TEK SYSTEM MANUAL

Part No. 070-5486-00
Product Group 65

UTek

TOOLS

VOLUME 2

First Printing NOV 1984

Tektronix®
COMMITTED TO EXCELLENCE

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development:

W. N. Joy
O. Babaouglu
R. S. Fabry
K. Sklower
Eric P. Allman
M. K. McKusick
E.Cooper
David Musher
S. J. Leffler

University of California at Berkeley
Department of Electrical Engineering and Computer Science

Revision
INFORMATION

PRODUCT: 6000 Family UTek Operating System: 64WP02, 64WP05, 64WP06
This manual supports the following versions of this product: V2.0

REV DATE	DESCRIPTION
NOV 1984	Original Issue

Contents

Section 5C Curses and Terminfo Package — Page

Section 5D Curses Examples — Page

Section 5E The Fortran 77 Compiler

Section 5F Ratfor — A Preprocessor for Rational Fortran Page

Section 5G Using Pascal on Utek

Section 5G Using Pascal on Utek (cont) Page

Section 5H Lexical Analyzer Generator (lex)

Section 5I The M4 Macro Processor — Page

Section 5J The Programming Language EFL

Section 5J The Programming Language EFL (cont) Page

Section 5K Introduction to Debugging

Section 5L Using the adb Debugger — Page

Section 5M Using sdb, a Symbolic Debugger

Section 6A RCS — A Revision Control System

Section 6B Using Make — Page

Section 6C Using Make and RCS Together

Section 6D The Awk Programming Language

Section 6D The Awk Programming Language (cont) Page

Section 7A Interactive Desk Calculator (DC)

Section 7A Interactive Desk Calculator (DC) (cont) Page

Section 7B Arbitrary Precision Desk Calculator Language (BC)

Figures

Examples

Tables

Lint — A C Program Checker

Lint examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. You can also use it to enforce portability restrictions. Another option detects a number of wasteful, error prone constructions that are still legal in C.

Lint accepts multiple input files and library specifications, and checks them for consistency. It runs more slowly than the C compiler, but it examines a program more carefully.

This document discusses the use of **lint**, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

Usage

Suppose there are two C source files, *file1. c* and *file2.c*, which are ordinarily compiled and loaded together.

lint *file1.c file2.c*

This produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers enforce them. For example:

lint **—p** *file1.c file2.c*

This produces additional messages that relate to the portability of the programs to other operating systems and machines. Replacing the **—p** by **—h** produces messages about various error-prone or wasteful constructions that are not bugs. Entering **—hp** detects both portability problems and wasteful constructions.

The next several topics describe the major messages; the last topics discuss the implementation and give suggestions for writing portable C.

Design Philosophy

Many of the facts that **lint** needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Thus, most of the **lint** algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, **lint** assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give highly relevant information. Messages of the form *xxx might be a bug* are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover.

Keeping these issues in mind, we now consider in more detail the classes of messages that **lint** produces.

Unused Variables and Functions

As sets of programs evolve and develop, previously–used variables and arguments to functions can become unused. External variables, or even entire functions, may become unnecessary and yet not be removed from the source. These errors rarely cause working programs to fail, but they are a source of inefficiency and make programs harder to understand and change. Moreover, information about such unused variables and functions can sometimes help discover bugs; if a function does a necessary job and is never called, something is wrong!

Lint complains about variables and functions that are defined but not otherwise mentioned. An exception is variables that are declared through explicit external statements but are never referenced. Thus the statement:

```
extern float sin( );
```

evokes no comment if *sin* is never used. This agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest. They can be discovered by adding the **—x** option to the **lint** invocation.

Certain styles of programming require many functions to be written with similar interfaces. Frequently, some of the arguments may be unused in many of the calls. The **—v** option is available to suppress the printing of complaints about unused arguments. When **—v** is in effect, no messages are produced about unused arguments, except for those arguments that are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

In one case, information about unused or undefined variables is more distracting than helpful. This is when **lint** is applied to some, but not all, files that are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. You can use the **—u** option to suppress the spurious messages, which might otherwise appear.

Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms use large resources and still produce messages saying a program is valid. **Lint** detects local variables (automatic and register storage classes) whose first use is earlier in the input file than the first assignment to the variable. **Lint** assumes that taking the address of a variable uses that variable, since the actual use may occur at any later time.

The algorithm to check use of variables is very simple to implement, since it is restricted to the physical orders of the variables in the file. It does mean that **lint** complains about some programs that are legal, but these programs might be considered bad on stylistic grounds . Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables and variables that are used in the expression that first sets them.

The set and/or used information also permits recognition of those local variables that are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Control Flow

Lint tries to detect unreachable portions of the programs that it processes. It complains about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It tries to detect loops that can never be left at the bottom, detecting the special cases **while**(1) and **for**(;;) as infinite loops. **Lint** also complains about loops that cannot be entered at the top.

Lint has no way of detecting functions that are called, but never return. Thus, a call to **exit** can cause unreachable code that **lint** does not detect. The most serious effects of this deficiency are in the determination of returned function values.

Usually, **lint** does not complain about one kind of unreachable statement: a **break** statement that cannot be reached. Programs generated by **yacc**, frequently generate unreachable **break** statements. The **—O** option in the C compiler often eliminates the resulting object code inefficiency. Thus, these unreached statements are of little importance; usually, you can't do anything about them. The resulting messages clutter up the **lint** output. If these messages are desired, **lint** can be invoked with the **—b** option.

Function Values

Sometimes functions return values that are never used; sometimes programs incorrectly use function values that have never been returned. **Lint** addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

return(*expr*);

and

return ;

statements is cause for alarm; **lint** gives the message:

function *name* **contains return(e) and return**

The problem is detecting when a function return is implied by control flow reaching the end of the function. For example:

```
f (aU(a) {
  if (a) return (3);
  g ();
  }
```

Notice that, if *a* tests false, *f* calls *g*, and then **return** with no defined return value; this triggers a complaint from **lint**. If *g*, like *exit*, never returns, the message is still produced when nothing is wrong. In practice, some potentially serious bugs have been discovered by this feature; but it also accounts for a substantial number of the unnecessary messages produced by **lint**.

On a global scale, **lint** detects cases where a function returns a value, but that is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (for example, not testing for error conditions).

This condition is detected using a function value when the function does not return one. This bug has been observed in working programs; the desired function value was computed in the function return register!

Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure selection operators
- between the definition and uses of functions
- in the use of enumerations

There are a number of operators that have an implied balancing between types of the operands. The assignment, conditional (? :), and relational operators have this property. The argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the —> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations, and that the only operations applied are =, initialization, = =, ! =, and function arguments and return values.

Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment:

```
p = 1 ;
```

where *p* is a character pointer. **Lint** will quite rightly complain. Now, consider the assignment:

```
p = (char *)1 ;
```

In this case a cast is used to convert the integer to a character pointer. It seems harsh for **lint** to continue to complain about this. On the other hand, if this code is moved to another machine, such code is looked at carefully. The **—c** option controls the printing of comments about casts. When **—c** is in effect, casts are treated as though they were assignments subject to complaint. Otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On the PDP–11, characters are signed quantities, with a range from —128 to 127. On most of the other C implementations, characters take on only positive values. Thus, **lint** flags certain comparisons and assignments as being illegal or nonportable. For example, the fragment:

```
char c;
        ...
if( (c = getchar( )) < 0 ) ....
```

works on the PDP–11, but fails on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, **lint** will say *nonportable character comparison.*

A similar issue arises with bit–fields. When assignments of constant values are made to bit–fields, the field may be too small to hold the value. This is especially true because, on some machines, bit–fields are considered as signed quantities. While it may seem unintuitive to consider that a two–bit field declared of type *int* cannot hold the value 3, the problem disappears if the bit–field is declared to have type *unsigned.*

Assignments of Type *long* to *int*

Bugs may arise from the assignment of *long* to an *int*, which loses accuracy. This may happen in programs that have been incompletely converted to use *typedefs.* When a *typedef* variable is changed from **int** to **long**, the program may stop working because some intermediate results are assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **—a** option.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by **lint**. The messages encourage better code quality, clearer style, and may even point out bugs. The **—h** option enables these checks. For example, in the statement

```
*p + + ;
```

the * does nothing; this provokes the message *null effect* from **lint**. The program fragment

```
unsigned x ;
if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x ! = 0 )
```

which may not be the intended action. **Lint** says *degenerate unsigned comparison* in these cases. If you enter:

```
if( 1 ! = 0 ) ....
```

lint reports *constant in conditional context*, since the comparison of 1 with 0 gives a constant result.

Another construction that **lint** detects involves operator precedence. Bugs that arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements:

```
if( x&077 = = 0 ) ...
```

or

```
x< < 2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this with an appropriate message.

Finally, when the **—h** option is in effect, **lint** complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many to be bad style, unnecessary, and frequently a bug.

History

There are several forms of older syntax that are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (for example, =+, =—) can cause ambiguous expressions, such as:

```
a =—1;
```

this can be interpreted as either:

```
a =— 1;
```

or

```
a = —1;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred, operators (+=, —=, etc.) have no such ambiguities. To spur the abandonment of the older forms, **lint** complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed:

```
int x 1;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example:

```
int x (—1);
```

This looks somewhat like the beginning of a function declaration:

```
int x (y) { ...
```

The compiler must read past this *x* to be sure what the declaration really is. Again, the problem is more perplexing when the initializer involves a macro. The current syntax places an equal sign (=) between the variable and the initializer:

```
int x = —1;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments are legal on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP–11, it is legal to assign integer pointers to double pointers, since double precision values can begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even–word boundaries; thus, not all such assignments make sense. **Lint** tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message *possible pointer alignment problem* results whenever either the **—p** or **—h** options are in effect.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP–11) in which the stack runs backwards, function arguments are probably best evaluated from right–to–left; on machines with a stack running forward, left–to–right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

In order not to compromise the efficiency of C on a particular machine, the C language leaves the order of evaluation of complicated expressions up to the local compiler. The various C compilers have considerable differences in the order in which they evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement:

```
a[i] = b[i++];
```

draws the complaint:

```
warning: i evaluation order undefined
```

Implementation

Lint consists of two programs and a driver. The first program is a version of the C Compiler. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file that is passed to a code generator, as the other compilers do, **lint** produces an intermediate file that consists of lines of ASCII text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second compiler then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process and is responsible for making the options available to both passes of **lint**.

Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations and discusses the **lint** features that encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as *int a ;* outside of any function. The UTek loader resolves these declarations and causes only a single word of storage to be set aside for *a*. Under the GCOS and IBM implementations, this is not feasible so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts, that prevent the proper operation of the program. If **lint** is invoked with the **—p** option, it detects such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UTek system, externally known names have seven significant characters, and distinguish between upper and lowercase. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UTek system, but encounter loader problems on the IBM or GCOS systems. **Lint —p** causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UTek system are 8-bit ascii, while they are 8-bit EBCDIC on the IBM, and 9-bit ascii on GCOS. Moreover, character strings go from high to low bit positions (left to right) on GCOS and IBM, and low to high (right to left) on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indexes into arrays, must be looked at with great suspicion. **Lint** is of little help here, except to flag multi-character character constants.

The different word sizes cause less trouble than might be expected, at least when moving from the UTek system (16-bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which works on all these machines. The right shift operator is arithmetic shift on the PDP–11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the PDP–11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP–11 hardware that has infiltrated itself into the C language. If there were a good way to discover the programs that would be affected, C could be changed; in any case, **lint** is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The most serious bar to the portability of UTek system utilities has been the inability to mimic essential UTek system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, **lint** has been very helpful in moving the UTek operating system and associated utility programs to other machines.

Silencing Lint

There are occasions when the programmer is smarter than **lint**. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by **lint** often has blind spots, causing occasional spurious messages about reasonable programs. So it is useful to communicate with **lint** and suppress parts of it.

Lint now recognizes a number of words when they are embedded in comments. The preprocessor passes comments through to its output, instead of deleting them as it did previously. Thus, **lint** directives are invisible to the compilers.

The first directive is concerned with control flow information. If a particular place in the program cannot be reached, but this is not apparent to **lint**, enter this at the appropriate place in the program:

```
/* NOTREACHED */
```

If you want to turn off strict type checking for the next expression, use the directive:

```
/* NOSTRICT */
```

The **—v** option can be turned on for one function by the directive:

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by this directive before the function definition:

```
/* VARARGS */
```

Sometimes you want to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments that should be checked:

```
/* VARARGS2 */
```

This causes the first two arguments to be checked, the others unchecked.

Finally, the directive:

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file.

