.

$_** The variable **$_ is automatically set by the **x** command to the last address examined. Other commands which provide a default address for **x** to examine also set **$_** to that address; these commands include **info line** and **info breakpoint**. The type of **$_** is **void *** except when set by the **x** command, in which case it is a pointer to the type of **$__**.

$__** The variable **$__ is automatically set by the **x** command to the value found in the last address examined. Its type is chosen to match the format in which the data was printed.

3.10.10 Registers

You can refer to machine register contents, in expressions, as variables with names starting with '**$**'. The names of registers are different for each machine; use **info registers** to see the names used on your machine.

info registers
Print the names and values of all registers except floating-point registers (in the selected stack frame).

info all-registers
Print the names and values of all registers, including floating-point registers.

info registers ***regname***
Print the relativized value of register *regname*. *regname* may be any register name valid on the machine you are using, with or without the initial '**$**'.

GDB has four "standard" register names that are available (in expressions) on most machines—whenever they do not conflict with an architecture's canonical mnemonics for registers. The register names **$pc** and **$sp** are used for the program counter register and the stack pointer. **$fp** is used for a register that contains a pointer to the current stack frame, and **$ps** is used for a register that contains the processor status. For example, you could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer [4] with

```
set $sp += 4
```

Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict. The **info registers** command shows the canonical names. For example, on the SPARC, **info registers** displays the processor status register as **$psr** but you can also refer to it as **$ps**.

[4] This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting **$sp** is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use **return**

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but floating point; these registers are considered to have floating point values. There is no way to refer to the contents of an ordinary register as floating point value (although you can *print* it as a floating point value with '**print/f $*regname*'**).

Some registers have distinct "raw" and "virtual" data formats. This means that the data format in which the register contents are saved by the operating system is not the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in "extended" (raw) format, but all C programs expect to work with "double" (virtual) format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the **info registers** command prints the data in both formats.

Normally, register values are relative to the selected stack frame. This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with '**frame 0**').

However, GDB must deduce where registers are saved, from the machine code generated by your compiler. If some registers are not saved, or if GDB is unable to locate the saved registers, the selected stack frame will make no difference.

3.10.11 Floating Point Hardware

Depending on the host machine architecture, GDB may be able to give you more information about the status of the floating point hardware.

info float

Display hardware-dependent information about the floating point unit. The exact contents and layout vary depending on the floating point chip; on some platforms, 'info float' is not available at all.

3.11 Examining the Symbol Table

The commands described in this section allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and does not change as your program executes. GDB finds it in your program's symbol table, in the file indicated when you started GDB, or by one of the file-management commands.

info address *symbol*

Describe where the data for *symbol* is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored.
Note the contrast with '**print &*symbol***', which does not work at all for a register variables, and for a stack local variable prints the exact address of the current instantiation of the variable.

whatis *exp*

Print the data type of expression *exp*. *exp* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place.

whatis Print the data type of **$**, the last value in the value history.

ptype ***typename***

Print a description of data type *typename*. *typename* may be the name of a type, or for C code it may have the form '**struct** ***struct-tag***', '**union** ***union-tag***' or '**enum** ***enum-tag***.'

ptype *exp*

ptype Print a description of the type of expression *exp*. **ptype** differs from **whatis** by printing a detailed description, instead of just the name of the type. For example, if your program declares a variable as

```
struct complex {double real; double imag;} v;
```

compare the output of the two commands:

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

As with **whatis**, using **ptype** without an argument refers to the type of **$**, the last value in the value history.

info types ***regexp***

info types

Print a brief description of all types whose name matches *regexp* (or all types in your program, if you supply no argument). Each complete typename is matched as though it were a complete line; thus, '**i type value**' gives information on all types in your program whose name includes the string **value**, but '**i type ^value$**' gives information only on types whose complete name is **value**.

This command differs from **ptype** in two ways: first, like **whatis**, it does not print a detailed description; second, it lists all source files where a type is defined.

info source

Show the name of the current source file—that is, the source file for the function containing the current point of execution—and the language it was written in.

info sources
: Print the names of all source files in your program for which there is debugging information, organized into two lists: files whose symbols have already been read, and files whose symbols will be read when needed.

info functions
: Print the names and data types of all defined functions.

info functions ***regexp***
: Print the names and data types of all defined functions whose names contain a match for regular expression *regexp*. Thus, '**info fun step**' finds all functions whose names include **step**; '**info fun ^step**' finds those whose names start with **step**.

info variables
: Print the names and data types of all variables that are declared outside of functions (i.e., excluding local variables).

info variables ***regexp***
: Print the names and data types of all variables (except for local variables) whose names contain a match for regular expression *regexp*.

maint print symbols ***filename***
maint print psymbols ***filename***
maint print msymbols ***filename***
: Write a dump of debugging symbol data into the file *filename*. These commands are used to debug the GDB symbol-reading code. Only symbols with debugging data are included. If you use **maint print symbols**, GDB includes all the symbols for which it has already collected full details: that is, *filename* reflects symbols for only those files whose symbols GDB has read. You can use the command **info sources** to find out which files these are. If you use **maint print psymbols**, the dump also shows information about symbols that GDB only knows partially—that is, symbols defined in files that GDB has skimmed, but not yet read completely. Finally, **maint print msymbols** dumps just the minimal symbol information required for each object file from which GDB has read some symbols.

3.12 Altering Execution

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program.

For example, you can store new values into variables or memory locations, give your program a signal, restart it at a different address, or even return prematurely from a function to its caller.

3.12.1 Assignment to Variables

To alter the value of a variable, evaluate an assignment expression. For example,

```
print x=4
```

stores the value 4 into the variable **x**, and then prints the value of the assignment expression (which is 4).

If you are not interested in seeing the value of the assignment, use the **set** command instead of the **print** command. **set** is really the same as **print** except that the expression's value is not printed and is not put in the value history. The expression is evaluated only for its effects.

If the beginning of the argument string of the **set** command appears identical to a **set** subcommand, use the **set variable** command instead of just **set**. This command is identical to **set** except for its lack of subcommands. For example, a program might well have a variable **width**—which leads to an error if we try to set a new value with just '**set width=13**', as we might if **set width** did not happen to be a GDB command:

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

The invalid expression, of course, is '**=47**'. What we can do in order to actually set our program's variable **width** is

```
(gdb) set var width=47
```

GDB allows more implicit conversions in assignments than C; you can freely store an integer value into a pointer variable or vice versa, and any structure can be converted to any other structure that is the same length or shorter.

To store values into arbitrary places in memory, use the '**{...}**' construct to generate a value of specified type at a specified address. For example, **{int}0x83040** refers to memory location **0x83040** as an integer (which implies a certain size and representation in memory), and

```
set {int}0x83040 = 4
```

stores the value 4 into that memory location.

3.12.2 Continuing at a Different Address

Ordinarily, when you continue your program, you do so at the place where it stopped, with the **continue** command. You can instead continue at an address of your own choosing, with the following commands:

jump ***linespec***

Resume execution at line *linespec*. Execution will stop immediately if there is a breakpoint there.

The **jump** command does not change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If line *linespec* is in a different function from the one currently executing, the results may be bizarre if the two functions expect different patterns of arguments or of local variables. For this reason, the **jump** command requests confirmation if the specified line is not in the function currently executing. However, even bizarre results are predictable if you are well acquainted with the machine-language code of your program.

jump ****address***

Resume execution at the instruction at address **address**.

You can get much the same effect as the **jump** command by storing a new value into the register **$pc**. The difference is that this does not start your program running; it only changes the address where it *will* run when it is continued. For example,

```
set $pc = 0x485
```

causes the next **continue** command or stepping command to execute at address **0x485**, rather than at the address where your program stopped.

The most common occasion to use the **jump** command is to back up, perhaps with more breakpoints set, over a portion of a program that has already executed, in order to examine its execution in more detail.

3.12.3 Giving your program a Signal

signal ***signal***

Resume execution where your program stopped, but give it immediately the signal number *signal*. *signal* can be the name or the number of a signal. For example, on many systems *signal 2* and *signal* **sigint** are both ways of sending an interrupt signal.

Alternatively, if *signal* is zero, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would ordinary see the signal when resumed with the **continue** command; '**signal 0**' causes it to resume without a signal.

signal does not repeat when you press `bf RET` a second time after executing the command.

Invoking the *signal* command is not the same as invoking the 'kill' utility from the shell. Sending a signal with 'kill' causes GDB to decide what to do with the signal depending on the signal handling tables. The *signal* command passes the signal directly to your program.

3.12.4 Returning from a Function

return
return expression
: You can cancel execution of a function call with the **return** command. If you give an *expression* argument, its value is used as the function's return value.

When you use **return**, GDB discards the selected stack frame (and all frames within it). You can think of this as making the discarded frame return prematurely. If you wish to specify a value to be returned, give that value as the argument to **return**.

This pops the selected stack frame, and any other frames inside of it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The **return** command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the **finish** command resumes execution until the selected stack frame returns naturally.

3.12.5 Calling program functions

call *expr* Evaluate the expression *expr* without displaying **void** returned values.

You can use this variant of the **print** command if you want to execute a function from your program, but without cluttering the output with **void** returned values. The result is printed and saved in the value history, if it is not void.

3.12.6 Patching your Program

By default, GDB opens the file containing your program's executable code (or the corefile) read-only. This prevents accidental alterations to machine code; but it also prevents you from intentionally patching your program's binary.

If you'd like to be able to patch the binary, you can specify that explicitly with the **set write** command. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

set write on
set write off
: If you specify '**set write on**', GDB will open executable and core files for both reading and writing; if you specify '**set write off**' (the default), GDB will open them read-only.
If you have already loaded a file, you must load it again (using the **exec-file** or **core-file** command) after changing **set write**, for your new setting to take effect.

show write
: Display whether executable files and core files will be opened for writing as well as reading.

3.13 GDB's Files

GDB needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start your program. To debug a core dump of a previous run, GDB must be told the file name of the core dump.

3.13.1 Commands to Specify Files

The usual way to specify executable and core dump file names is with the command arguments given when you start GDB.

Occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify the files you want to use. In these situations the GDB commands to specify new files are useful.

file ***filename***
: Use *filename* as the program to be debugged. It is read for its symbols and for the contents of pure memory. It is also the program executed when you use the **run** command. If you do not specify a directory and the file is not found in GDB's working directory, GDB uses the environment variable **PATH** as a list of directories to search, just as the shell does when looking for a program to run. You can change the value of this variable, for both GDB and your program, using the **path** command.
On systems with memory-mapped files, an auxiliary file *filename*.syms may hold symbol table information for *filename*. If so, GDB maps in the symbol table from *filename*.syms, starting up more quickly. See the descriptions of the options '-mapped' and '-readnow' (available on the command line, and with the commands 'file', 'symbol-file', or 'add-symbol-file'), for more information.

file
: **file** with no argument makes GDB discard any information it has on both executable file and the symbol table.

exec-file [***filename***]
: Specify that the program to be run (but not the symbol table) is found in *filename*. GDB will search the environment variable **PATH** if necessary to locate your program. Omitting *filename* means to discard information on the executable file.

symbol-file [***filename***]

Read symbol table information from file *filename*. **PATH** is searched when necessary. Use the **file** command to get both symbol table and program to run from the same file.

symbol-file with no argument clears out GDB's information on your program's symbol table.

The **symbol-file** command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside GDB.

symbol-file will not repeat if you press **RET** again after executing it once.

On some kinds of object files, the **symbol-file** command does not actually read the symbol table in full right away. Instead, it scans the symbol table quickly to find which source files and which symbols are present. The details are read later, one source file at a time, as they are needed.

The purpose of this two-stage reading strategy is to make GDB start up faster. For the most part, it is invisible except for occasional pauses while the symbol table details for a particular source file are being read. (The **set verbose** command can turn these pauses into messages if desired.

When the symbol table is stored in COFF format, **symbol-file** does read the symbol table data in full right away. We have not implemented the two-stage strategy for COFF yet.

symbol-file ***filename***[***-readnow***] [***-mapped***]
-file ***filename*** [***-readnow***] [***-mapped***]

You can override the GDB two-stage strategy for reading symbol tables by using the *-readnow* option with any of the commands that load symbol table information, if you want to be sure GDB has the entire symbol table available.

If memory-mapped files are available on your system through the 'mmap' system call, you can use another option, *-mapped*, to cause GDB to write the symbols for your program into a reusable file. Future GDB debugging sessions map in symbol information from this auxiliary symbol file (if the program has not changed), rather than spending time reading the symbol table from the executable program. Using the *-mapped* option has the same effect as starting GDB with the *-mapped* command-line option.

You can use both options together, to make sure the auxiliary symbol file has all the symbol information for your program.

The auxiliary symbol file for a program called **myprog** is called **myprog.syms**. Once this file exists (so long as it is newer than the corresponding executable), GDB always attempts to use it when you debug **myprog**; no special options or commands are needed.

The *.syms* file is specific to the host machine where you run GDB. It holds an exact image of the internal GDB symbol table. It cannot be shared across multiple host platforms.

core-file [***filename***]

Specify the whereabouts of a core dump file to be used as the "contents of memory". Traditionally, core files contain only some parts of the address space of the process that generated them; GDB can access the executable file itself for other parts.

core-file with no argument specifies that no core file is to be used.

Note that the core file is ignored when your program is actually running under GDB. So, if you have been running your program and you wish to debug a core file instead, you must kill the subprocess in which the program is running. To do this, use the **kill** command.

load ***filename***

Depending on what remote debugging facilities are configured into GDB, the **load** command may be available. Where it exists, it is meant to make *filename* (an executable) available for debugging on the remote system—by downloading, or dynamic linking, for example. **load** also records *filename*'s symbol table in GDB, like the **add-symbol-file** command.

If **load** is not available on your GDB, attempting to execute it gets the error message "**You can't do that when your target is ...**"

On VxWorks, **load** will dynamically link *filename* on the current target system as well as adding its symbols in GDB.

With the Nindy interface to an Intel 960 board, **load** will download *filename* to the 960 as well as adding its symbols in GDB.

load will not repeat if you press **RET** again after using it.

add-symbol-file ***filename address***

The **add-symbol-file** command reads additional symbol table information from the file *filename*. You would use this command when *filename* has been dynamically loaded (by some other means) into the program that is running. *address* should be the memory address at which the file has been loaded; GDB cannot figure this out for itself. You can specify *address* as an expression.

The symbol table of the file *filename* is added to the symbol table originally read with the **symbol-file** command. You can use the **add-symbol-file** command any number of times; the new symbol data thus read keeps adding to the old. To discard all old symbol data instead, use the **symbol-file** command.

add-symbol-file will not repeat if you press **RET** after using it.

You can use the *-mapped* and *-readnow* options just as with the **symbol-file** command, to change how GDB manages the symbol table information for *filename*.

info files
info target

info files and **info target** are synonymous; both print the current targets, including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The command **help targets** lists all possible targets rather than current ones.

All file-specifying commands allow both absolute and relative file names as arguments. GDB always converts the file name to an absolute path name and remembers it that way.

GDB supports the SunOS shared library format. GDB automatically loads symbol definitions from shared libraries when you use the **run** command, or when you examine a core file. (Before you issue the **run** command, GDB will not understand references to a function in a shared library, however—unless you are debugging a core file).

info share
info sharedlibrary
: Print the names of the shared libraries which are currently loaded.

sharedlibrary *regex*
share *regex*
: This is an obsolescent command; you can use it to explicitly load shared object library symbols for files matching a UNIX regular expression, but as with files loaded automatically, it will only load shared libraries required by your program for a core file or after typing **run**. If *regex* is omitted all shared libraries required by your program are loaded.