Library Declaration Files

Lint accepts certain library directives, such as:

```
—ly
```

It also tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive:

```
/* LINTLIBRARY */
```

This is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed similarly to ordinary source files. The only difference is that functions that are defined on a library file, but are not used on a source file, draw no complaints. **Lint** does not simulate a full library search algorithm and complains if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file, which contains descriptions of the programs normally loaded when a C program is run. When the **—p** option is in effect, another file is checked containing descriptions of the standard I/O library routines, which are expected to be portable across various machines. The **—n** option can be used to suppress all library checking.

Yacc: Yet Another Compiler-Compiler

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" that it accepts. An input language can 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 provides a general tool for describing the input to a computer program. You specify the structures of the input, together with the 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 your application handled by this subroutine.

The input subroutine produced by Yacc calls a user–supplied routine to return the next basic input item. Thus, you can specify your 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.

Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. You prepare 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 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 : month_name day ',' year ;

Here, **date** , **month_name** , **day** , and **year** represent structures of interest in the input process; presumably, **month_name** , **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 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 are usually 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

```
month_name  :  'J' 'a' 'n'  ;
month_name  :  'F' 'e' 'b'  ;

        . . .

month_name  :  'D' 'e' 'c'  ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and `month_name` 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 `month_name` was seen; in this case, `month_name` 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 realively 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 favorably with similar systems; moreover, the constructions that 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. Yacc has been extensively used in numerous practical applications, including **lint**, and the Portable C Compiler.

The next several sections describe the basic process of preparing a Yacc specification. *Basic Specifications* describes the preparation of grammar rules, *Actions* the preparation of the user supplied actions associated with these rules, and *Lexical Analysis* the preparation of lexical analyzers. *How the Parser Works* describes the operation of the parser. *Ambiguity and Conflicts* discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. *Precedence* describes a simple mechanism for handling operator precedences in arithmetic expressions. *Error Handling* discusses error detection and recovery. *The Yacc Environment* discusses the operating environment and special features of the parsers Yacc produces. *Hints for Preparing Specifications* gives some suggestions that should improve the style and efficiency of the specifications. *Advanced Topics* discusses some advanced topics. *Yacc Input Syntax* gives a summary of the Yacc input form. *A Simple Example* has a brief example, and *An Advanced Example* gives an example using some of the more complicated features of Yacc.

Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in the topic *Lexical Analysis*, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations* , *(grammar) rules*, and *programs* . The sections are separated by double percent **%%** marks. (The percent sign (%) is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

declarations
%%
rules
%%
programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also. Thus, the smallest legal Yacc specification is:

%%
rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A : BODY ;

The *A* represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon (:) and the semicolon (;) are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot (.), underscore (_), and non–initial digits. Upper– and lower–case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes (' '). As in C, the backslash \ is an escape character within literals, and all the C escapes are recognized. Thus,

'\n'	newline
'\r'	return
'\''	single quote '
'\\'	backslash \
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	*xxx* in octal

For a number of technical reasons, the *null* character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left side, the vertical bar or pipe (|) can be used to avoid rewriting the left side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A       :       B  C  D   ;
A       :       E  F   ;
A       :       G   ;
```

can be given to Yacc as

```
A       :       B  C  D
        |       E  F
        |       G
        ;
```

If all grammar rules with the same left side appear together in the grammar rules section, the input is much more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

empty : ;

Names representing tokens must be declared; this is most simply by writing the following in the declarations section:

%token *name1 name2 . . .*

Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the **%start** keyword:

%start *symbol*

The end of the input to the parser is signaled by a special token, called the *endmarker* . If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the **parser** function returns to its caller after the endmarker is seen— it *accepts* the input. If the endmarker is seen in any other context, it is an error.

The user-supplied lexical analyzer should return the endmarker when appropriate; see *Actions* below. Usually the endmarker represents some reasonably obvious I/O status, such as *end-of-file* or *end-of-record*.

Actions

With each grammar rule, you can 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 braces ({ and }). For example,

```
A       :       '(' B ')'
                {       hello( 1, "abc'' );  }
```

and

```
XXX     :       YYY  ZZZ
                {       printf("a message\n");
                        option = 25;    }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The *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 parsed fully. 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 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* **)**

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.

You can define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the characters **%{** 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*; you should avoid such names.

In these examples, all the values are integers: a discussion of values of other types is found in *Advanced Topics*.

Lexical Analysis

You 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 you. In either case, the *# define* mechanism of C lets the lexical analyzer 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** is defined as the token number associated with the token **DIGIT**.

This mechanism leads to clear, easily modified lexical analyzers. The only condition 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.

As mentioned above, the token numbers may be chosen by Yacc or by you. 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 you. 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. 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) that do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

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 is not discussed here. The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, nevertheless makes 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 look–ahead 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 side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right side by the left 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 side symbol (A in this case), and the number of symbols on the right 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; it 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 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 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 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. Error recovery (as opposed to the detection of error) is covered in subsection *Error Handling*.

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 backslash 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

Follow the steps of the parser while it is 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 *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 side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. When consulting the description of state 0, and 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 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.

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. 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 you 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 Yacc applies disambiguating rules to produce a correct parser, you can also 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. Rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc produces 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

This 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, each conflict is 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 that 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 can shift into state 45. State 45 then has, as part of its description, the line

stat : IF (cond) stat ELSE_stat

since the **ELSE** has been shifted in this state. Back in state 23, the alternative action, described by a dot (.), 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 that can be reduced. In most states, there is, at most, one reduce action possible in the state, and this is the default command. If you encounter unexpected shift/reduce conflicts you should look at the verbose output to decide whether the default actions are appropriate.

Precedence

The rules given above for resolving conflicts are not sufficient 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, you specify 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 asterisk (*) 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* minuses (–); unary minus can 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. 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.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, you may need to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

Do not stop all processing when an error is found; 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 you 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 scans 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 does 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 is reduced, and any "cleanup" action associated with it is 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 has this effect. For example, suppose the action after **error** were to call some sophisticated resynchronization routine, supplied by you, 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, you can get control to deal with the error actions required by other portions of the program.

The Yacc Environment

When you input 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 you, 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.

You 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 you. 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 you (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 outputs 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, you may be able to set this variable by using a debugging system.

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.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints may help you:

- Use all uppercase letters for token names, all lowercase letters for nonterminal names.
- Put grammar rules and actions on separate lines. This lets either be changed without an automatic need to change the other.
- Put all rules with the same left side together. Put the left side in only once, and let all following rules begin with a vertical bar.
- Put a semicolon only after the last rule with a given left side, and put the semicolon on a separate line. This allows new rules to be easily added.
- Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in subsection *A Simple Example* is written following this style, as are the examples in the text of this paper (where space permits). You must make up your own mind about these stylistic questions; the central problem, however, is to make the rules visible through the action code.

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 is reduced for the first item only, and the second rule is 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, you 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.

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 option that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of zero or more declarations, followed by zero 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 option *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 represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit you to use words such as *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". You can try using the mechanism described in the last subsection, but it is difficult.

It is better that the keywords be *reserved*; that is, they should be forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

Advanced Topics

This section discusses a number of advanced features of Yacc.

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.

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. This also seems unstructured. 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.

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 is strictly type checked. The Yacc value stack (see Subsection *How the Parser Works*) is declared to be a *union* of the various types of values desired. You declare 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 automatically inserts the appropriate union name, so that no unwanted conversions takes place. In addition, type checking commands such as **Lint** are 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 you 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, you include 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> '+' '-'

causes 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 rarely arises.

The facilities in this subsection are not triggered until they are used; in particular, the use of **%type** turns 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.

Yacc Input Syntax

This subsection 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 a LR(2) grammar; the difficult 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 decides 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_IDENTIFIER**s.

```
       /*  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 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  ';'
        ;
```

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 );
}
```

An Advanced Example

This subsection gives an example of a grammar using some of the advanced features discussed in *Advanced Topics*. The desk calculator example in *A Simple Example* 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 *A Simple Example*; 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 (for example, 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 comma (,) 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 is 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 are 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;
                }
```

```
        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 ) );
        }

                *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;
```

Curses and Terminfo Package

Introduction

This chapter is an introduction to **curses** and **terminfo**. It is intended for the programmer who must write a screen–oriented program using the **curses** package. Several example programs are discussed. The example programs can be found in section 5D. This section also documents each **curses** function, and is intended as a reference.

For **curses** to produce terminal–dependent output, it has to know what kind of terminal you have. The UTek system convention for this is to put the name of the terminal in the variable TERM in the environment. Thus, a user on a DEC VT100 would set *TERM = vt100* when logging in. **Curses** uses this convention.

Output

A program using **curses** always starts by calling **iinitscr().** Other modes can then be set as needed by the program. Possible modes include **cbreak**(), and **idlok(stdscr, TRUE).** These modes will be explained later. During the execution of the program, output to the screen is done with routines such as **addch(ch)** and **printw(fmt,args)**. (These routines behave just like **putchar** and **printf** except that they go through **curses**). The cursor can be moved with the call **move(row,col)**. These routines only output to a data structure called a *window*, not to the actual screen. A window is a representation of a video terminal screen, containing such things as an array of characters to be displayed on the screen, a cursor, a current set of video attributes, and various modes and options. You don't need to worry about windows unless you use more than one of them, except to realize that a window is buffering your requests to output to the screen.

To send all accumulated output call **refresh**().

(This can be thought of as a **flush**.) Finally, before the program exits, it should call **endwin()**, which restores all terminal settings and positions the cursor at the bottom of the screen. See Example 5C–1 for the framework of a **curses** program.

```
#include <curses.h>
        initscr();          /* Initialization */

        cbreak();           /* Various optional mode settings */
        nonl();
        noecho();
        while (!done)  {/* Main body of program */
                ...
                /* Sample calls to draw on screen */
                move(row, col);
                addch(ch);
                printw("Formatted print with value %d\n",
                value);
                ...
                /* Flush output */
                refresh();
                ...
        }

        endwin();           /* Clean up */
        exit(0);
```

Example 5C-1. Framework of a Curses Program.

See the program **scatter** in Section 5D for an example program. This program reads a file, and displays the file in a random order on the screen. Some programs assume all screens are 24 lines by 80 columns. It is important to understand that many are not. The variables LINES and COLS are defined by **initscr** with the current screen size. Programs should use them instead of assuming a 24x80 screen.

No output to the terminal actually happens until **refresh** is called. Instead, routines such as **move** and **addch** draw on a window data structure called **stdscr** (standard screen). **Curses** always keeps track of what is on the physical screen, as well as what is in **stdscr**.

When **refresh** is called, **curses** compares the two screen images and sends a stream of characters to the terminal that will turn the current screen into what is desired. **Curses** considers many different ways to do this, taking into account the various capabilities of the terminal, and similarities between what is on the screen and what is desired. It usually outputs as few characters as possible. This feature is called *cursor optimization*, and it is the source of the name of the **curses** package.

NOTE

Due to the hardware scrolling of terminals, writing to the lower righthand character position is impossible.

Input

Curses can do more than just draw on the screen. Functions are also provided for input from the keyboard. The primary function is **getch()** which waits for the user to type a character on the keyboard, and then returns that character. This function is like **getchar** except that it goes through **curses**. Its use is recommended for programs using the **cbreak()** or **noecho()** options, since several terminal- or system-dependent options become available that are not possible with **getchar**.

Options available with **getch** include **keypad**, which allows extra keys such as arrow keys, function keys, and other special keys that transmit escape sequences, to be treated as just another key. (The values returned for these keys are listed in Table 5C-1.) The values for these keys are over octal 400, so they should be sorted in a variable larger than a *char*. *Nodelay* mode causes the value -1 to be returned if there is no input waiting. Normally, **getch** will wait until a character is typed. Finally, the routine **getstr(str)** can be called, allowing input of an entire line, up to a newline. This routine handles echoing and the erase and kill characters of the user. Examples of the use of these options are in later example programs.

The function keys might be returned by **getch** if **keypad** has been enabled. Note that not all of these are currently supported, due to lack of definitions in **terminfo** or the terminal not transmitting a unique code when the key is pressed.

Table 5C-1
VALUES RETURNED BY KEYPAD KEYS

Name	Value	Key name
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward + left arrow)
KEY_BACKSPACE	0407	Backspace (unreliable)
KEY_F0	0410	Function keys
KEY_F(n)	(KEY_FO + (n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	Soft reset (unreliable)
KEY_RESET	0531	Reset or hard reset (unreliable)
KEY_PRINT	0532	Print or copy
KEY_LL	0533	Home down or bottom (lower left)

See the program **show** in the Section 5D for an example use of **getch. Show** pages through a file, showing one screen full each time the user presses the space bar. By creating an input file for **show** made up of 24 line pages, each segment varying slightly from the previous page, nearly any exercise for **curses** can be created. Such input files are called *show scripts.*

In the **show** program, **cbreak** is called so that the user can press the space bar without having to press <RETURN>. The **noecho** function is called to prevent the space from echoing in the middle of **refresh**, and garbling the screen. The **nonl** function is called to enable more screen optimization. The **idlok** function is called to allow insert and delete line, since many **show** scripts are constructed to duplicate bugs caused by that feature. The **clrtoeol** and **clrtobot** functions clear from the cursor to the end of the line and screen, respectively.

Highlighting

The function **addch** always draws two things on a window. In addition to the character itself, a set of *attributes* is associated with the character. These attributes cover various forms of highlighting of the character. For example, the character can be put in reverse video, bold, or be underlined. You can think of the attributes as the color of the ink used to draw the character.

A window always has a set of *current attributes* associated with it. The current attributes are associated with each character as it is written to the window. The current attributes can be changed with a call to **attrset(attrs).** The names of the attributes are A_STANDOUT, A_REVERSE, A_BOLD, A_DIM, A_INVIS, and A_UNDERLINE. For example, to put a word in bold, the code in Example 5C-2 might be used. The word *boldface* will be shown in bold.