3.13.2 Errors Reading Symbol Files

While reading a symbol file, GDB will occasionally encounter problems, such as symbol types it does not recognize, or known bugs in compiler output. By default, GDB does not notify you of such problems, since they are relatively common and primarily of interest to people debugging compilers. If you are interested in seeing information about ill-constructed symbol tables, you can either ask GDB to print only one message about each such type of problem, no matter how many times the problem occurs; or you can ask GDB to print more messages, to see how many times the problems occur, with the **set complaints** command.

The messages currently printed, and their meanings, are:

inner block not inside outer block in *symbol*
: The symbol information shows where symbol scopes begin and end (such as at the start of a function or a block of statements). This error indicates that an inner scope block is not fully contained in its outer scope blocks.
GDB circumvents the problem by treating the inner block as if it had the same scope as the outer block. In the error message, *symbol* may be shown as "**(don't know)**" if the outer block is not a function.

block at *address* **out of order**
: The symbol information for symbol scope blocks should occur in order of increasing addresses. This error indicates that it does not do so.
GDB does not circumvent this problem, and will have trouble locating symbols in the source file whose symbols being read. (You can often determine what source file is affected by specifying **set verbose on**.

bad block start address patched
: The symbol information for a symbol scope block has a start address smaller than the address of the preceding source line. This is known to occur in the SunOS 4.1.1 (and earlier) C compiler.
GDB circumvents the problem by treating the symbol scope block as starting on the previous source line.

bad string table offset in symbol *n*
: Symbol number *n* contains a pointer into the string table which is larger than the size of the string table.
GDB circumvents the problem by considering the symbol to have the name **foo**, which may cause other problems if many symbols end up with this name.

unknown symbol type 0x*nn*
: The symbol information contains new data types that GDB does not yet know how to read. **0x*nn*** is the symbol type of the misunderstood information, in hexadecimal.
GDB circumvents the error by ignoring this symbol information. This will usually allow your program to be debugged, though certain symbols will not be accessible. If you encounter such a problem and feel like debugging it, you can debug **gdb** with itself, breakpoint on **complain**, then go up to the function **read_dbx_symtab** and examine ***bufp** to see the symbol.

stub type has NULL name
: GDB could not find the full definition for a struct or class.

const/volatile indicator missing (ok if using g++ v1.x), got ...
: The symbol information for a C++ member function is missing some information that recent versions of the compiler should have output for it.

info mismatch between compiler and debugger
: GDB could not parse a type specification output by the compiler.

3.14 Controlling GDB

You can alter many aspects of GDB's interaction with you by using the **set** command.

3.14.1 Prompt

GDB indicates its readiness to read a command by printing a string called the *prompt*. This string is normally '**(gdb)**'. You can change the prompt string with the **set prompt** command. For instance, when debugging GDB with GDB, it is useful to change the prompt in one of the GDBs so that you can always tell which one you are talking to.

set prompt *newprompt*
: Directs GDB to use *newprompt* as its prompt string henceforth.

show prompt
: Prints a line of the form: '**Gdb's prompt is: *your-prompt***'

3.14.2 Command Editing

GDB reads its input commands via the *readline* interface. This GNU library provides consistent behavior for programs which provide a command line interface to the user. Advantages are **emacs**-style or **vi**-style inline editing of commands, **csh**-like history substitution, and a storage and recall of command history across debugging sessions.

You may control the behavior of command line editing in GDB with the command **set**.

set editing
set editing on
: Enable command line editing (enabled by default).

set editing off
: Disable command line editing.

show editing
: Show whether command line editing is enabled.

3.14.3 Command History

GDB can keep track of the commands you type during your debugging sessions, so that you can be certain of precisely what happened. Use these commands to manage the GDB command history facility.

set history filename ***fname***
: Set the name of the GDB command history file to *fname*. This is the file from which GDB will read an initial command history list or to which it will write this list when it exits. This list is accessed through history expansion or through the history command editing characters listed below. This file defaults to the value of the environment variable **GDBHISTFILE**, or to

set history save
set history save on
: Record command history in a file, whose name may be specified with the **set history filename** command. By default, this option is disabled.

set history save off
: Stop recording command history in a file.

set history size ***size***
: Set the number of commands which GDB will keep in its history list. This defaults to the value of the environment variable **HISTSIZE**, or to 256 if this variable is not set.

History expansion assigns special meaning to the character **!**. Since **!** is also the logical not operator in C, history expansion is off by default. If you decide to enable history expansion with the **set history expansion on** command, you may sometimes need to follow **!** (when it is used as logical not, in an expression) with a space or a tab to prevent it from being expanded. The readline history facilities will not attempt substitution on the strings **!=** and **!(**, even when history expansion is enabled.

The commands to control history expansion are:

set history expansion on
set history expansion
: Enable history expansion. History expansion is off by default.

set history expansion off
: Disable history expansion.
The readline code comes with more complete documentation of editing and history expansion features. Users unfamiliar with **emacs** or **vi** may wish to read it.

show history
show history filename
show history save
show history size
show history expansion
: These commands display the state of the GDB history parameters. **show history** by itself displays all four states.

show commands
: Display the last ten commands in the command history.

show commands *n*
: Print ten commands centered on command number *n*.

show commands +
: Print ten commands just after the commands last printed.

3.14.4 Screen Size

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Type **RET** when you want to continue the output. GDB also uses the screen width setting to determine when to wrap lines of output. Depending on what is being printed, it tries to break the line at a readable place, rather than simply letting it overflow onto the following line.

Normally GDB knows the size of the screen from the termcap data base together with the value of the **TERM** environment variable and the **stty rows** and **stty cols** settings. If this is not correct, you can override it with the **set height** and **set width** commands:

set height *lpp*
show height
set width *cpl*
show width
: These **set** commands specify a screen height of *lpp* lines and a screen width of *cpl* characters. The associated **show** commands display the current settings.
If you specify a height of zero lines, GDB will not pause during output no matter how long the output is. This is useful if output is to a file or to an editor buffer.

3.14.5 Numbers

You can always enter numbers in octal, decimal, or hexadecimal in GDB by the usual conventions: octal numbers begin with '**0**', decimal numbers end with '**.**', and hexadecimal numbers begin with '**0x**'. Numbers that begin with none of these are, by default, entered in base 10; likewise, the default display for numbers—when no particular format is specified—is base 10. You can change the default base for both input and output with the **set radix** command.

set radix ***base***
: Set the default base for numeric input and display. Supported choices for *base* are decimal 2, 8, 10, 16. *base* must itself be specified either unambiguously or using the current default radix; for example, any of

```
set radix 012
set radix 10.
set radix 0xa
```

will set the base to decimal. On the other hand, '**set radix 10**' will leave the radix unchanged no matter what it was.

show radix
: Display the current default base for numeric input and display.

3.14.6 Optional Warnings and Messages

By default, GDB is silent about its inner workings. If you are running on a slow machine, you may want to use the **set verbose** command. It will make GDB tell you when it does a lengthy internal operation, so you will not think it has crashed.

Currently, the messages controlled by **set verbose** are those which announce that the symbol table for a source file is being read, in the description of the command **symbol-file**).

set verbose on
: Enables GDB's output of certain informational messages.

set verbose off
: Disables GDB's output of certain informational messages.

show verbose
: Displays whether **set verbose** is on or off.

By default, if GDB encounters bugs in the symbol table of an object file, it is silent; but if you are debugging a compiler, you may find this information useful.

set complaints ***limit***
: Permits GDB to output *limit* complaints about each type of unusual symbols before becoming silent about the problem. Set *limit* to zero to suppress all complaints; set it to a large number to prevent complaints from being suppressed.

show complaints
: Displays how many symbol complaints GDB is permitted to produce.

By default, GDB is cautious, and asks what sometimes seem to be a lot of stupid questions

to confirm certain commands. For example, if you try to run a program which is already running:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

If you are willing to unflinchingly face the consequences of your own commands, you can disable this "feature":

set confirm off
: Disables confirmation requests.

set confirm on
: Enables confirmation requests (the default).

show confirm
: Displays state of confirmation requests.

Some systems allow individual object files that make up your program to be replaced without stopping and restarting your program. For example, in VxWorks you can simply recompile a defective object file and keep on running. If you are running on one of these systems, you can allow GDB to reload the symbols for automatically relinked modules:

set symbol-reloading on
: Replace symbol definitions for the corresponding source file when an object file with a particular name is seen again.

set symbol-reloading off
: Do not replace symbol definitions when re-encountering object files of the same name. This is the default state; if you are not running on a system that permits automatically relinking modules, you should leave **symbol-reloading** off, since otherwise GDB may discard symbols when linking large programs, that may contain several modules (from different directories or libraries) with the same name.

show symbol-reloading
: Show the current **on** or **off** setting.

A Copyright conditions

B GNU General Public License

B.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

B.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public

License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software

which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

B.3 Applying These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> ***one line to give the program's name and an idea of what it does.***
> **Copyright (C) 19*yy name of author***
> **This program is free software; you can redistribute it and/or**
> **modify it under the terms of the GNU General Public License**
> **as published by the Free Software Foundation; either version 2**
> **of the License, or (at your option) any later version.**
> **This program is distributed in the hope that it will be useful,**
> **but WITHOUT ANY WARRANTY; without even the implied warranty of**
> **MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.**
> **See the GNU General Public License for more details.**
> **You should have received a copy of the GNU General Public License**
> **along with this program; if not, write to the**
> **Free Software Foundation, Inc., 675 Mass Ave,**
> **Cambridge, MA 02139, USA.**

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> **Gnomovision version 69, Copyright (C) 19*yy name of author***
> **Gnomovision comes with ABSOLUTELY NO WARRANTY; for details**
> **type 'show w'. This is free software, and you are welcome**
> **to redistribute it under certain conditions; type 'show c'**
> **for details.**

The hypothetical commands '**show w**' and '**show c**' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '**show w**' and '**show c**'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> **Yoyodyne, Inc., hereby disclaims all copyright**
> **interest in the program 'Gnomovision'**
> **(which makes passes at compilers) written**
> **by James Hacker.**

***signature of Ty Coon*, 1 April 1989**
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

4 Make — A Program for Maintaining Computer Programs

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

4.1 Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc[1] or Lex[2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

make

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — *make* — test . . .

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. *Make* was designed for use on Unix, but a version runs on GCOS.

4.2 Basic Features

The basic operation of *make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *lS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  –lS  –o  prog

x.o  y.o :    defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three "*.c*" files corresponding to the needed "*.o*" files, and uses built-in information on how to generate an object from a source file (*i.e.,* issue a "cc –c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*'s innate knowledge:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  –lS  –o  prog

x.o :  x.c  defs
        cc  –c  x.c
```

```
y.o :  y.c  defs
        cc  –c  y.c
z.o :  z.c
        cc  –c  z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new "*.o*" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

OBJECTS = x.o y.o z.o

LIBES = –lS

prog: $(OBJECTS)

cc $(OBJECTS) $(LIBES) –o prog

. . .

The command

make

loads the three object files with the *lS* library. The command

make "LIBES= –ll –lS"

loads them with both the Lex ("–ll") and the Standard ("–lS") libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in UNIX commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

4.3 Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

2 = xyz

abc = –ll –ly –lS

LIBES =

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

target1 [target2 . . .] :[:] [dependent1 . . .] [; commands] [# . . .]

[*(tab)* commands] [# . . .]

. . .

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.

2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the "–i" flags has been specified on the *make* command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $@ is set to the name of the file to be "made". $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $ is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, *make* prints a message and stops.

4.4 Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

make [flags] [macro definitions] [targets]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

–i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.

–s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.

–r Do not use the built-in rules.

–n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.

–t Touch the target files (causing them to be up to date) rather than issue the usual commands.

–q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up to date.

–p Print out the complete set of macro definitions and target descriptions

–d Debug mode. Print out detailed information on files and times examined.

–f Description file name. The next argument is assumed to be the name of a description file. A file name of "–" denotes the standard input. If there are no "–f" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

4.5 Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

.o Object file

.c C source file

.e Efl source file

.r Ratfor source file

.f Fortran source file

.s Assembler source file

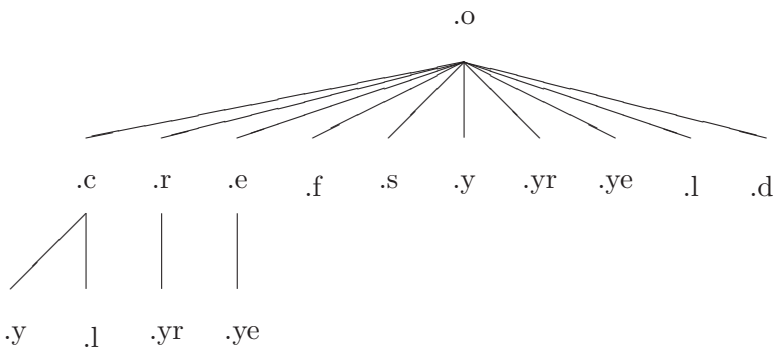
.y Yacc-C source grammar

.yr Yacc-Ratfor source grammar

.ye Yacc-Efl source grammar

.l Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

make CC=newcc

will cause the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

make "CFLAGS= O"

causes the optimizing C compiler to be used.

4.6 Example

As an example of the use of *make,* we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

P = und –3 | opr –r2        # send to GCOS to be printed
```

```
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.c gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= –lS
LINT = lint –p
CFLAGS = –O

make:  $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) –o make
        size make

$(OBJECTS):  defs
gram.o: lex.c

cleanup:
        rm *.o gram.c
        du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print:  $(FILES) # print recently changed files
        pr $? | $P
        touch print

test:
        make –dp | grep –v TIME >1zap
        /usr/bin/make –dp | grep –v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint :  dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc  –c version.c
cc  –c main.c
cc  –c doname.c
cc  –c misc.c
cc  –c files.c
cc  –c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  –c gram.c
cc  version.o main.o doname.o misc.o files.o dosys.o gram.o –lS –o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr –sp"
```

or

```
make print "P=  cat >zap"
```

4.7 Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the "–n" option is very useful. The command

```
make –n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the "–t" (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

make –ts

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag ("–d") causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

4.8 Acknowledgments

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

References

1. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler", Bell Laboratories Computing Science Technical Report #32, July 1978.