```
printw("A work in ");
attrset( A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out. \n");
refresh( );
```

Example 5C-2. Use of Attributes.

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, **curses** will attempt to find a substitute attribute. If none is possible, the attribute is ignored.

The *standout* attribute is used to make text attract the attention of the user. The particular hardware attribute used for *standout* varies from terminal to terminal, and is chosen to be the most visually pleasing attribute the terminal has. *Standout* is typically implemented as reverse video or bold. Many programs do not really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the A_STANDOUT attribute is recommended. Two convenient functions, **standout()** and **standend()** turn on and off this attribute.

Attributes can be turned on in combination. Thus, to turn on blinking bold text, use **attrset(A_BLINK ¦A_BOLD).** Individual attributes can be turned on and off with **attron** and **attroff** without affecting other attributes.

For an example program using attributes, see **highlight** in Section 5D. The program takes a text file as input and allows embedded escape sequences to control attributes. In this example program, \U turns on underlining, \B turns on bold, and \N restores normal text. Note the initial call to **scrollok**. This lets the terminal scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, **curses** automatically scrolls the terminal up a line and calls **refresh**.

Highlight comes about as close to being a filter as is possible with **curses**. It is not a true filter, because **curses** must take over the video screen. In order to determine how to update the screen, it must know what is on the screen at all times. This requires **curses** to clear the screen in the first call to **refresh**, and to know the cursor position and screen contents at all times.

Multiple Windows

A window is a data structure representing all or part of the video screen. It has room for a two–dimensional array of characters, attributes for each character (a total of 16 bits per character: 7 for text and 9 for attributes) a cursor, a set of current attributes, and a number of flags. **Curses** provides a full screen window, called **stdscr**, and a set of functions that use **stdscr**. Another window is provided called **curscr**, representing the physical screen.

It is important to understand that a window is only a data structure. Use of more than one window does not imply use of more than one terminal, nor does it involve more than one process. A window is merely an object that can be copied to all or part of the terminal screen. The current implementation of **curses** does not allow windows that are bigger than the screen.

You can create additional windows with the function **newwin(lines, cols, begin_row, begin_col)** and can return a pointer to a newly–created window. The window is be **lines** by **cols**, and the upper left corner of the window is be at screen position **(begin_row, begin_col).** All operations that affect **stdscr** have corresponding functions that affect an arbitrary, named window. Generally, these functions have names formed by putting a "w" on the front of the **stdscr** function, and the window name is added as the first parameter. Thus, **waddch(mywin, c)** would write the character **c** to window **mywin**. The **wrefresh(win)** function is used to flush the contents of a window to the screen.

Windows are useful for maintaining several different screen images, and letting the user alternate among them. You can also subdivide the screen into several windows, refreshing each of them as desired. When windows overlap, the contents of the screen will be the more–recently refreshed window.

In all cases, the non–w version of the function calls the w version of the function, using **stdscr** as the additional argument. Thus a call to **addch(c)** results in a call to **waddch(stdscr, c).**

The program **window** in Section 5D is an example of the use of multiple windows. The main display is kept in **stdscr**. When the user temporarily wants to put something else on the screen, a new window is created covering part of the screen. A call to **wrefresh** on that window causes the window to be written over **stdscr** on the screen. Calling **refresh** or **stdscr** results in the original window being redrawn on the screen. Note the calls to **touchwin** before writing out an overlapping window. These are necessary to defeat an optimization in **curses**. If you have trouble refreshing a new window that overlaps an old window, it you may need to call **touchwin** on the new window to get it completely written out.

For convenience, a set of move functions is also provided for most of the common functions. These result in a call to **move** before the other function. For example, **mvaddch(row, col, c)** is the same as **move(row, col); addch(c).** Combinations also exist; for example, **mvwaddch(row, col, win, c).**

Multiple Terminals

Curses can produce output on more than one terminal at once. This is useful for single process programs that access a common database, such as multiplayer games. Output to multiple terminals is a difficult business, and **curses** does not solve all the problems for the programmer. The program must determine the filename of each terminal line, and what kind of terminal is on each of those lines. The standard method (checking $TERM in the environment) does not work, since each process can only examine its own environment. Another problem that must be solved is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. Nonetheless, a program wishing to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. However, for some applications, such as an inter–terminal communication program, or a program that takes over unused tty lines, it would be appropriate). A typical solution requires the user logged in on each line to run a program that notifies the master program that the user is interested in joining the master program, telling it the notification program's process id, the name of the tty line and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program, and all programs exit.

Curses handles multiple terminals by always having a *current terminal.* All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals have type **struct screen ***. A new terminal is initialized by calling **newterm(type, fd). newterm** returns a screen reference to the terminal being set up. **type** is a character string, naming the kind of terminal being used. The parameter *fd* is a file descriptor to be used for input and output to the terminal. (If only output is needed, the file can be open for output only). This call replaces the normal call to **initscr**, which calls **newterm(getenv("TERM"), stdout).**

To change the current terminal, call **set_term***(sp)* where *sp* is the screen reference to be made current. **set_term** returns a reference to the previous terminal.

Each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**. Options such as **cbreak** and **noecho** must be set separately for each terminal. The functions **endwin** and **refresh** must be called separately for each terminal. See Example 5C–3 for a typical scenario to output a message to each terminal.