2. M. E. Lesk, "Lex — A Lexical Analyzer Generator", Computing Science Technical Report #39, October 1975.

A Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the "–r" flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a "*.r*" file to a "*.o*" file is thus "*.r.o*". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro $ is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ".SUFFIXES" in his own description file; the dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
       $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
       $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
       $(AS) -o $@ $<
.y.o :
       $(YACC) $(YFLAGS) $<
       $(CC) $(CFLAGS) -c y.tab.c
```

```
	rm y.tab.c
	mv y.tab.o $@
.y.c :
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@
```

5 RCS and Configuration Control

This section is an adaptation of a document entitled "Using RCS for Configuration Control of Large Projects" written by Thomas Mackey (tomm@voodoo.ca.boeing.com) and is taken with thanks to the author.

5.1 Introduction

Managing a large development effort is a difficult task, so any tool that makes the job easier is welcome. This paper describes the use of RCS for managing revision control of a large software project.

As with any other task, one tool alone is not usually sufficient to get the job done. RCS is no exception. RCS is but one of several tools and techniques that should be used in managing system development. The UNIX make(1) utility is invaluable for controlling the compilation and linking process; any and all dependencies are handled automatically once the dependency rules are defined. These rules can be built by hand, or by using the depends(1) utility if it available on your system. The power of make cannot be overstated. It can be used as a comprehensive build manager in its own right, controlling the building of the application libraries, the application itself, the documentation, and distribution of all deliverables. Another tool available for use by the development team is the logical separation of source code modules through effective use of the directory structure.

Other tools that make the development of UNIX applications smoother include ctags(1), grep(1) and its cousins, the file processing tool awk(1), and sed(1) the stream editing utility.

One of the biggest strengths of UNIX as a development environment is its portability. In order to make your application as portable as possible, it is recommended that you stick with standard UNIX utilities in the development process. Source that is maintained under RCS can be moved to new development platforms with a minimum of difficulty, and makefiles can be used to build the application tailored to the platform that it is being built on.

The power and simplicity of Bourne shell scripts should not be discounted. They cannot be beat for simple tasks that do not need to be done very often. They are usually more easily understood than C programs and can be developed in a fraction of the time. The Bourne shell is available on all UNIX systems, so shell scripts are very portable. Writing shell scripts is a skill that every developer should be comfortable with.

5.1.1 RCS Man Pages

This paper cannot hope to give the reader all the information about RCS. For a complete list of the various RCS capabilities, please read the man pages which should be available on your system. At the end of most of the man pages there will be references to other related man pages.

Here is a list of the commands and man pages which make up RCS:

- co (1)
- ci (1)

- ident (1)
- rcs (1)
- rcsdiff (1)
- rcsintro (1) <— read this one first
- rcsmerge (1)
- rlog (1)
- rcsfile (5)
- sccstorcs (8) (may not be available on all systems)

5.2 Source Directory Structure

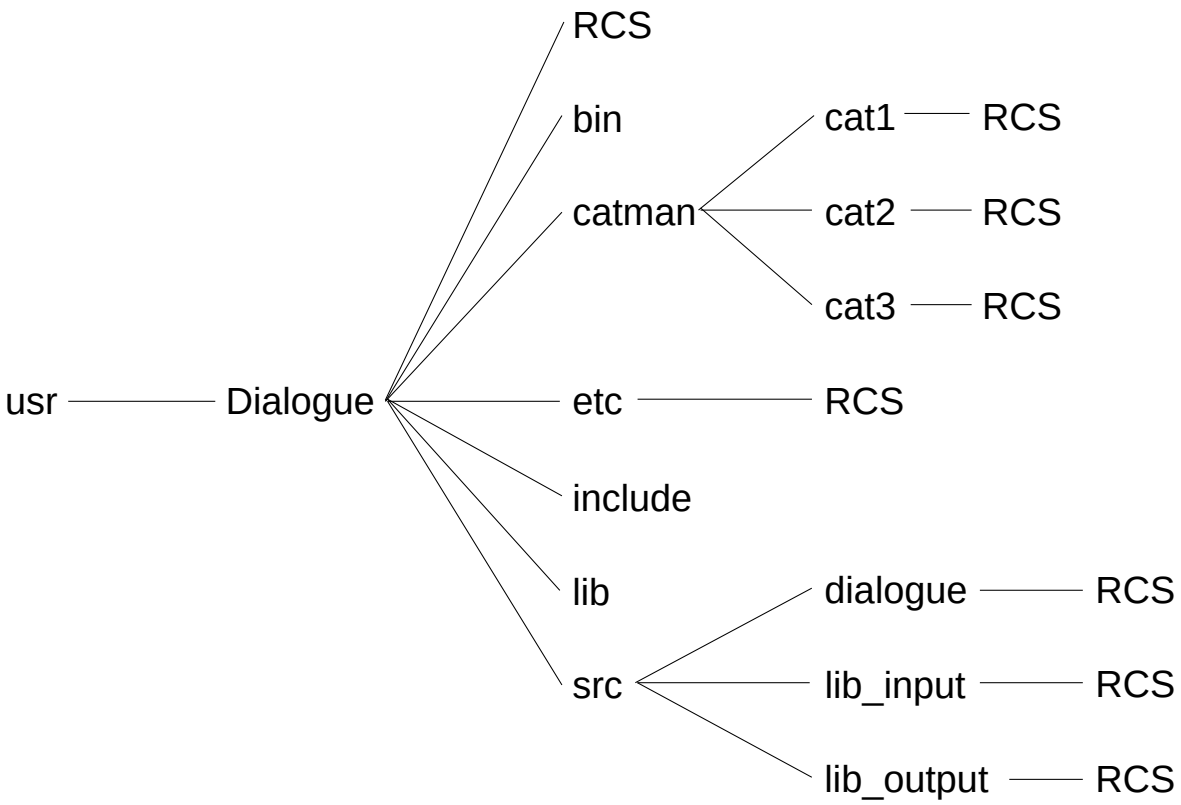
As a project grows and the number of source files increases, it becomes important to isolate the effects of changes; it is wasteful of time and computer resources to recompile large amounts of code for a small change. This is one of the reasons for breaking code into separate source modules in the first place. It is natural, then, to further group related source modules into libraries which are precompiled and contain information that enables the linking pass to efficiently link the library with other object modules. These related source modules, which can be precompiled into a library archive, should reside in separate subdirectories. The archiving utility ar(1) is used to build the library archives.

One machine is chosen as the source repository machine; this is where releases for beta test and final builds are done. If anyone uses this workstation as their development workstation, they must work in their own area; see the section titled "Individual Development Areas" for more information on this topic. For the purpose of entering the repository area, add an account with the name of the application; those in charge of doing official builds can then log into the repository workstation using the application account and do their work in a secure manner.

5.2.1 A Sample Application

For example, if you have an application that does a lot of input, a lot of output, and the I/O routines are not subject to a lot of change, then you might break up your application (lets call it "dialogue") into an input library, output library, and the main application body. Further, since the include files for the libraries will probably be utilized by the body of the application, and the precompiled library archives need a place to live, directories should be provided for the stable include files and stable archives. After adding directories for the binary, other related stuff (etc), man pages, and the RCS directories, the repository directory on the archive machine will look like this:

Directory tree of the "Dialogue" account on machine "bigbro"



Ideally, this directory structure should contain everything required for the "dialogue" utility. The top level directory (Dialogue) should contain a Makefile that will recursively enter catman, etc, and src and execute any makefile in those directories. It is kept under RCS control in Dialogue/RCS. Here also are any shell configuration files (.alias, .cshrc, .profile, etc.) that the users of the Dialogue account may need.

When the "dialogue" executable is created, it is copied into the Dialogue/bin directory. This is the executable that is used for beta testing by the other users, and is the one that is publicly available to other users when they log onto the development system. If the application consists of more than one binary, they all get stored in Dialogue/bin, as do any executable shell scripts. If there are any shell scripts, they would be developed, under RCS control, of course, in Dialogue/scripts or Dialogue/src/scripts.

The directories under Dialogue/catman follow the same pattern as the UNIX man page directories. Man pages for utilities that can be executed at the shell level (as our "dialogue" utility) are kept in cat1. Man pages for kernel level subroutines (or, in our case, subroutines that are a part of "dialogue") are kept in cat2. And man pages for subroutines that are a part of the libraries (lib_input and lib_output) are kept in cat3.

If there are any configuration files that "dialogue" needs to read, they should be in Dialogue/etc, under RCS control, if appropriate.

If there are any include files from the library directories that will be used by modules in the "dialogue" source files, then those include files should be copied into the Dialogue/include directory. This is analogous to including "curses.h" or "math.h" when using the UNIX curses or math libraries. The Makefile for the libraries, as part of the "install" target, should do the copying.

Also as part of the "install" target, the library archives should be copied to Dialogue/lib so that other developers can access them during the compile and link process.

The Dialogue/src subdirectory contains subdirectories for all source code. Each library has its own subdirectory, as does the main body of application code. Each of these sub-directories should contain a file containing a list of all the source files required to build that directory's deliverable. If you call that list "Filelist", then when you need to build the deliverable you would go through the following steps:

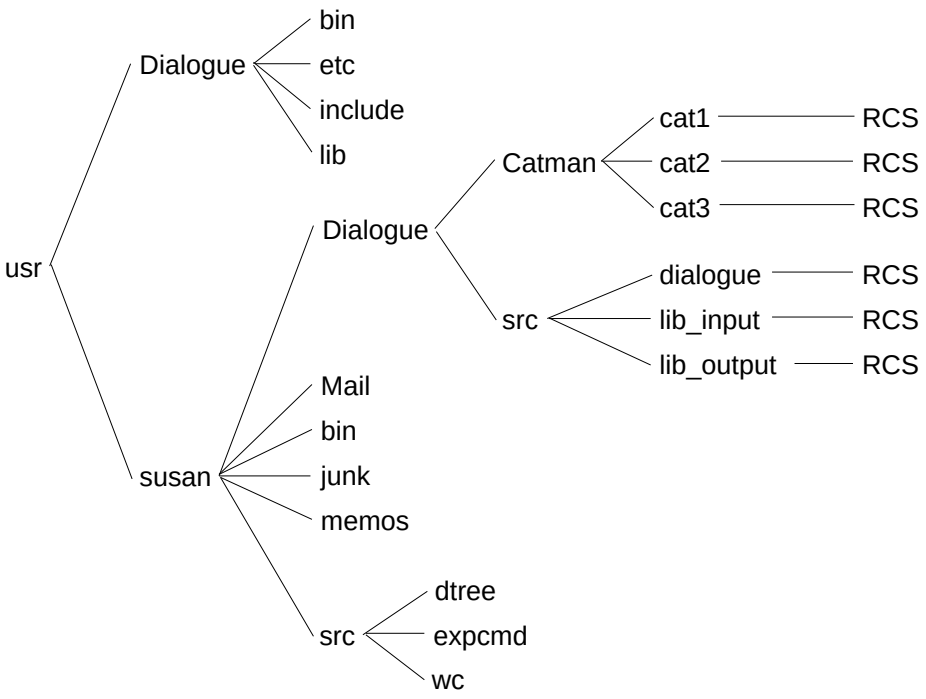
```
% co Filelist  \newline
% co ‘cat Filelist‘ \newline
% make
```

5.3 Individual Development Areas

Each developer should do their work in their own area. Depending on what they are working on, they may choose to duplicate the repository directory structure in their own area, or just the required pieces. Usually, the "bin" subdirectory is not needed, nor is the "etc", as these are most likely present on their workstation already as a necessary part of installing the application. This will be illustrated later on.

Lets say that we have a developer "susan" working on machine "odie". The workstation will have the application installed in /usr/Dialogue, the binary in /usr/Dialogue/bin, and special stuff in /usr/Dialogue/etc. The include and lib subdirectories will probably appear on developers' machines, but most likely will not be needed on production machines. Then if susan's account is /usr/susan the tree for the development of the "dialogue" project would appear in her directory like this:

Directory trees of /usr/Dialogue and /usr/susan on machine "odie"



The /usr file system on "bigbro" is mounted via NFS or similar mechanism on "odie":

```
% hostname
odie
% df
Filesystem     Type  blocks     use   avail %use  Mounted on
/dev/root       efs   30780   16218   14562  53%  /
/dev/usr        efs  472500  384618   87882  81%  /usr
bigbro:/usr     nfs  486588  256098  230490  53%  /bigbro/usr
%
```

The RCS subdirectories in the src and catman subdirectories are symbolic links to the true RCS directories which exist on "bigbro". These links are created like this:

```
% pwd
/usr/susan/Dialogue/src/dialogue
% ln -s /bigbro/usr/Dialogue/src/dialogue/RCS ./RCS
% ls -lF RCS
l---------   1 susan  RCS@ -> /bigbro/usr/Dialogue/src/dialogue/RCS
%
```

If a developer has the archive machine as their home machine, their RCS subdirectories are also symbolic links to the true RCS subdirectories on the archive machine.

5.4 Normal Activities

This section describes the activities normally associated with developing software under RCS control. Source files have to be created and edited. Editing a locked file is the only assurance that someone else is not modifying that file at the same time you are. When the source files are ready, they are compiled, using the "make" utility to control the compile and link process. There will be times that more than one developer needs to change the same source file; there are methods that make this as safe as possible. It is sometimes necessary to make a special version of the application, maybe for testing purposes, or for debugging. There are ways to build you own special versions of the source files so as not to disturb the other developers.

5.4.1 Editing a File

RCS allows only one person at a time to have a particular file checked out with a lock. To make a change to the file "main.c" in the dialogue subdirectory, susan would do the following:

```
% co -l main.c
RCS/main.c,v  -->  main.c
revision 2.2 (locked)
done
% ex main.c
"main.c" 32 lines, 1326 characters
```

```
:3l
*  FILE:  source_file.c$
:3:s/ILE/ile/
*  File:  source_file.c
:wq
"main.c" 32 lines, 1326 characters
% ci -u main.c
RCS/main.c,v  <--  main.c
new revision: 2.3; previous revision: 2.2
enter log message:
(terminate with ^D or single '.')
>> de-capitalized the word FILE (now File) in file prolog
>> .
done
%
```

Susan first checked out main.c with a lock. This ensures that only she can make changes to the file. Using a line-oriented editor "ex" (Blech! Used here for demonstration purposes, only!), she listed line 3 and then modified line 3. Ex printed out the modified line, susan saved the changes and checked the file back in. The -u option tells the RCS system to save a copy (unlocked) in her directory after the changes are checked in. The RCS system prompts for a log message, and susan tells briefly what changes she made.

If, while susan has main.c checked out with a lock, user "john" attempts to also check it out with a lock, he would see this:

```
# co -l main.c
RCS/main.c,v  -->  main.c
co error: revision 2.2 of main.c already locked by susan
```

User john must go ask user susan how long she will have it checked out, whether she can add his changes, too (if they are very simple), or decide if he should wait or make changes to his own version to be merged in later (see the section titled "Several Developers Working on One File").

5.4.2 Running Make

An entire book could be written about the make utility. In fact, one has. The following references are recommended:

- Make
  Programmers Guide in your manual set
- Make(1)
  Man page in your manual set
- Managing Projects with Make
  (A Nutshell Handbook)

Steve Talbot
O'Reilly and Associates, Inc.
981 Chestnut Street
Newton, MA 02164
1-800-338-NUTS

LocalImakefile

By adding the line "include Localmakefile" into your Makefile, the action of the Makefile can be changed without modifying the Makefile itself. This way, all developers can make use of a standard Makefile, and modify its actions to suit their individual needs by changing their Localmakefile.

Here is an example of a Localmakefile:

```
UI_INCDIR = /usr/tomm/bogart/include
UI_LIBDIR = /usr/tomm/bogart/lib
CFLAGS = $(NORMFLAGS) $(SPECIALFLAGS) \
         -I$(INCDIR) -I$(UI_INCDIR) -I$(DL_INCDIR)
SPECIALFLAGS = -g -DRASTER -DNORMALPLANES -DSGI -DFOURSITE
BCSLIBS = $(DL_LIBDIR)/libdl.a $(UI_LIBDIR)/libuidbx.a
```

This Localmakefile changes the definitions of five of the macros defined in the application Makefile. The first two cause personal directories to be named as the repositories of the user interface include files and user interface library. The redefinition of CFLAGS causes the include directories to be searched in a different order, ensuring that the personal copies of the user interface include files will be used. The SPECIALFLAGS definition adds the "g" flag, and the redefinition of BCSLIBS causes a debugging version of the user interface library to be linked in.

5.4.3 Creating and Using Your Own Libraries

If developers need to build special versions of the application libraries, they need only check out the files (without a lock), and tweak their Localmakefile so that their special version is placed somewhere in their personal directory tree. Then they change the protections of the required files using chmod(1) to make them editable, make the required changes, and run make. Their actions will affect no one else, so several developers can have several special versions. Of course, these special versions are for testing or debugging only; the "official" libraries must be used to build the actual releases of the system.

5.4.4 Making a Personal Special Version

To build a special version of the application using an individual special copy of a library, the developer must change their Localmakefile so that the special library is used instead of the normal library. Note the example in the section titled "Localmakefile". Special versions of the application are invaluable for those nasty debugging jobs that require the use of dbx(1)

or lots of printf()'s. A developer can make a particular source file writable with chmod(1), add debugging stuff or link with a special debugging library, and only check the offending file out with a lock when the problem is narrowed down. It must be understood that this particular approach can be dangerous if the developers lose track of what they are doing. If, for example, they make a lot of changes to an unlocked file and fail to make the same changes to the file once they lock it, then there is a chance that they could fail to completely fix a bug. Another possibility is that another developer might lock the file before the first developer does. Checking out a file with a lock to make any changes is still the best way to modify source.

5.5 Starting a New Branch

There comes a time in the development cycle when changes to the code must be stopped, a beta release built, and some place be created where on-going development for the next release can continue. RCS handles this situation easily by considering each major branch of the RCS tree as a separate revision. To demonstrate, consider this list of lines as it is checked in initially, then checked out and back in on a new branch:

```
(Check in the file "list", and force RCS to use 1.0 as the
initial revision number; default initial revision is 1.1)

% ci -u -r1.0 list
RCS/list,v  <--  list
initial revision: 1.0
enter description, terminated with ^D or '.':
NOTE: This is NOT the log message!
>> list of lines for rcs demo
>> .
done

(Show what's in the file so far)

% cat list
line one
line two
line three
%

(Checkout the file and start a new revision branch,
 no changes made to the file)

% co -l list
RCS/list,v  -->  list
revision 1.0 (locked)
done
% ci -u -r2.0 list
RCS/list,v  <--  list
new revision: 2.0; previous revision: 1.0
```

```
File list is unchanged with respect to revision 1.0
checkin anyway? [ny](n): y
enter log message:
(terminate with ^D or single '.')
>> starting a new branch for on going development
>> .
done
%
```

5.5.1 Assigning a Symbolic Name

Now that there is more than one revision to worry about, you will want to assign symbolic names to make retrieval of a particular version easier. A symbolic name may be assigned to a particular RCS revision, or to some branch in the RCS tree.

```
(Assign symbolic names for the old and new branches)

% rcs -Nrev1:1 list
RCS file: RCS/list,v
done
% rcs -Nrev2:2 list
RCS file: RCS/list,v
done
%

(See what we have so far)

% rlog -h list
RCS file:        RCS/list,v;   Working file:    list
head:            2.0
locks:           ;  strict
access list:
symbolic names:  rev2: 2;  rev1: 1;
comment leader:  "# "
total revisions: 2;
=============================================================
%
```

The two symbolic names point to two branches of the revision tree. Rev1 points to the base of branch 1. It will retrieve the latest revision on the main branch; rev2 will do the same for branch 2. If there are any side branches (1.3.x.y), they will be ignored when retrieving using the symbolic names as defined above. In order for a symbolic name be used to retrieve the latest version from some side branch, the symbolic name must be attached to the base of that branch. For example, if you have a revision 1.3 which you have modified for some special purpose, like the addition of debugging code, or a trial implementation of some special feature, then the revisions will be numbered 1.3.1.x. There can be several side branches which trace their "roots" back to revision 1.3, so you could also have side branches of revisions

numbered 1.3.2.x, 1.3.3.x, and so forth. If you want to assign a symbolic name to one of these side branches such that it will always retrieve the latest revision on that branch, you need to assign it to exactly one of the (possibly many) side branches. If 1.3.2.x contains trial improvements, and you want to use the symbolic name "trial", then the symbolic name would be assigned thus:

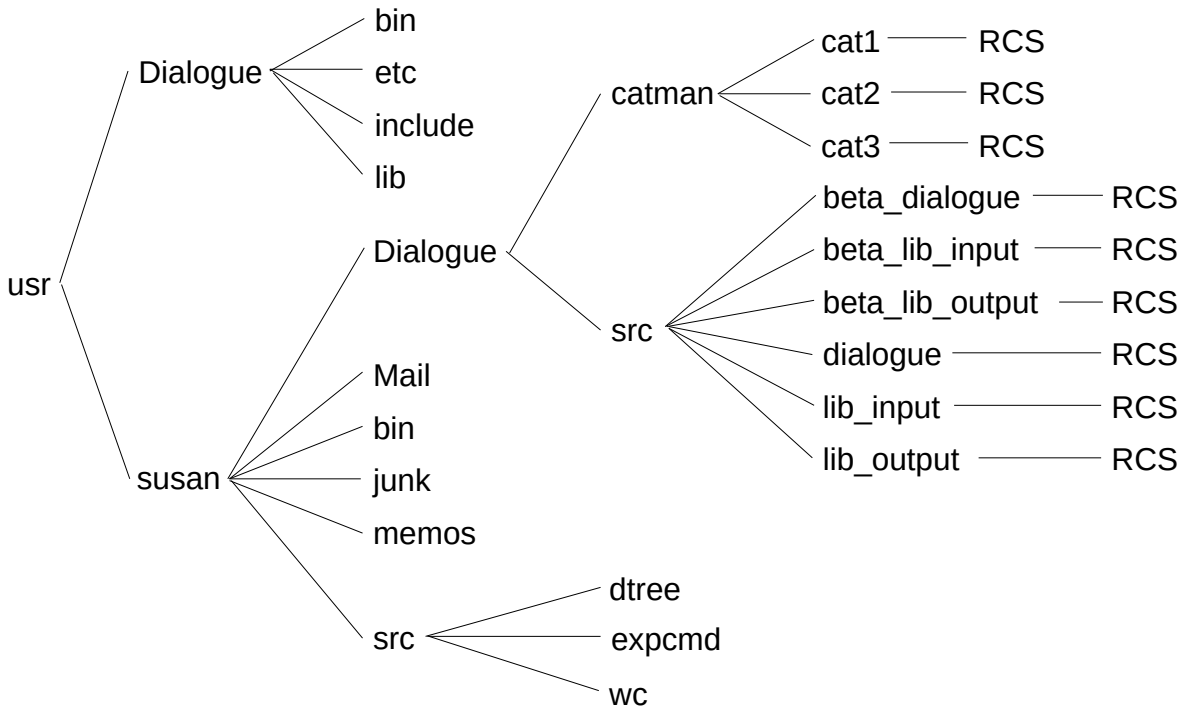
```
rcs -Ntrial:1.3.2 filename
```

5.6 Maintaining Multiple Releases

As a direct consequence of the truism "There is always one more bug", there will be problems in the beta release which must be fixed. In terms of where a bug is fixed, there are two ways that the fix must be merged into the revision branches. If the bug is fixed in the on-going development branch, it must be merged back into the beta branch. On the other hand, if the bug is fixed in the beta branch, then the change must be merged forward into the on-going development branch.

During the period of time when the final few bugs are being stamped out in the beta code, and on going development is continuing, it will be necessary for some of the developers to do work on both branches of the development tree. The easiest way to do this is to build a second set of directories which match the major development source tree. Going back to the application "dialogue", susan would make a second set of directories if she needed to fix bugs in both libraries and the main application. She would use this second set of directories to build and test a new beta version:

Directory trees of /usr/Dialogue and /usr/susan on machine "odie" showing the second set of source subdirectories which are used to build and test a new beta release after on going development has resumed.

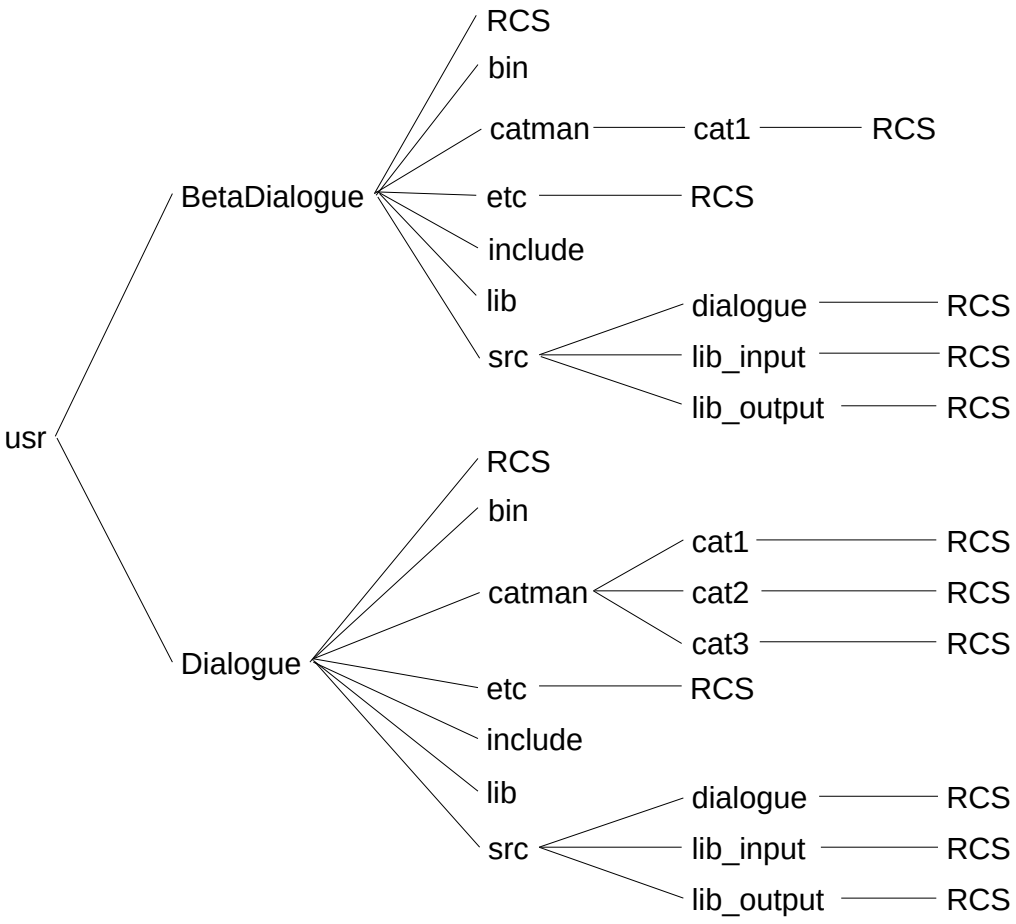


Note that the RCS "subdirectories" are actually symbolic links to the same true RCS subdirectories which exist on "bigbro". That is, src/beta_dialogue/RCS points to the same subdirectory as src/dialogue/RCS; that true subdirectory is /usr/Dialogue/src/dialogue/RCS on bigbro.

Once there is more than one release to take care of, there must be a set of directories for each of those releases on the archive machine. Normally, there will only be two such releases: the release that is in beta test or production, and the version that is in on going development. It works well to keep the original set of directories with their true RCS subdirectories for on going development, and a second set of directories with symbolic links to the RCS subdirectories for the release that is in beta test and eventual production. Since the beta directory is primarily for building a release for production, it does not have to contain all the subdirectories that the developers might need.

Note here that only the man pages for section 1 commands need be available for beta and production release. The object is to minimize the amount of work you need to do while delivering the highest quality system.

Directory tree of the "Dialogue" account on machine "bigbro" showing a second set of directories parallel to the original set and used to build the beta/production release of the application. All RCS subdirectories in the BetaDialogue directory tree are actually symbolic links to the true RCS subdirectories in the Dialogue directory tree.



5.6.1 Using RCSMERGE

There are three scenarios in a normal development cycle that require the use of rcsmerge. One is when two or more developers need to work on the same file at the same time. The other two deal with making parallel changes to several releases. Rcsmerge always uses three revisions of a file to do its work. It merges the differences between two revisions of a file into a third file. The output can either overwrite the third file, or can be written on standard output so the results can be previewed.

5.6.2 Fixing Bugs in the Beta Production System

This section demonstrates how to fix a bug in the beta code and then duplicate that bug fix in the on going development code. It also demonstrates how to use RCS to set symbolic names for particular branches of the revision tree, and how to change the point that a symbolic name refers to. The fixing of beta bugs takes place in the set of beta directories.

```
(Check out the revision of "list" that is used in rev 1)

% co -l -rrev1 list
RCS/list,v  -->  list
revision 1.0 (locked)
done

(Edit "list", changing second line)

% ex list
"list" 3 lines, 29 characters
:2:s/two/II/
line II
:1,$l
line one$
line II$
line three$
:wq
"list" 3 lines, 28 characters

(Verify the differences using rcsdiff)

% rcsdiff list
RCS file: RCS/list,v
retrieving revision 1.0
diff  -r1.0 list
2c2
< line two
---
> line II

(Check "list" back in)
```

```
% ci -u list
RCS/list,v  <--  list
new revision: 1.0.1.1; previous revision: 1.0
enter log message:
(terminate with ^D or single '.')
>> changed from english word to roman numeral in second line
>> .
done
```

Re-assigning Symbolic Names

At this point, if you try to use the symbolic name "rev1" to retrieve the revision of "list" that coincides with revision one of the system, RCS would give you revision 1.0, not 1.0.1.1. In order for RCS to follow the correct branch of the revision tree, you must tell RCS that you want the new side branch of 1.0 to represent revision one of your system. You do this by re-assigning the symbolic name "rev1" to the 1.0.1 branch of the tree. Since 1.0.1.1 is the highest leaf on the branch, you will get the correct revision. If there are further modifications to "list", they will be stored as 1.0.1.2, 1.0.1.3, and so forth, so 1.0.1 is sufficient to identify the correct branch.

```
(Change which revision entity the symbolic name points to)

% rcs -Nrev1:1.0.1 list
RCS file: RCS/list,v
done

(See what we have so far)

% rlog -h list
RCS file:        RCS/list,v;   Working file:    list
head:            2.0
locks:           ;  strict
access list:
symbolic names:  rev2: 2;  rev1: 1.0.1;
comment leader:  "# "
total revisions: 3;
=============================================
%
```

Merging Bug Fixes

While the application was in beta test, there have been some changes made in "list" as part of the on going development. Two more lines have been added. Now you need to apply the same fix to the development code as was made in the beta code. The first step is to check out the latest development version with a lock. Then use rcsmerge to merge the bug fix into the new version and redirect the result to a temporary file. After checking that the differences are

correct, move the temporary file back to "list" and check in the new version. These changes are made while working in your on going development directory set.

```
(Check out latest version and see what it looks like)

\% co -l list
RCS/list,v  -->  list
revision 2.1 (locked)
done
%
% cat list
line one
line two
line three
line 4
line 5
%

(Merge the differences using rcsmerge and then verify)

% rcsmerge -p -r1.0 -rrev1 list > list.merged
RCS file: RCS/list,v
retrieving revision 1.0
retrieving revision 1.0.1.1
Merging differences between 1.0 and 1.0.1.1 into list; result to stdout
% cat list.merged
line one
line II
line three
line 4
line 5
%
% diff list list.merged
2c2
< line two
---
> line II
%

(Move temporary file to real file and check it in)

% mv list.merged list
% ci -u list
RCS/list,v  <--  list
new revision: 2.2; previous revision: 2.1
enter log message:
(terminate with ^D or single '.')
>> changed from english word to roman numeral in second line
```

```
>> (same as 1.0.1.1)
>> .
done
%
```

5.6.3 Fixing Bugs in the Development System

In the example that follows, lines 4 and 5 represent new code added during on going development. Line TWO_DOT_FIVE represents a fix for a bug discovered while testing the new features. It is decided that this bug is serious and must be fixed in the production system as well. You do not want lines 4 and 5 merged back into the production code, only the bug fix. This minimizes the amount of re-testing that must be done before the production release can be replaced by the revision with the bug fix.