```
for (i=0; i<nterm; i++) {
            set_term(terms[i]);
            mvaddstr(0, 0, "Important message");
            refresh( );
}
```

Example 5C–3. Sending a Message to Several Terminals.

See the sample program **two** in section 5D for a full example. This program pages through a file, showing one page to the first terminal and the next page to the second terminal. It then waits for a space to be entered on either terminal, and shows the next page to the terminal typing the space. Each terminal has to be separately put into nodelay mode. Since no standard multiplexor is available in current versions of the UTek system, it is necessary to either busy wait, or call **sleep**; between each, check for keyboard input. This program sleeps for a second between checks.

The **two** program is just a simple example of two terminal **curses**. It does not handle notification, as described above. Instead, it requires the name and type of the second terminal on the command line. As written, the command **sleep 100000** must be entered on the second terminal to put it to sleep while the program runs, and the first user must have both read and write permission on the second terminal.

Low–level Terminfo Usage

Some programs need to use lower–level primitives than those offered by **curses**. For such programs, the *terminfo level interface* is offered. This interface does not manage your video screen, but rather gives you access to strings and capabilities that you can use yourself to manipulate the terminal.

Normally you should not use this level. Whenever possible, the higher level **curses** routines should be used. This makes your programs more portable to other UTek systems and to a wider class of terminals. **Curses** takes care of all the glitches and misfeatures present in physical terminals, but at the **terminfo** level you must deal with them yourself. Also, this part of the interface might change.

There are two circumstances when it is proper to use **terminfo**. The first is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. The second situation is when writing a filter. A typical filter does one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal–dependent and clearing the screen is inappropriate, use of **terminfo** is indicated.

A program writing at the terminfo level uses the framework shown in Example 5C–4.