```
This is what the file looks like after fixing
a subtle but dangerous bug:

% cat list
line one
line II
line TWO_DOT_FIVE
line three
line 4
line 5

Check "list" back into the development branch

% ci -u list
RCS/list,v  <--  list
new revision: 2.3; previous revision: 2.2
enter log message:
(terminate with ^D or single '.')
>> fixed subtle bug by adding line TWO_DOT_FIVE
>> .
done

Check out the latest production source (after changing to
a directory set up for working on the production release)

% co -l -rrev1 list
RCS/list,v  -->  list
revision 1.0.1.1 (locked)
done

Merge the differences using rcsmerge and then verify

% rcsmerge -p -r2.2 -r2.3 list > list.merged
RCS file: RCS/list,v
```

```
retrieving revision 2.2
retrieving revision 2.3
Merging differences between 2.2 and 2.3 into list; result to stdout
% diff list list.merged
2a3
> line TWO_DOT_FIVE

Move temporary file to real file, take one more look at it
to satisfy paranoia, and then check it in.  Note that no
reassignment of the symbolic name is necessary, as this
new revision is automatically added to the 1.0.1 branch.

% mv list.merged list
% cat list
line one
line II
line TWO_DOT_FIVE
line three
% ci -u list
RCS/list,v  <--  list
new revision: 1.0.1.2; previous revision: 1.0.1.1
enter log message:
(terminate with ^D or single '.')
>> fixed subtle bug by adding line TWO_DOT_FIVE
>> (same as 2.3)
>> .
done
%
```

As with the rest of the examples, the obvious steps of compilation and careful testing of the changes have been omitted in order to focus on the RCS issues.

5.6.4 Several Developers Working on One File

If several developers need to work on the same file at the same time, then one of them will have to make "local" changes to an unlocked revision of the file, then merge the other person's changes into their copy after the other person checks it in.

User john checks out "list" with a lock

```
% co -l list
RCS/list,v  -->  list
revision 2.3 (locked)
done
%
```

User susan tries to check out the same file with a lock and is given the bad news.

```
% co -l list
```

```
RCS/list,v  -->  list
co error: revision 2.3 of list already locked by john
%
So OK: she just gets the latest revision and makes it
writable in her own directory.
% co list
RCS/list,v  -->  list
revision 2.3
done
% chmod 644 list
%
```

She edits the file, modifying one line and adding two more.

```
% ex list
"list" 6 lines, 60 characters
:1,$l
line one$
line II$
line TWO_DOT_FIVE$
line three$
line 4$
line 5$
:$
line 5$
:s/5/FIVE //susan/
line FIVE //susan
:a
new line 5.1 //susan
new line 5.2 //susan
:wq
"list" 8 lines, 113 characters
%
```

The file now looks like this in susan's directory:

```
% cat list
line one
line II
line TWO_DOT_FIVE
line three
line 4
line FIVE //susan
new line 5.1 //susan
new line 5.2 //susan
%
```

Meanwhile, john has checked in a new revision 2.4 which looks like this:

```
line one
line II
line TWO_DOT_FIVE
line three
line 4
line 5
new stuff 6 // john
new stuff 7 // john
```

Susan does an rcsmerge to merge the changes between revision 2.3 and 2.4 into her version (the unlocked copy of "list" in her directory). Note the overlap because she changed the line "line 5" to "line FIVE" and john did not. She will have to decide whether or not this will affect john's change. Looks like its "yell over the cube" time!

Once rcsmerge creates the temporary file list.merged, susan edits it to determine where the overlap is and what needs to be done to combine their work.

```
rcsmerge -p -r2.3 -r2.4 list > list.merged
RCS file: RCS/list,v
retrieving revision 2.3
retrieving revision 2.4
Merging differences between 2.3 and 2.4 into list; result to stdout
Warning: 1 overlaps during merge.
%
% ex list.merged
"list.merged" 14 lines, 193 characters
:1,$l
line one$
line II$
line TWO_DOT_FIVE$
line three$
line 4$
<<<<<<< list$
line FIVE //susan$
new line 5.1 //susan$
new line 5.2 //susan$
=======$
line 5$
new stuff 6 // john$
new stuff 7 // john$
>>>>>>> 2.4$
:14d
new stuff 7 // john
:10,11d
new stuff 6 // john
:6d
line FIVE //susan
:1,$l
line one$
```

```
line II$
line TWO_DOT_FIVE$
line three$
line 4$
line FIVE //susan$
new line 5.1 //susan$
new line 5.2 //susan$
new stuff 6 // john$
new stuff 7 // john$
:wq
"list.merged" 10 lines, 153 characters
```

The rcsmerge utility surrounds the overlaps with <<<<<< and ======; susan decides that her change to "line 5" is OK, so she removes the overlap delimiting lines and "line 5", then lists out and saves the file.

Here she checks out revision 2.4 with a lock and moves the temporary file onto the real file. Then she does an rcsdiff to make sure everything is still ok.

```
% co -l list
RCS/list,v  -->  list
revision 2.4 (locked)
writable list exists; overwrite? [ny](n): y
done
% mv list.merged list
%
% rcsdiff list
RCS file: RCS/list,v
retrieving revision 2.4
diff  -r2.4 list
6c6,8
< line 5
---
> line FIVE //susan
> new line 5.1 //susan
> new line 5.2 //susan
```

This is what the file looks like after john and susan have finished their changes. Susan checks it in and enters the log message. That's all there is to it!

```
% cat list
line one
line II
line TWO_DOT_FIVE
line three
line 4
line FIVE //susan
new line 5.1 //susan
```

```
new line 5.2 //susan
new stuff 6 // john
new stuff 7 // john
%
% ci -u list
RCS/list,v  <--  list
new revision: 2.5; previous revision: 2.4
enter log message:
(terminate with ^D or single '.')
>> added new lines 5.1 and 5.2, and modified line 5
>> .
done
%
```

5.7 Building an Arbitrary Release

There are several reasons for building some arbitrary release. You might get new programmers into the development team and want to show them the evolution of the application through major releases. You might want to go back to a previous date to see when a particular bug or behavior first showed up. You might have to roll back the operating system of one system and need to build the last release that worked with that release of the operating system. You may never want to do it.... Who knows?

The first thing to do is to create a parallel set of directories similar to what you use to build the beta/production release, and create the symbolic links to all the RCS subdirectories. Then check out the appropriate revision of each source and run "make". RCS allows the retrieval by major branch number, symbolic name, date (down to hour and minute, if necessary), author, or "state" if you make use of the standard experimental, stable, and released states).

There is no magic involved in any of this. If you build an early release, be aware that configuration files, format of application data files, and window system interaction may all have changed. It is up to the developers of the system to keep track of any outside dependencies; RCS simply gives you the ability to accurately retrieve former revisions of the source files.

5.7.1 Some Examples of Arbitrary Builds

Following are examples of using different criteria for extracting revisions from the revision branches. There are many other possibilities; see the man page on co(1) for more ideas.

```
Building Application as it Existed at Some Date:

% co -d"Feb 12, 1989" Filelist
% co -d"Feb 12, 1989" `cat Filelist`
% make
%

If the latest revision branch does not go back to 2/12/89,
you will have to add the appropriate branch number:
```

```
% co -d"Feb 12, 1989" -r2 Filelist
%

Building Application From Last ‘‘stable" Sources

% co -sStab Filelist
% co -sStab ‘cat Filelist‘
% make
%

Building Initial ‘‘alpha" Release of Application

% co -ralpha Filelist
% co -ralpha ‘cat Filelist‘
% make
%
```

6 Concurrent Versions System (CVS)

This section is an adaptation of a document written by Gray Watson (gray.watson@antaire.com) and taken, with thanks to the author, from electronic news.

6.1 Introduction

CVS is a system that lets groups of people work simultaneously on groups of files (for instance program sources).

It works by holding a central 'repository' of the most recent version of the files. You may at any time create a personal copy of these files by 'checking out' the files from the repository into one of your directories. If at a later date newer versions of the files are put in the repository, you can 'update' your copy.

You may edit your copy of the files freely. If new versions of the files have been put in the repository in the meantime, doing an update merges the changes in the central copy into your copy.

When you are satisfied with the changes you have made in your copy of the files, you can 'commit' them into the central repository.

When you are finally done with your personal copy of the files, you can release them and then remove them.

6.2 Basic Terms

Repository: The directory storing the master copies of the files. The main or master repository is a tree of directories.

Module: A specific directory in the main repository. Modules are defined in the cvs modules file.

RCS: Revision Control System. A lower-level set of utilities on which CVS is layered.

Check out: To make a copy of a file from its repository that can be worked on or examined.

Revision: A numerical tag identifying the version of a file.

6.3 Basic CVS Commands

Most of the below commands should be executing while in the directory you checked out. If you did a 'cvs checkout malloc' then you should be in the malloc sub-directory to execute most of these commands. 'cvs release' is different and must be executed from the directory above.

cvs checkout (or 'cvs co')

To make a local copy of a module's files from the repository execute 'cvs checkout module' where module is an entry in your modules file (see below). This will create a sub-directory module and check-out the files from the repository into the sub-directory for you to work on.

cvs update

To update your copy of a module with any changes from the central repository, execute 'cvs update'. This will tell you which files have been updated (their names are displayed with a U before them), and which have been modified by you and not yet committed (preceded by an M).

It can be that when you do an update, the changes in the central copy clash with changes you have made in your own copy. You will be warned of any files that contain clashes, the clashes will be marked in the file surrounded by lines of the form <<<< and >>>>. You have to resolve the clashes in your copy. After an update where there have been clashes, your original version of the file is saved as .#file.version.

To lose your changes and go back to the version from the repository, delete the file and do an update.

cvs commit

When you think your files are ready to be merged back into the repository for the rest of your developers to see, execute 'cvs commit'. You will be put in an editor to make a message that describes the changes that you have made (for future reference). Your changes will then be added to the central copy.

When you do a commit, if you haven't updated to the most recent version of the files, cvs tells you this; then you have to first update, resolve any possible clashes, and then redo the commit.

'cvs add' and 'cvs remove'

It can be that the changes you want to make involve a completely new file, or removing an existing one. The commands to use here are:

cvs add <filename>
cvs remove <filename>

You still have to do a commit after these commands to make the additions and removes actually take affect. You may make any number of new files in your copy of the repository, but they will not be committed to the central copy unless you do a 'cvs add'.

cvs release

When you are done with your local copy of the files for the time being and want to remove your local copy use 'cvs release module'. This must be done in the directory about the module sub-directory you which to release.

If you wish to have CVS also remove the module sub-directory and your local copy of the files then your 'cvs release –d module'.

Take your time here. CVS will inform you of files that may have changed or it does not know about (watch for the ? lines) and then with ask you to confirm this action. Make sure you want to do this.

cvs log

To see the commit messages for files, and who made them, use:

cvs log [filenames]

cvs diff

To see the differences between your version of the files and the version in the repository do:

cvs diff [filenames]

cvs tag

One of the exciting features of CVS is its ability to mark all the files in a module at once with a symbolic name. You can say 'this copy of my files is version 3'. And then later say 'this file I am working on looked better in version 3 so check out the copy that I marked as version 3.'

Use 'cvs tag' to tag the version of the files that you have checked out. You can then at a later date retrieve this version of the files with the tag.

cvs tag tag–name [filenames]

Later you can do:

cvs co –r tag–name module

cvs rtag

Like tag, rtag marks the current versions of files but it does not work on your local copies but on the files in the repository. To tag all my libraries with a version name I can do:

cvs tag LIBRARY_2_0 lib

This is one of the most use features of cvs (IMHO). Use this feature if you about to release a copy of the files to the outside world or just want to mark a point in the developmental progression of the files.

cvs history

To find out information about your cvs repositories use the 'cvs history' command. By default history will show you all the entries that correspond to you. Use the –a option to show information about everyone.

cvs history –a –o shows you (a)ll the checked (o)ut modules
cvs history –a –T reports (a)ll the r(T)ags for the modules (!!!)
cvs history –a –e reports (a)ll the information about (e)verything

6.4 Getting Started

Make sure all your developers have the CVSROOT environmental variable set to the directory that is to hold your main file repository (mine is set to /usr/src/master).

Run the cvsinit script that comes with cvs to initialise the repository tree.

Encourage all your developers to make a working directory where they will be working on the files (mine is ~/src/work).

- Edit the modules file to add the local "modules".

  Either cd to ${CVSROOT}/CVSROOT and 'co –l modules' and then edit it. Or better, cd to your working directory and do a 'cvs co modules'.

  NOTE: co is an alias for checkout

  Add your modules to the file. I add the below lines to my file:

```
# libraries
lib             antaire/lib

db              antaire/lib/db
dt              antaire/lib/dt
inc             antaire/lib/inc
lwp             antaire/lib/lwp
malloc          antaire/lib/malloc
inter           antaire/lib/inter
prt             antaire/lib/inter/prt
```

  The above entries now allow me to 'cvs co malloc' which will create a directory malloc where I am and check out the files from ${CVSROOT}/antaire/lib/db into that directory.

  'cvs co lib' will check out all my libraries and make a whole tree under lib: lib/db/*, lib/dt/*, lib/inc/*, etc

- Create a cvsignore file in ${CVSROOT}/CVSROOT:

  This file contains the local files that you want cvs to ignore. If you have standard temporary files, or log files, etc. that you would never want cvs to notice then you need to create this file.

  The first time you should go into the CVSROOT directory, edit the file and 'ci –u cvsignore' to check it in.

  You should apply the mkmodules.patch (included at the end of this file) and recompile and install the mkmodules file and add the below line to your modules file (see above) so you can use cvs to edit the file in the future.

      cvsignore –i mkmodules CVSROOT cvsignore

  I have in my file:

```
*.t
*.zip
MAKE.LOG
Makefile.dep
a.out
logfile
...
```

CVS ignores a number of common temp files (*~, #*, RCS, SCCS, etc..) automatically. (see cvs(5)).

WARNING: cvs is good at this. :-) any files in the cvsignore file will be ignored completely without a single warning.

- Add your files into their respective module directories:

  cd into your current directory which holds the files.

  Build clean/clobber and make sure that only the files you want to be checked into the repository are in the current directory.

  Execute:

  cvs import –m 'comment' repository vendortag releasetag

  The comment is for you to document the module

  The repository should be a path under ${CVSROOT}. My malloc library is checked into antaire/lib/malloc

  Vendortag is a "release tag" that the vendor assigned to the files. If you are the vendor then put whatever you want there: (PRT_INITIAL, MALLOC_1_01, etc);

  Releasetag is your local tag for this copy of the files. (PRT_1, malloc_1_01, etc);

6.5 Basic Usage

- cd to your work directory (I do 'cd ~/src/work')
- Execute 'cvs co module' where module is an entry from the modules file (see above):

  I do 'cvs co malloc' to get my malloc library. It will create the sub-directory malloc and will load the files into this new directory.

  Edit the files to your heart's content.

  If you add any new files to the directory that you want the repository to know about you need to do a

  cvs add file1 [file2 ...]

  If you remove any files you need to do a

  cvs remove file1 [file2...]

  If you rename you need to do a combination remove and then add.

  Execute 'cvs update' to pull in the changes from the repository that others made. It will resolve conflicts semi-automatically. It will tell you about the files it updates. U means updated, C means there was a conflict that it could not automatically resolve. You need to edit the file by hand, look for the <<<<< and >>>>> lines and figure out how the file should look.

  Execute 'cvs commit' inside the directory you checked out to apply your changes to the repository so others can use them (if they have the module in question checked out already, they need to do a 'cvs update' to see your changes).

- When you are done with the files (for the time being) you cd .. to the above directory and do a 'cvs release [–d] module–name' which will "check-in" the files. The optional –d will remove the directory and files from your work directory when it is done releasing them.

  The release command will inform you if your made modifications to the files and if there are files it doesn't know about that you may have forgotten to add. watch for '? file' lines printed. You may have to stop the release and commit or cvs add/remove the files.

  WARNING: release –d is unrecoverable. Make sure you take your time here.

  Don't worry, release should ask whether you really want to do this before doing anything.

6.6 General Help

All cvs commands take a –H option to give help:

- 'cvs –H' shows you the options and commands in cvs.
- 'cvs history –H' shows you the options for cvs history.

All the cvs commands mentioned also accept a flag '–n', that doesn't do the action, but lets you see what would happen. For instance, you can use 'cvs –n update' to see which files would be updated.

To get more information, see the manual page 'man cvs' for full (and much more complicated) details.

A basic knowledge of the Revision Control System (RCS) on which CVS is layered may also be of some assistance. see 'man co' or 'man ci' for more details.

6.7 RCS and CVS — what's the difference?

Collection of files

One of the strong points about CVS is that it does not only lets you retrieve old versions of specific files. You can collect some files (or directories of files) into "modules" and a lot of the CVS commands can operate on an entire module at once. The RCS history files of all modules are kept at a central place in the file system hierarchy. When someone wants to work an a certain module he just types "cvs checkout bugtrack" which causes the directory "bugtrack" to be created and populated with the files that make up the bugtrack project.

With "cvs tag bugtrack–1.0" you can give the symbolic tag "bugtrack–1.0" to all the versions of the file in the bugtrack module. Later on, you can do "cvs checkout –r bugtrack–1.0 bugtrack" to retrieve the files that make up the 1.0 release of bugtrack. You can even do things like "cvs diff –c –r bugtrack–1.0 – –r bugtrack–1.5" to get a context diff of all files that have changed between release 1.0 and release 1.5!

No locking

If you work in a group of programmers you have probably often wanted to edit the function foo() in bar.c, but Joe had locked bar.c because he is editing gazonk().

CVS does not lock files. Instead, both you and Joe can edit bar.c. The first one to check in it won't realize that the other have been editing it. (So if you are quicker than Joe you wont have any trouble at all). Poor Joe just have to do "cvs update bar.c" to merge in your changes in his copy of the file. As long as you are changing different sections of the file the merge is totally automatic. If you change the same lines you will have to resolve the conflicts manually.

Friendlier user interface

If you don't remember the syntax of "cvs diff" you just type "cvs –H diff" and you will get a short description of all the flags. Just "cvs –H" lists all the sub-commands. I find the commands less cryptic than the RCS equivalents. Compare "cvs checkout module" (which can be abbreviated to "cvs co module") with "co –l RCS/*,v" (or whatever it is you are supposed to say — it was a year since I used RCS seriously).

6.8 General Questions

When I say 'cvs checkout module/sub–directory' and then 'cvs release module/sub–directory' it says unknown module name. why?

- because module/sub–directory is not a module in the modules file.
- cvs release module *should* work with this.

Because of incorrect releasing of directories, I noticed that 'cvs history' reports that modules are still checked out. how do I correct this?

- by editing $CVSROOT/CVSROOT/history VERY carefully
- I do not know the correct way of doing this but the O lines are for checking Out modules. Removing the last O line that corresponds to the module is question may work.

If I just typoed and started to check out a tree I did not want to. Can I hit control–c? What are the ramifications?

- don't know exactly

I screw up and removed the tree that I was about to start working on. How do I tell cvs that I want to release it if I don't have it anymore?

- maybe you can't
- you need to edit $CVSROOT/CVSROOT/history VERY carefully to fix this problem (see above)
- FEATURE: an option to release to assure it that no changes were made and the status of the directory can be released

What is the proper way to configure cvs for multi-user operations? What sort of file directory modes are appropriate aside from 770 modes everywhere. Any setgid support?

- normal 770 directory modes should work.
- set your default umask to 027
- make your $CVSROOT directories set-gid all with a common group (src would make sense) and mode 770
- make sure all your developers are in the src group.

I am constantly running into different library modules to fix problems or add need features. This may not be a good practice but how does cvs fit in this scenario. Should I checkout the modules I need and once and a while commit them?

- yes.

I have 4 releases of my debug malloc subsystem. They are 1.01, 1.02, 1.03, 1.05. Should I:

```
cvs import –b 1.1.1 –m 'Malloc 1.01' MALLOC_1_01 malloc_1_01
cvs import –b 1.1.2 –m 'Malloc 1.02' MALLOC_1_02 malloc_1_02
cvs import –b 1.1.3 –m 'Malloc 1.03' MALLOC_1_03 malloc_1_03
cvs import –b 1.1.5 –m 'Malloc 1.05' MALLOC_1_05 malloc_1_05
```

for each set of files? Are these sane incantations?

- not really. cvs import may not be the right may to do this.
- you should cvs import the *first* set of files then for each release you should:
  - copy in the new version
  - cvs add/remove the files that have been added/deleted
  - cvs commit the new version
  - cvs tag the files with the appropriate release-tag
  - repeat

When I say 'cvs checkout malloc' I do not get 1.05 files. I get them all. I have to say 'cvs checkout –r malloc_1_05 malloc' to do this. Is this correct? Do I need to 'cvs remove' the files that have been removed between the different versions?

- yes, you need to cvs add/remove files to have the repository know about them. the files that are removed are placed in the Attic so that old-revisions can find them.

Is there a cvs feature to tell me what files have changed, what are new what have been removed from the current directory?

- when you do a cvs release, cvs will inform you of the missing files and the new files.

- FEATURE: cvs commit with an option should tell you the files that are missing or added.

If I had fileX at one point, then I 'cvs remove' it, then I recreate it. How can I re-add it. It complains that it is in the attic. Can it live there for old versions but have a new copy also?

- I believe you can move the file out of the Attic directory by hand. this seems to work fine.

7 Introduction to PROLOG

Prolog is a programming language derived from first-order logic. To understand Prolog, it is helpful to put aside any knowledge of conventional programming, as Prolog's behaviour is entirely unlike that of languages such as Pascal or C. Although much of the same terminology is used, it often refers to very different concepts.

A Prolog program consists of **facts** and **rules**. When a Prolog program is executed, these facts and rules are used to determine whether or not a new fact, or **query**, is **true** or **false**. The following English-language example illustrates this process. Suppose that there are the facts:

it is wet;
it is night;

and the rules:

roads are dangerous if slippery and if it is dark;
roads are slippery if wet;
roads are slippery if icy;
it is dark at night.

To answer the query 'are the roads dangerous', it is necessary to see if they are slippery, and whether or not it is dark. They are slippery because it is wet, and it is dark because it is night – therefore, the query is true.

7.1 Clauses and predicates

There are many variants of Prolog, although they are all essentially similar. **Nu-Prolog** is the local dialect, described fully in the *NU-Prolog Reference Manual*. In NU-Prolog, facts look like this:

night.
wet.

and rules like this:

dangerous :– slippery, dark.
slippery :– wet.
slippery :– icy.
dark :– night.

Each rule or fact is a **clause**. The **head** of a clause is the part to the left of the implied-by symbol ('`:-`'), and the **body** of a clause is the part to the right. ',' in the body of a clause means **and**. If the heads of two clauses have the same name (functor) and number of arguments (arity), they are said to belong to the same **predicate**.

Suppose that Prolog is given the query **dangerous**. The first rule above says that to prove **dangerous**, it is necessary to prove **slippery** and **dark**. It will be **slippery** if **wet**; **wet** is a fact and therefore true; similarly, it is **dark** because **night**. Therefore, **dangerous** is true.

7.2 Constants and variables

A **constant** is represented in Prolog as an alphanumeric string beginning with a lower-case letter; a number; or any string enclosed in single quotes.

The concept of **variable** is entirely different from that of a conventional language – in particular, there is no assignment. This is discussed in **Variables, unification and binding**. Variables are represented either by '_' or by alphanumeric strings beginning with an upper-case letter. The 'un-named' variable, represented by '_', is used as a place holder for arguments which are not of interest. There are no types in Prolog; any variable may be associated with constants of any type; and there may be several different types within a structure.

Consider the following Prolog program:

```
nationality(sam, australian).
nationality(mikhail, russian).

of_voting_age(sam).

does_vote(Person) :-
    of_voting_age(Person),
    nationality(Person, australian).
```

In this example, **sam**, **australian**, **mikhail** and **russian** are constants and **Person** is a variable. The single clause of **does_vote/1** (1 refers to the arity) has two subgoals, **of_voting_age(Person)** and **nationality(Person, australian)**. The predicate **nationality/2** has two clauses; note that if the arity of two heads is different, they are not part of the same predicate. This is a very common bug in Prolog.

A **goal** or **query** is a statement of a fact to be proved. Prolog executes goals by attempting to match subgoals of the present goal to heads in the program; the body corresponding to a matching head replaces the matched subgoal in the original goal. For example, given the goal:

```
?- does_vote(Voter).
```

does_vote(Voter) will match with **does_vote(Person)**, so that the goal becomes:

```
?- of_voting_age(Voter), nationality(Voter, australian).
```

of_voting_age(Voter) will match with **of_voting_age(sam)**, associating **sam** with **Voter**, and **nationality(sam, australian)** is true. The goal will return:

```
Voter = sam ?
```

indicating that for the query to be true **Voter** must have the value **sam**.

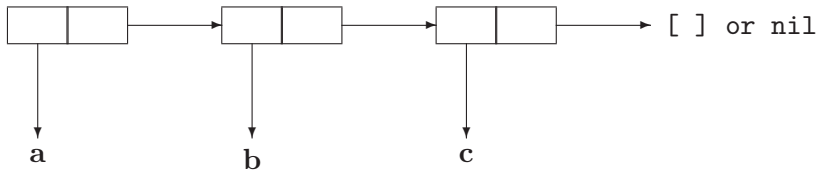
7.3 Lists

An argument to a subgoal or head does not have to be single variable or constant; it may also be a data structure. The most common data structure is probably the **list**. A list consisting of the constants **a**, **b**, and **c** would be:

```
a.b.c.[ ]
```

where [ ] is used to represent end-of-list. This notation corresponds to the conventional idea

of a linked list:



Similarly, in the notation:

Head.Tail

Head refers to the first element in the list, and **Tail** the remainder of the list. An alternative notation for lists is:

[a, b, c] and **[Head | Tail]**

with the same semantics as above. For example, either:

students(303, larry.curly.mo.[ ]), or
students(303, [larry, curly, mo])

might be used to indicate that **larry**, **curly**, and **mo** are the **303** students. Lists may have sublists, or substructures, to any degree. For example, **[[smith, [john, mark]], 24]** is quite legitimate, where **[john, mark]** is a sublist of **[smith, [john, mark]]**, and so on.

Users beware: a list such as **1.2.3.**[] is ambiguous, because periods are used to indicate floating point as well as list concatenation. This list would have to be written as **[1, 2, 3]** or **(1).(2).(3).**[].

7.4 Variables, unification and binding

A **variable** in Prolog is much the same as a variable in mathematics, and not at all like a variable in standard programming languages. In mathematics, it is meaningless to say that $X = 1$ and $X = 2$, because X can only have one value; it represents a single (perhaps unknown) entity. This is in contrast to Pascal, where a variable is a label for a memory location which can hold different values at different times. Furthermore, in Prolog variables do not assume values through assignment; rather, when a subgoal to be proved is matched to the head of a clause, the variables and constants that are the arguments become associated – formally, **unified** – with one another. For example, consider:

?– ..., p(A, b.L, yes, X), . . .

p(5, T.n.L, B, B) :– . . .

Here, **A** will unify with **5**, **T** with **b**, **L** with **n.L** (the two usages of **L** are distinct; all variable names are local), **B** with **yes**, **B** with **X**, and hence, **X** with **yes**. Consider:

?– . . . , equals(A, B), . . .

equals(A, A).

equals is an identity predicate. If **A** and **B** are identical, or if either is a variable, **equals** will

unify them; otherwise, **equals**, and the query, will fail. Some examples of goals and heads which will not unify are:

```
?– p(a).
p(b).

?– q(X.L).
q(y).
```

When two terms unify, they become **bound**, and when a variable is bound to a constant, or a term composed entirely of constants, then that variable is **ground**. If **foo(A, n.X)** and **foo(5, Y)** were unified, **A** would become bound to **5**, so that **A** is ground, and there would be a **binding** between **n.X** and **Y**.

A more complex example is given by the following program, which concatenates the first two arguments (both lists) to form the third argument.

```
append([ ], List, List). (1)
append(El.List1, List2, El.NewList) :– (2)
        append(List1, List2, NewList).
```

(1) represents the fact that the result of appending any list to an empty list is that list. **(2)** is a rule stating that, in appending any list to a non-empty list, the first element of the new list will be the same as the first element of the non-empty list. The remainder of the new list will be the second list appended to the remainder of the first list. For example, to see if **[a, b, c]** is comprised of the lists **[a]** and **[b, c]**, it is necessary to see if the query:

```
?– append([a], [b, c], [a, b, c]).
```

is true. The given query will not unify with **(1)**, as **[a]** won't unify with [ ], so Prolog attempts to unify it with **(2)**. **El** will bind to **a**, as **[a]** is equivalent to **a.[ ]**, **List1** to [ ], **List2** to **[b, c]**, and **NewList** to **[b, c]**, and the query is replaced by:

```
?– append([ ], [b, c], [b, c]).
```

which unifies with **(1)**. There are no more subgoals to be proved; therefore, the query is shown to be true.

In addition to the above, all of the following queries to append will succeed:

```
?– append([a, b, c], [d, e], New).
?– append([a, b, c], List2, [a, b, c, d, e]).
?– append(List1, [d, e], [a, b, c, d, e]).
```

The first would bind **New** to **[a, b, c, d, e]**, the second would bind **List2** to **[d, e]**, and the third would bind **List1** to **[a, b, c]**.

7.5 Negation and delay

It is sometimes useful to have negation in a Prolog program, that is, to deduce that something is true when something else is false. Consider the following program:

```
single_parent(Person) :-
        not married(Person),
        parent_child(Person, _).

married(thomas).
married(huckleberry).

parent_child(alice, thomas).

?- single_parent(alice).
```

where **not** means negation and the underscore implies that the name of the child is not interesting; what is important is that **Person** takes part in a **parent_child** predicate.

To execute the given goal, Prolog would first attempt to prove that **married(alice)** is true. If this succeeds, **not married(alice)**, and the program, will fail; if it fails, then **not married(alice)** will succeed, and Prolog will attempt to prove **parent_child(alice, _)**. However, consider the query:

```
?- single_parent(X).
```

which is intended to find a single parent. If Prolog were to immediately attempt to prove **married(X)**, it would unify this subgoal with **married(thomas)**, so that **not married(X)** would fail. However, there is a solution (**alice**) so that clearly this behaviour (a property of most versions of Prolog) is unsound.

NU-Prolog, however, incorporates the notion of **delay**. A negated subgoal is delayed – not executed – until all of its arguments are ground. Given the query above, the call to **not married(X)** will be delayed and Prolog will attempt to prove **parent_child(X, _)**. This will bind **X** to **alice**, so that **X** is ground, and the call to **not married(X)** can proceed.

Other predicates which will delay if their arguments are not ground (and are therefore soundly implemented) are **~=** (not equals) and **if-then-else**. Both of these, and **not**, have unsound equivalents, which may be needed in some circumstances. However, they should be avoided if possible.

It is an error for a program to complete execution with some calls delayed. Any predicate may be caused to delay until some arguments are bound with **when** declarations. These are discussed in the NU-Prolog Reference Manual.

7.6 Backtracking

When a subgoal fails – that is, won't unify with any head – Prolog will **backtrack** to the previous subgoal, and retry it; this may change the bindings which caused the subgoal to fail. Consider a trace of the execution of the following program:

```
enrolled(tom, 303).
enrolled(dick, 380).
enrolled(harry, 303).

attendance(tom, part_time).
attendance(dick, full_time).
attendance(harry, full_time).