```
#include <curses.h>
#include <term.h>
            setupterm(0, 1, 0);
            ...
            putp(clear_screen);
            ...
            reset_shell_mode( );
            exit(0);
```

Example 5C–4. Terminfo Level Framework.

Initialization is done by calling **setupterm**. Passing the values 0, 1, and 0 invoke reasonable defaults. If **setupterm** cannot figure out what kind of terminal you are on, it prints an error message and exits. The program should call **reset_shell_mode** before it exits.

Global variables with names like **clear_screen** and **cursor_address** are defined by the call to **setupterm**. They can be output using **putp**, or also using **tputs**, which allows the programmer more control. These strings should not be directly output to the terminal using **printf** since they contain padding information. A program that directly outputs strings will fail on terminals that require padding, or that use the XON/XOFF flow control protocol.

In the **terminfo** level, the higher–level routines described previously are not available. It is up to the programmer to output whatever is needed. For a list of capabilities and a description of what they do, see the *UTek Command Reference, terminfo(4).*

The example program **termhl** shows simple use of **terminfo**. It is a version of **highlight** that uses **terminfo** instead of **curses**. This version can be used as a filter. The strings to enter bold and underline mode, and to turn off all attributes, are used.

This program is more complex than it needs to be in order to illustrate some properties of **terminfo**. The routine **vidattr** could have been used instead of directly outputting **enter_bold_mode, enter_underline_mode,** and **exit_attribute_mode**. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes. This program was written to illustrate typical use of **terminfo**.

The function **tputs(cap, affcnt, outc)** applies padding information. Some capabilities contain strings like $<20>, which means to pad for 20 milliseconds. **Tputs** generates enough pad characters to delay for the appropriate time. The first parameter is the string capability to be output. The second is the number of lines affected by the capability.

NOTE

Some capabilities may require padding that depends on the number of lines affected. For example, **insert-line** *may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention,* **affcnt** *is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since* **affcnt** *is multiplied by the amount of time per item, and anything multiplied by 0 is 0.*

The third parameter is a routine to be called with each character.

For many simple programs **affcnt** is always 1 and **outc** always calls **putchar**. For these programs, the routine **putp(cap)** is a convenient abbreviation. The program **termhl** could be simplified by using **putp**.

Note also the special check for the **underline_char** capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. The program **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, it outputs **underline_char** if necessary. Details such as this are precisely why the **curses** level is recommended over the **terminfo** level. Programs at the **terminfo** level must handle such details themselves.

A Larger Example

For a final example, see the program **editor** in Section 5D. This program is a very simple screen editor, patterned after the **vi** editor. The program shows how to use **curses** to write a screen editor. This editor keeps the buffer in **stdscr** to keep the program simple — obviously a real screen editor would keep a separate data structure. Many simplifications have been made here: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

The routine to write out the file shows the use of the **mvinch** function, which returns the character in a window at a given position. The data structure used here does not keep track of the number of characters in a line, or the number of lines in the file, so trailing blanks are eliminated when the file is written out.

The program uses the built-in **curses** functions **insch, delch, insertln,**and **deleteln.** These functions behave much as the similar functions on intelligent terminals behave, inserting and deleting a character or line.

The command interpreter accepts not only ASCII characters, but also special keys. This is important— a good program accepts both. It is important to handle special keys because this makes it easier for new users to learn your program if they can use the arrow keys, instead of having to memorize that **h** means left, **j** means down, **k** means up, and **l** means right. On the other hand, not all terminals have arrow keys, so your program will be usable on a larger class of terminals if there is an ASCII character which is a synonym for a special key. Also, experienced users dislike having to move their hands from the "home row" position to use special keys, since they can work faster with the alphabetic keys.

Note the call to **mvaddstr** in the input routine. **Addstr** is roughly like the C **fputs** function, which writes out a string of characters. Like **fputs**, **addstr** does not add a trailing newline character. It is the same as a series of calls to **addch** using the characters in the string. **Mvaddstr** is the **mv** version of **addstr**, which moves to the given location in the windows before writing.

The <CTRL–L> command illustrates a feature most programs using **curses** should add. Often some program beyond the control of **curses** has written something to the screen, or some line noise has messed up the screen beyond what **curses** can keep track of. In this case, the user usually types <CTRL–L>, causing the screen to be cleared and redrawn. This is done with the call to **clearok(curscr)**, which sets a flag causing the next **refresh** to first clear the screen. Then **refresh** is called to force the redraw.

Note also the call to **flash()**, which flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within earshot. The routine **beep()** can be called when a real beep is desired. (If, for some reason, the terminal is unable to beep, but able to flash, a call to **beep** will flash the screen.)

Another important point is that the input command is terminated by <CTRL–D>, not <ESC>. It is very tempting to use <ESC> as a command, since <ESC> is one of the few special keys that is available on every keyboard. (Return and break are the only others.) However, using <ESC> as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with <ESC> (escape sequences) to control the terminal, and have special keys that send escape sequences to the computer. If the computer sees an <ESC> coming from the terminal, it cannot tell for sure whether the user pushed <ESC>, or whether a special key was pressed. Curses handles the ambiguity by waiting for up to one second. If another character is received during this second, and if that character might be the beginning of a special key, more input is read (waiting for up to one second for each character) until either a full special key is read, one second passes, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press <ESC>, then to type another key quickly, which causes **curses** to think a special key has been pressed. Also, there is a one second pause until the <ESC> can be passed to the user program, resulting in slower response to the <ESC> key. Many existing programs use escape as a fundamental command, which cannot be changed without frustrating users. Such programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time–out solution. The moral is clear: when designing your program, avoid the <ESC> key.

List of Routines

This section describes all the routines available to the programmer in the **curses** package. The routines are organized by function.

Structure

All programs using **curses** should include the file *curses.h*. This file defines several **curses** functions as macros, and defines several global variables and the data type WINDOW. References to windows are always of type WINDOW *. **Curses** also defines WINDOW * constants **stdscr** (the standard screen, used as a default to routines expecting a window), and **curscr** (the current screen, used only for certain low–level operations like clearing and redrawing a messed up screen). Integer constants LINE and COLS are defined, containing the size of the screen. Constants TRUE and FALSE are defined, with values 1 and 0 respectively. Additional constants that are values returned from most **curses** functions are ERR and OK. OK is returned if the function could be properly completed, and ERR is returned if there was some error, such as moving the cursor outside of a window.

The include file *curses.h* automatically includes *stdio.h* and an appropriate tty driver interface file, currently either *agtty.h** or *termio.h*.

NOTE

Including stdio.h again is harmless but wasteful. Including sgtty.h again will usually result in a fatal error.

A program using **curses** should include the loader option **-lcurses** in the makefile. This is true for both the **terminfo** level and the **curses** level. The compilation option **-DMINICURSES** can be included if you restrict your program to a small subset of **curses** concerned primarily with screen output and optimization. The routines possible with **minicurses** are listed in the later topic *Minicurses*.

Initialization

These functions are called when initializing a program.

initscr()

The first function called always should be **initscr**. This determines the terminal type and initializes **curses** data structures. **Initscr** also arranges that the first call to **refresh** will clear the screen.

endwin()

A program should always call **endwin()** before exiting. This function will restore tty modes, move the cursor to the lower left corner of the screen, reset the terminal into the proper non-visual mode, and tear down all appropriate data structures.

newterm(type, fd)

A program that outputs to more than one terminal should use **newterm** instead of **initscr**. **Newterm** should be called once for each terminal. It returns a variable of type SCREEN *, which should be saved as a reference to that terminal. The arguments are the type of the terminal (a string) and a stdio file descriptor (FILE*) for output to the terminal. The file descriptor should be open for both reading and writing if input from the terminal is desired. The program should also call **endwin** for each terminal being used (see **set_term** below). If an error occurs, the value NULL is returned.

set_term(new)

This function is used to switch to a different terminal. The screen reference **new** becomes the new current terminal. The previous terminal is returned by the function. All other calls affect only the current terminal.

longname()

This function returns a pointer to a static area containing a verbose description of the current terminal. It is defined only after a call to **initscr, newterm**, or **setupterm**.

Setting Options

These functions set options within **curses**. In each case, **win** is the window affected, and **bf** is a Boolean flag with value TRUE or FALSE indicating whether to enable or disable the option. All options are initially FALSE. It is not necessary to turn these options off before calling **endwin**.

clearok(win,bf)
If set, the next call to **wrefresh** with this window clears the screen and redraw the entire screen. If **win** is **curscr**, the next call to **wrefresh** with any window causes the screen to be cleared. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win,bf)
If enabled, **curses** considers using the hardware to insert/delete line feature of terminals so equipped. If disabled, **curses** never uses this feature. The insert/delete character feature is always considered. Enable this option only if your application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it is not really necessary. If insert/delete line cannot be used, **curses** redraws the changed portions of all lines that do not match the desired line.