?– enrolled(Student, 303), attendance(Student, full_time).
```

This program is used to determine which students are enrolled full-time in 303. With the given query, **Student** will bind to **tom**, but **attendance(tom, full_time)** will fail. Prolog will backtrack to **enrolled(Student, 303)**, this time binding **Student** to **harry**, and **attendance(harry, full_time)** succeeds. Consider a more involved example:

```
member(El, El.List).
member(El, Head.List) :–
        El ~= Head,
        member(El, List).

common(El, List1, List2) :–
        member(El, List1),
        member(El, List2).

?– common(A, [a, b, c, d], [e, b, g]).
```

If **member** is called with the first argument unbound, it will return a member of the list in the second argument. So, **common** will get any element from **List1** and test it against **List2**. If it is not in **List2**, **member(El, List2)** will fail and **member(El, List1)** will be retried, getting another element from **List1**, and so on, until a common element is found or **List1** is exhausted, in which case **common** will fail.

Backtracking is not a random process; it is an orderly search through the possible solutions, based upon the following ordering. Prolog uses left-to-right evaluation; that is, the leftmost subgoal in a rule or query will be proved first, the rightmost last. If a predicate has more than one clause, the topmost is considered first, and the bottom last. If the given subgoal cannot be proved with the first clause of a predicate, then Prolog will attempt to prove it using the second clause, and so on.

7.7 Functors

Arguments to subgoals are not restricted to being variables, constants, or lists: they may be a function, or any expression in parentheses. For example, both **person(frank, 26, rm7)** and $(12 + 13 * 4)$ are legitimate arguments. Consider:

```
odd(s(0)).
odd(s(X)) :- even(X).

even(0).
even(s(X)) :- odd(X).

?- even(s(s(s(s(s(0)))))).
```

Unless explicitly tested, arguments are not evaluated for truth or falsity. They are simply a structure for holding information, of which lists are a particular instance.

For clarity, some unary or binary terms maybe written as infix, prefix or postfix operators. For example, $X + Y$ is a convenient notation for the expression $+(X, Y)$ and **all X p(X)** (**all** might be universal quantification) is an alternative for **all(X, p(X))**. Operators – unless built into Prolog – must have operator declarations.

7.8 Numbers

As each variable may take only one value, tools such as counters are not straightforward. The simplest solutions include using aggregate functions or property lists (neither of which are described in this manual).

Users should always be careful to use the appropriate form of comparison, of which there are several available. First of all, there is =, which is used to unify its arguments, or – if they are ground – to test for identity. Naturally, this is not only used for integers. There are many cases where, although it seems natural to use =, unification is in fact more appropriate. Consider the following example:

```
test(A, B) :-
        A = 1,
        B = 1.
test(A, B) :-
        A ~= 1,
        B = 0.
```

where **B** will be set to **1** if **A** is **1**, and set to **0** otherwise. This may be rewritten as:

```
test(1, 1).
test(A, 0) :-
        A ~= 1.
```

Some arithmetic predicates which are built into Prolog evaluate their arguments, which may be expressions as well as numbers. For example, the predicate **is/2** evaluates its second argument and unifies the result with its first, so that **X is 7 +2** would bind **X** to **9**. The predicate =:= evaluates both of its arguments, and succeeds if the results are equal, so $3 * 4 =:= 6 * 2$ would succeed. These predicates delay until their arguments are ground.

7.9 System predicates

There is an extensive set of system predicates, which are described in the NU-Prolog Reference Manual. These include predicates for input and output, database manipulation, debugging, and arithmetic.

A number of the system predicates are non-logical; that is, they may have some side effect as well as evaluating to true or false, and may not behave well on backtracking. The most important of these is **!**, known as **cut**. This is a highly dangerous predicate, and should be used only when absolutely required. Essentially, **cut** prevents backtracking; if the subgoals following **cut** in a clause fail, then the subgoal which was matched with the head of that clause will fail also, without retrying earlier subgoals of that clause of other clauses with the same head. Consider the following program:

```
p :- q, r.

q :- a, !, b.
q :- c.
```

If **a** succeeds and **b** fails, the subgoal **q** in predicate **p** will fail also, leading to the failure of predicate **p**.

Also of interest are **assert** and **retract**, which may be used to dynamically change the program by adding or deleting facts or rules, and property lists, which are used for global data storage.

7.10 Connectives

The connective '**,**' (and) is not the only connective available in NU-Prolog. Other commonly used connectives are '**;**' (**or**) and **if-then-else**. For example, the following three predicates are equivalent.

```
sign1(Q, +) :–
        N is Q,         % evaluate Q (it might be an expression)
        N >= 0.
sign1(Q, –) :–
        N is Q,
        N < 0.


sign2(Q, Sign) :–
        N is Q,         % evaluate Q (it might be an expression)
        ( N >= 0,       % parentheses to override precedence of ‘,’
                Sign = +
        ; N < 0,
                Sign = –
        ).


sign3(Q, Sign) :–
        N is Q,         % evaluate Q (it might be an expression)
        (if N >= 0 then         % parentheses to override precedence
of ‘,’
                Sign = +
        else
                Sign = –
        ).
```

The character ‘%’ starts a comment; any text between the strings `/*` and `*/` is also treated as a comment.

Although each of the above predicates has the same semantics, their behaviour is different. In **sign1/2**, the expression **Q** may be evaluated twice. In **sign2/2**, although the cases are exclusive, both branches of the **or** will be tried if **Q** is positive and a call following the call to **sign2/2** fails. In contrast, the body of **sign3/2** would only be tried once.

Other connectives include **all/2** and **some/2** (logical quantification), **not**, and ⇒ and ⇐ (implication).

7.11 Aggregate Functions

The following aggregate functions are available; **solutions/3**, **count/3**, **max/3**.

solutions(Term, Formula, Set)

In a declarative sense, *Set* is the list of instances of *Term* for which *Formula* is true.

count(Term, Formula, Result)

Result is the number of distinct instantiations of *Term* by *Formula*

max(Term, Formula, Result)

Result is the maximum instantiation of *Term* by *Formula*.

The scope of variables contained in *Term* is local to the call to each respective aggregate function. Additional aggregate functions are described in detail in the NU–Prolog Reference Manual.

8 Lex – A Lexical Analyzer Generator

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

8.1 Introduction

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries

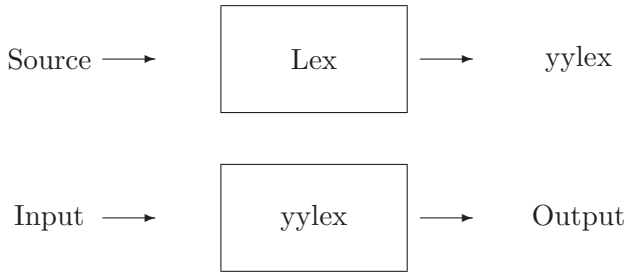


Figure 1: An overview of Lex

for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex.* The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$   ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$   ;
[ \t]+    printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars,

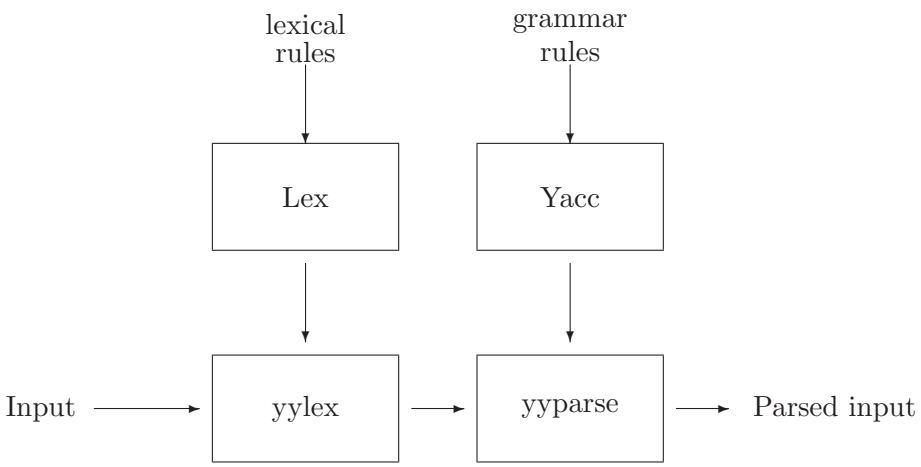


Figure 2: Lex with Yacc

but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.

Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of re˜scanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character look˜ahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg* , and the input stream is *abcdefh* , Lex will recognize *ab* and leave the input pointer just before *cd. . .* Such backup is more costly than the processing of simpler languages.

8.2 Lex Source

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second *%%* is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions,* program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum* ; a way of dealing with this will be described later.

8.3 Lex Regular Expressions

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

a57D

looks for the string *a57D.*

Operators. The operator characters are

" \ [ ] ^ - ? . + | ( ) $ / { } % < >

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

xyz"++"

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

"xyz++"

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

xyz\+\+

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes Classes of characters can be specified using the operator pair []. The construction *[abc]* matches a single character, which may be *a* , *b* , or *c* . Within square brackets, most operator meanings are ignored. Only three characters are special: these are `\ - and ^`. The - character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0–z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the
character in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters; or

```
[^a-zA-Z]
```

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

```
.
```

is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator *?* indicates an optional element of an expression. Thus

ab?c

matches either *ac* or *abc* .

Repeated expressions. Repetitions of classes are indicated by the operators * and + .

```
a*
```

is any number of consecutive *a* characters, including zero; while

```
a+
```

is one or more instances of *a.* For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

```
(ab|cd)
```

matches either *ab* or *cd.* Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)
```

matches such strings as *abefef* , *efefef* , *cdef* , or *cddd* ; but not *abc* , *abcd* , or *abcdef* .

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and **$** . If the first character of an expression is ^ , the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^ , complementation of character classes, since that only applies within the [] operators. If the very last character is **$** , the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string
ab , but only if followed by *cd.* Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition
x, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE* , then the ^ operator would be equivalent to <ONE>

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of *a* .

Finally, initial % is special, being the separator for Lex source segments.

8.4 Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, *;* as an action causes this result. A frequent rule is

```
[ \t\n]    ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "     |
"\t"    |
"\n"    ;
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[az]+* . Lex leaves this text in an external character array named *yytext.* Thus, to print the name found, a rule like

```
[a–z]+    printf("%s", yytext);
```

will print the string in *yytext.* The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext.* So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a–z]+    ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust* ; to avoid this, a rule of the form *[az]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a–zA–Z]+   {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng–1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext.* Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of look˜ahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*   {
          if (yytext[yyleng–1] == \\)
          yymore();
          else
          . . . normal user processing
          }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc* ; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labelled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "`=-a`". Suppose it is desired to treat this as "`=- a`" but print a message. A rule might be

```
=-[a-zA-Z]   {
             printf("Op (=-) ambiguous\n");
             yyless(yyleng–1);
             . . . action for =- . . .
             }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "`=-`". Alternatively it might be desired to treat this as "`= -a`". To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]   {
             printf("Op (=-) ambiguous\n");
             yyless(yyleng–2);
             ... action for = ...
             }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3",however, makes

```
=-/[^\t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;

2) *output(c)* which writes the character *c* on the output; and

3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex look˜ahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + ? or $ or containing / implies look˜ahead. Look˜ahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access

to this condition is through *yywrap.* In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

8.5 Ambiguous Source Rules.

Lex can handle ambiguous specifications.
When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.

2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer     keyword action ...;
[a--z]+     identifier action ...;
```

to be given in that order. If the input is *integers* , it is taken as an identifier, because *[a–z]+* matches 8 characters while *integer* matches only 7. If the input is *integer* , both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like *.** dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

'first' quoted string here, 'second' here the above expression will match

'first quoted string here, second'

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after '*first*' . The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like . stop on the current line. Don't try to defeat this with expressions like *(.|\n)+* or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some Lex rules to do this might be

```
she   s++;
he    h++;
\n    |
.     ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that . does not include newline. Since *she* includes *he*, Lex will normally *not* recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*:

```
she   {s++; REJECT;}
he    {h++; REJECT;}
\n    |
.     ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he* but not vice versa, and omit the REJECT action on *he*; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+   { ... ; REJECT;}
a[cd]+   { ... ; REJECT;}
```

If the input is *ab* , only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he* . Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%% [a–z][a–z]   {
                 digram[yytext[0]][yytext[1]]++;
                 REJECT;
                 }
                 ;
\n               ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

8.6 Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                   [0--9]
E                   [DEde][-+]?{D}+
%%
{D}+                printf("integer");
{D}+"."{D}({E})?    |
{D}"."{D}+({E})?    |
{D}+{E}             printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second

requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I* , which does not contain a real number, a context-sensitive rule such as

[0–9]+/"."EQ printf("integer");

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

8.7 Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c* . The I/O library is defined in terms of the C standard library [6].

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same.

UNIX. The library is accessed by the loader flag *–ll* . So an appropriate set of commands is

lex source cc lex.yy.c –ll

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided.

8.8 Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex(),* the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex().* In this case each Lex rule should end with

return(token);

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

include "lex.yy.c"

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c –ly –ll
```

The Yacc library (–ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

8.9 Examples

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
            int k;
[0--9]+     {
            k = atoi(yytext);
            if (k%7 == 0)
            printf("%d", k+3);
            else
            printf("%d",k);
            }
```

to do just that. The rule [0–9]+ recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k.* The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7* . Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                            int k;
–?[0–9]+                    {
                            k = atoi(yytext);
                            printf("%d",
                            k%7 == 0 ? k+3 : k);
                            }
–?[0–9.]+                   ECHO;
[A-Za-z][A-Za-z0-9]+        ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if–else* has been replaced by a C conditional expression to save space; the form *a?b:c* means "if *a* then *b* else *c* ".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
                                int lengs[100];
%%
[a--z]+                         lengs[yyleng]++;
.                               |
\n                              ;
%%
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
if (lengs[i] > 0)
printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table.
The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including
both cases of each letter:

```
a     [aA]
b     [bB]
c     [cC]
...
z     [zZ]
```

An additional class recognizes white space:

```
W     [\t]*
```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
printf(yytext[0]==d? "real" : "REAL");
}
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^" "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of ^ . There follow some rules to change double precision constants to ordinary floating constants.

```
[0--9]+{W}{d}{W}[+-]?{W}[0--9]+      |
[0--9]+{W}"."{W}{d}{W}[+-]?{W}[0--9]+      |
"."{W}[0--9]+{W}{d}{W}[+--]?{W}[0--9]+      {
/* convert constants */
for(p=yytext; p != 0; p++)
{
if (p == d || *p == D)
p=+ e– d;
ECHO;
}
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D* . The program than adds *e–d* , which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}              |
{d}{c}{o}{s}              |
{d}{s}{q}{r}{t}           |
{d}{a}{t}{a}{n}           |
... {d}{f}{l}{o}{a}{t}    printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}      |
{d}{l}{o}{g}10    |
{d}{m}{i}{n}1     |
{d}{m}{a}{x}1     {
                  yytext[0] =+ a – d;
                  ECHO;
                  }
```

And one routine must have initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h}    {yytext[0] =+ r – d;
                    ECHO;
                    }
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A–Za–z][A–Za–z0–9]    |
[0--9]+                |
\n                     |
.                      ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

8.10 Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as *$* recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
            int flag;
%% ^a       {flag = a; ECHO;}
^b          {flag = b; ECHO;}
^c          {flag = c; ECHO;}
\n          {flag = 0 ; ECHO;}
magic       {
            switch (flag)
            {
            case a: printf("first"); break;
            case b: printf("second"); break;
            case c: printf("third"); break;
            default: ECHO; break;
            }
            }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

<name1>expression

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

BEGIN name1;

which changes the start condition to *name1*. To resume the normal state,

BEGIN 0;

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

<name1,name2,name3>

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC %% ^a    {ECHO; BEGIN AA;}
^b                       {ECHO; BEGIN BB;}
^c                       {ECHO; BEGIN CC;}
\n                       {ECHO; BEGIN 0;}
<AA>magic                printf("first");
<BB>magic                printf("second");
<CC>magic                printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

8.11 Character Set

The programs generated by Lex handle character I/O only through the routines *input, output,* and *unput.* Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext.* For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant *a* . If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus the next example

```
1    Aa
2    Bb
...
26   Zz
27   \n
28   +
29   –
30   0
31   1
...
39   9
```

Sample character table.

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and – into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either
in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

8.12 Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form "name space translation".

2) Included code, in the form "space code".

3) Included code, in the form

```
%{
code
%}
```

4) Start conditions, given in the form

```
%S name1 name2 ...
```

5) Character set tables, in the form

%T
number space character-string
...
%T

6) Changes to internal array sizes, in the form

%x nnn

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

6) Changes to internal array sizes, in the form %x nnn
where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follow:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
`[xy]`	the character x or y.
`[x--z]`	the characters x, y or z.
`[^x]`	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

8.13 Acknowledgments

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

8.14 References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, N. J. (1978).

2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran,* Software – Practice and Experience, **5**, pp. 395-496 (1975).

3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler,* Computing Science Technical Report No. 32, 1975,

4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search,* Comm. ACM **18,** 333-340 (1975).

5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor,* Computing Science Technical Report No. 5, 1972,

6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library,* Computing Science Technical Report No. 31,

9 Yacc: Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc[5] provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

9.1 Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser* , calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules* ; when one of these rules has been recognized, then user code supplied for this rule, an *action* , is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C [2] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : monthname day ',' year ;
```

Here, *date* , *monthname* , *day* , and *year* represent structures of interest in the input process; presumably, *monthname* , *day* , and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the

[5]This document is edited version of a portion of the UNIX Programmer's Manual see [1]

input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol* , while the structure recognized by the parser is called a *nonterminal symbol* . To avoid confusion, terminal symbols will usually be referred to as *tokens* .

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
monthname : 'J' 'a' 'n' ;
monthname : 'F' 'e' 'b' ;
.
.
.
monthname : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *monthname* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *monthname* was seen; in this case, *monthname* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power

compares favourably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere. [3, 4, 5] Yacc has been extensively used in numerous practical applications, including *lint*, [6] the Portable C Compiler, [7] and a system for typesetting mathematics. [8]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

9.2 Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A       :       '(' B ')'
                        {       hello( 1, "abc" ); }
```

and

```
XXX     :       YYY ZZZ
                        {       printf("a message\n");
                                flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :       '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr    :       '(' expr ')'        { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A       :       B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A       :       B
                        { $$ = 1; }
                C
                        { x = $2; y = $3; }
        ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT    :       /* empty */
                        { $$ = 1; }
        ;
A       :       B $ACT C
                        { x = $2; y = $3; }
        ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node* , written so that the call

```
node( L, n1, n2 )
```

j creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr    :       expr '+' expr
                        { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

9.3 Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex* . The function returns an integer, the *token number* , representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval* .

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to

return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
                ...
        case '0':
        case '1':
        ...
        case '9':
                yylval = c-'0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk[9]. These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules.

Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

9.4 How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift* , *reduce* , *accept* , and *error* . A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.       reduce 18
```

refers to *grammar rule* 18, while the action

```
IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A       :       x y z ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x , y , and z , and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme    :        sound place
;
sound    :        DING DONG
;
place    :        DELL
         ;
```

When Yacc is invoked with the **−v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept  :        _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept  :        rhyme_$end

        $end accept
        . error

state 2
        rhyme    :        sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound    :        DING_DONG

        DONG shift 6
        . error

state 4
        rhyme    :        sound place (1)

        . reduce 1
```

```
state 5
        place    :      DELL (3)

        . reduce 3

state 6
        sound    :      DING DONG (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING* , is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG* , is read, becoming the lookahead token. The action in state 3 on the token *DONG* is "shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
        sound    :      DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound* ,

```
        sound goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL* , must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place* , the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG* , *DING DONG* , *DING DONG DELL DELL* , etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

9.5 Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

expr : expr '–' expr

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

expr – expr – expr

the rule allows this input to be structured as either

(expr – expr) – expr

or as

expr – (expr – expr)

(The first is called *left association* , the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr – expr – expr

When the parser has read the second expr, the input that it has seen:

expr – expr

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

– expr

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

expr – expr

it could defer the immediate application of the rule, and continue reading the input until it had seen

expr – expr – expr

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

expr – expr

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

expr – expr

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict* . It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict* . Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule* .

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat    :       IF '(' cond ')' stat
        |       IF '(' cond ')' stat ELSE stat
        ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
        IF ( C2 ) S1
        }
        ELSE S2
```

or

```
IF ( C1 ) {
        IF ( C2 ) S1
        ELSE S2
        }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*d" *IF* . In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

IF (C1) IF (C2) S1

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser. The conflict messages of Yacc are best understood by examining the verbose (**−v**) option output file. For example, the output corresponding to the above conflict state might be:

23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

stat : IF (cond) stat (18)
stat : IF (cond) stat_ELSE stat

ELSE shift 45
. reduce 18

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

IF (cond) stat

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE* , it is possible to shift into state 45. State 45 will have, as part of its description, the line

stat : IF (cond) stat ELSE_stat

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by "**.**", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE* , the parser reduces by grammar rule 18:

stat : IF '(' cond ')' stat

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references [3, 4, 5] might be consulted; the services of a local guru might also be appropriate.

9.6 Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
        expr : expr OP expr
and
        expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '=' expr
        |       expr '+' expr
        |       expr '-' expr
```

```
        |       expr '*' expr
        |       expr '/' expr
        |       NAME
        ;
```

might be used to structure the input

$a = b = c * d - e - f * g$

as follows:

$a = (b = (((c * d) - e) - (f * g)))$

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '–'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '–'
%left '*' '/'

%%

expr    :       expr '+' expr
        |       expr '–' expr
        |       expr '*' expr
        |       expr '/' expr
        |       '–' expr %prec '*'
        |       NAME
        ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two

disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

9.7 Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat : error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :       error '\n' { printf( "Reenter last line: " ); } input
                        { $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :       error '\n'
                        {       yyerrok;
                                printf( "Reenter last line: " ); }
                input
                        { $$ = $4; }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
```

```
                        {       resynch();
                                yyerrok ;
                                yyclearin ; }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

9.8 The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse* ; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex* , the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse* . In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror* . The name of this library is system dependent; on many systems the library is accessed by a **–ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

        yyerror(s) char *s; {
                fprintf( stderr, "%s\n", s );
                }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9.9 Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

9.9.1 Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

1. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

9.9.2 Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name    :       name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list ',' item
        ;
```

and

```
seq     :       item
        |       seq item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :       item
        |       item seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

9.9.3 Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
... other declarations ...

%%

prog    :       decls stats
        ;
```

```
decls   :       /* empty */
                        {       dflag = 1; }
        |       decls declaration
        ;

stats   :       /* empty */
                        { dflag = 0; }
        |       stats statement
        ;

... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

9.9.4 Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved* ; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

9.10 Advanced Topics

This section discusses a number of advanced features of Yacc.

9.10.1 Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

9.10.2 Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :       adj noun verb adj noun
                        {look at the sentence ... }
        ;

adj     :       THE             {       $$ = THE; }
        |       YOUNG           {       $$ = YOUNG; }
        ...
        ;

noun    :       DOG
                        {       $$ = DOG; }
        |       CRONE
                        {       if( $0 == YOUNG ){
                                        printf( "what?\n" );
                                        }
                                $$ = CRONE;
                                }
        ;
        ...
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

9.10.3 Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[6] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval* , to have type equal to this union. If Yacc was invoked with the **–d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ... . .
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '–'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype* . Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa { $<intval>$ = 3; } bbb
                        { fun( $<intval>2, $<other>0 ); }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int'* s, as was true historically.

9.11 Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

[1] S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

[3] A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys* **6**(2) pp. 99-124(June 1974).

[4] A. V. Aho, S. C.Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammers," *Comm. Assoc. Comp. Mach.* **18**(8) pp. 441-452 (August 1975)

[5] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

[6] S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (December 1977).

[7] S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978)

[8] B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18** pp 151-157 (March 1975).

[9] M. E. Lesk, "Lex – A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

A A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, –, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The

major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left   '|'
%left   '&'
%left   '+' '-'
%left   '*' '/' '%'
%left   UMINUS/* supplies precedence for unary minus */

%%      /* beginning of rules section */

list    :       /* empty */
        |       list stat '\n'
        |       list error '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER '=' expr
                        {       regs[$1] = $3; }
        ;

expr    :       '(' expr ')'
                        {       $$ = $2; }
        |       expr '+' expr
                        {       $$ = $1 + $3; }
        |       expr '-' expr
                        {       $$ = $1 - $3; }
        |       expr '*' expr
                        {       $$ = $1 * $3; }
        |       expr '/' expr
                        {       $$ = $1 / $3; }
```

```
        |       expr '%' expr
                        {       $$ = $1 % $3; }
        |       expr '&' expr
                        {       $$ = $1 & $3; }
        |       expr '|' expr
                        {       $$ = $1 | $3; }
        |       '-' expr  %prec UMINUS
                        { $$ = - $2; }
        |       LETTER
                        {       $$ = regs[$1]; }
        |       number
        ;

number:         DIGIT
                        {       $$ = $1; base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                        {       $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {                       /* lexical analysis routine */
        /* returns LETTER for a lower case letter, yylval = 0 through 25 */
        /* return DIGIT for a digit, yylval = 0 through 9 */
        /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) { /* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

B Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token IDENTIFIER               /* includes identifiers and literals */
%token C_IDENTIFIER             /* identifier (but not literal) followed by colon */
%token NUMBER                   /* [0-9]+ */

        /* reserved words: %type => TYPE, %left => LEFT, etc. */

%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK   /* the %% mark */
%token LCURL  /* the %{ mark */
%token RCURL  /* the %} mark */

        /* ascii character literals stand for themselves */
%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK   {In this action, eat up the rest of the file}
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        | defs def
        ;

def     :       START IDENTIFIER
        |       UNION {Copy union definition to output}
        |       LCURL {Copy C code to output file} RCURL
        |       ndefs rword tag nlist
```

```
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' {Copy action, translate $$, etc.} '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

C An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, –, *, /, unary –, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals* , written

(x , y)

where x is less than or equal to y . There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double* 's. This structure is given a type name, INTERVAL, by using *typedef* . The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 – 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is

instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
        double lo, hi;
        } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start  lines

%union {
        int ival;
        double dval;
        INTERVAL vval;
        }

%token  <ival> DREG VREG        /* indices into dreg, vreg arrays */

%token  <dval> CONST            /* floating point constant */

%type   <dval> dexp             /* expression */

%type   <vval> vexp             /* interval expression */

        /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS/* precedence for unary minus */
```

```
%%

lines   :       /* empty */
        |       lines line
        ;
line    :       dexp '\n'
                        {       printf( "%15.8f\n", $1 ); }
        |       vexp '\n'
                        {       printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
        |       DREG '=' dexp '\n'
                        {       dreg[$1] = $3; }
        |       VREG '=' vexp '\n'
                        {       vreg[$1] = $3; }
        |       error '\n'
                        {       yyerrok; }
        ;

dexp    :       CONST
        |       DREG
                        {       $$ = dreg[$1]; }
        |       dexp '+' dexp
                        {       $$ = $1 + $3; }
        |       dexp '-' dexp
                        {       $$ = $1 - $3; }
        |       dexp '*' dexp
                        {       $$ = $1 * $3; }
        |       dexp '/' dexp
                        {       $$ = $1 / $3; }
        |       '-' dexp  %prec UMINUS
                        {       $$ = - $2; }
        |       '(' dexp ')'
                        {       $$ = $2; }
        ;

vexp    :       dexp
                        {       $$.hi = $$.lo = $1; }
        |       '(' dexp ',' dexp ')'
                        {
                        $$.lo = $2;
                        $$.hi = $4;
                        if( $$.lo > $$.hi ){
                                printf( "interval out of order\n" );
                                YYERROR;
                                }
                        }
        |       VREG
                        { $$ = vreg[$1]; }
```

```
        |       vexp '+' vexp
                        {       $$.hi = $1.hi + $3.hi;
                                $$.lo = $1.lo + $3.lo; }
        |       dexp '+' vexp
                        {       $$.hi = $1 + $3.hi;
                                $$.lo = $1 + $3.lo; }
        |       vexp '-' vexp
                        {       $$.hi = $1.hi - $3.lo;
                                $$.lo = $1.lo - $3.hi; }
        |       dexp '-' vexp
                        {       $$.hi = $1 - $3.lo;
                                $$.lo = $1 - $3.hi; }
        |       vexp '*' vexp
                        {       $$ = vmul( $1.lo, $1.hi, $3 ); }
        |       dexp '*' vexp
                        {       $$ = vmul( $1, $1, $3 ); }
        |       vexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1.lo, $1.hi, $3 ); }
        |       dexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1, $1, $3 ); }
        |       '-' vexp  %prec UMINUS
                        {       $$.hi = -$2.lo; $$.lo = -$2.hi; }
        |       '(' vexp ')'
                        {       $$ = $2; }
        ;

%%

# define BSZ 50                 /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;
        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if( islower( c ) ){
                yylval.ival = c - 'a';
                return( DREG );
                }

        if( isdigit( c ) || c=='.' ){
```

```
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp–buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == '.' ){
                        if( dot++ || exp ) return( '.' ); /* will cause syntax error */
                                continue;
                                }

                        if( c == 'e' ){
                                if( exp++ ) return( 'e' ); /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if( (cp–buf) >= BSZ ) printf( "constant too long: truncated\n" );
                else ungetc( c, stdin ); /* push back last char read */
                yylval.dval = atof( buf );
                return( CONST );
                }
        return( c );
        }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a; v.lo = b; }
        else { v.hi = b; v.lo = a; }

        if( c>d ) {
                if( c>v.hi ) v.hi = c;
                if( d<v.lo ) v.lo = d;
                }
        else {
                if( d>v.hi ) v.hi = d;
                if( c<v.lo ) v.lo = c;
                }
        return( v );
```

```
        }

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return( 1 );
                }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

D Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes "'".
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or , the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.
3. The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.
4. Most places where % is legal, backslash "\" may be used. In particular, \\is the same as %%, \left the same as %left, etc.
5. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

6. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.

7. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.