keypad(win,bf)
This option enables the keypad of the user's terminal. If enabled, the user can press a function key and **getch** returns a single value representing the function key. If disabled, **curses** does not treat function keys specially. If the keypad in the terminal can be turned on and off, turning on this option turns on the terminal keypad.

leaveok(win,bf)
Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

meta(win,bf)
If enabled, characters returned by **getch** are transmitted with all 8 bits, instead of stripping the highest bit. The value OK is returned if the request succeeded; the value ERR is returned if the terminal or system is not capable of 8–bit input.

Meta mode is useful for extending the nontext command set in applications where the terminal has a meta shift key. **Curses** takes whatever measures are necessary to arrange for 8–bit input. On other versions of UNIX systems, raw mode is used. On this system, the character size is set to 8, parity checking disabled, and stripping of the 8th bit turned off.

Note that 8–bit input is a fragile mode. Many programs and networks only pass 7 bits. If any link in the chain from the terminal to the application program strips the 8th bit, 8–bit input is impossible.

nodelay(win,bf)
This option causes **getch** to be a nonblocking call. If no input is ready, **getch** returns —1. If disabled, **getch** hangs until a key is pressed.

intrflush(win,bf)
If this option is enabled when an interrupt key is pressed on the keyboard, all output in the tty driver queue is flushed, giving the effect of faster response to the interrupt but causing **curses** to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default is for the option to be enabled. This option depends on support in the underlying teletype driver.

typeahead(fd)
Sets the file descriptor for type–ahead check. The parameter *fd* should be an integer returned from **open** or **fileno**. Setting type–ahead to –1 disables type–ahead check. By default, file descriptor 0 (stdin) is used. Type–ahead is checked independently for each screen, and for multiple interactive terminals it should probably be set to the appropriate input for each screen. A call to **typeahead** always affects only the current screen.

scrollok(win,bf)
This option controls what happens when the cursor of a window is moved off the edge of the window, either from a newline on the bottom line, or typing the last character of the last line. If disabled, the cursor is left on the bottom line. If enabled, **wrefresh** is called on the window, and then the physical terminal and window are scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**.

setscrreg(t,b)
wsetscrreg(win,t,b)
These functions allow the user to set a software scrolling region in a window **win** or **stdscr**. **t** and **b** are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok** are enabled, an attempt to move off the bottom margin line causes all lines in the scrolling region to scroll up one line. Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the VT100. Only the text of the window is scrolled. If **idlok** is enabled and the terminal has either a scrolling region or insert/delete line capability, they probably are used by the output routines.

Terminal Mode Setting

These functions are used to set modes in the tty driver. The initial mode usually depends on the setting when the program was called; the initial modes documented here represent the normal situation.

cbreak()
nocbreak()
These two functions put the terminal into and out of CBREAK mode. In this mode, characters typed by the user are immediately available to the program. When out of this mode, the teletype driver buffers characters typed until a newline is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal is not in CBREAK mode. Most interactive programs using **curses** set this mode.

echo()
noecho()
These functions control whether characters typed by the user are echoed as typed. Initially, characters typed are echoed by the teletype driver. Authors of many interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing.

nl()
nonl()
These functions control whether newline is translated into carriage return and linefeed on output, and whether <RETURN> is translated into newline on input. Initially, the translations do occur. By disabling these translations, **curses** is able to make better use of the linefeed capability, resulting in faster cursor motion.

raw()
noraw()
The terminal is placed into or out of raw mode. *Raw* mode is similar to cbreak mode in that characters typed are immediately passed through to the user program. The differences are that in RAW mode, the interrupt, quit, and suspend characters are passed through uninterpreted instead of generating a signal. RAW mode also causes 8–bit input and output. The behavior of the <BREAK> key may be different on different systems.

resetty()
savetty()
These functions save and restore the state of the tty modes. **Savetty** saves the current state in a buffer, **resetty** restores the state to what it was at the last call to **savetty**.

Window Manipulation

newwin(num_lines, num_cols, beg_row, beg_col)
Create a new window with the given number of lines and columns. The upper left corner of the window is at line **beg_row** column **beg_col**. If either **num_lines** or **num_cols** is zero, they are defaulted to **LINES–beg_row** and **COLS–beg_col**. A new full–screen window is created by calling **newwin(0,0,0)**.

newpad(num_lines, num_cols)
Creates a new *pad* data structure. A pad is like a window, except that it is not restricted by the screen size, and is not associated with a particular part of the screen. Pads can be used when you need a large window, and only a part of the window is on the screen at one time. Automatic refreshes of pads (for example, from scrolling or echoing of input) do not occur. It is not legal to call **refresh** with a pad as an argument; the routines **prefresh** or **pnoutrefresh** should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

subwin(orig, num_lines, num_cols, begy, begx)
Create a new window with the given number of lines and columns. The window is at position (*begy, begx*) on the screen. (It is relative to the screen, not **orig**.) The window is made in the middle of the window **orig**, so that changes made to one window affect both windows. When using this function, often it is necessary to call **touchwin** before calling **wrefresh**.

delwin(win)
Deletes the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

mvwin(win, br, bc)
Move the window so that the upper left corner is at position (**br, bc**). If the move would cause the window to be off the screen, it is an error and the window is not moved.

touchwin(win)
Throw away all optimization information about which parts of the window have been touched, by pretending the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window affects the other window, but the records of which lines have been changed in the other window do not reflect the change.

overlay(win1, win2)
overwrite(win1, win2)
These functions overlay **win1** on top of **win2**; that is, all text in **win1** is copied into **win2**. The difference is that **overlay** is nondestructive (blanks are not copied), while **overwrite** is destructive.

Causing Output to the Terminal

refresh()
wrefresh(win)
These functions must be called to get any output on the terminal, as other routines merely manipulate data structures. **Wrefresh** copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. **Refresh** is the same, using **stdscr** as a default screen. Unless **leaveok** has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

doupdate()
wnoutrefresh(win)
These two functions allow multiple updates with more efficiency than **wrefresh.** To use them, you must understand how **curses** works. In addition to all the window structures, **curses** keeps two data structures representing the terminal screen: a *physical* screen, describing what is actually on the screen, and a *virtual* screen, describing what the programmer wants to have on the screen. **Wrefresh** works by first copying the named window to the virtual screen **(wnoutrefresh),** and then calling the routine to update the screen **(doupdate).** If the programmer wishes to output several windows at once, a series of calls to **wrefresh** will result in alternating calls to **wnoutrefresh** and **doupdate,** causing several bursts of output to the screen. By calling **wnoutrefresh** for each window, it is then possible to call **doupdate** once, resulting in only one burst of output, with probably fewer total characters transmitted.

prefresh(pad,pminrow,pmincol,sminrow,
smincol,smaxrow,smaxcol)
pnoutrefresh(pad,pminrow,pmincol,sminrow,
smincolm,smaxrow,smaxcol)
These routines are analogous to **wrefresh** and **wnoutrefresh** except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. **Pminrow** and **pmincol** specify the upper left corner, in the pad, of the rectangle to be displayed. **Sminrow, smincol, smaxrow,** and **smaxcol** specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures.

Writing on Window Structures

These routines are used to "draw" text on windows. In all cases, a missing **win** is taken to be **stdscr. y** and **x** are the row and column, respectively. The upper left corner is always (0,0), not (1,1). The **mv** functions imply a call to **move** before the call to the other function.

Moving the Cursor

move(y, x)
wmove(win, y, x)
The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until **refresh** is called. The position specified is relative to the upper left corner of the window.

Writing One Character

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)
The character *ch* is put in the window at the current cursor position of the window. If *ch* is a tab, newline, or backspace, the cursor is moved appropriately in the window. If *ch* is a different control character, it is drawn in the <CTRL–X> notation. The position of the window cursor is advanced. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok** is enabled, the scrolling region is scrolled up one line.

The *ch* parameter is actually an integer, not a character. Video attributes can be combined with a character by **or**–ing them into the parameter. This results in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another with **inch** and **addch**.)

Writing a String

addstr(str)
waddstr(win,str)
mvaddstr(y,x,str)
mvwaddstr(win,y,x,str)

These functions write all the characters of the null terminated character string **str** on the given window. They are identical to a series of calls to **addch**.

Clearing Areas of the Screen

erase( )
werase(win)
These functions copy blanks to every position in the window.

clear( )
wclear(win)
These functions are like **erase** and **werase** but they also call **clearok**, arranging that the screen is cleared on the next call to **refresh** for that window.

clrtobot( )
wclear(win)
All lines below the cursor in this window are erased. Also, the current line to the right of the cursor is erased.

clrtoeol( )
wclrtoeol(win)
The current line to the right of the cursor is erased.

Inserting and Deleting Text

delch( )
wdelch(win)
mvdelch(y,x)
mvwdelch(win,y,x)
The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position. This does not imply use of the hardware delete character feature.

deleteln( )
wdeleteln(win)
The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. This does not imply use of the hardware delete line feature.

insch(c)
winsch(win, c)
mvinsch(y,x,c)
mvwinsch(win,y,x,c)
The character **c** is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. This does not imply use of the hardware insert character feature.

insertln( )
winsertln(win)
A blank line is inserted above the current line. The bottom line is lost. This does not imply use of the hardware insert line feature.

Formatted Output

printw(fmt, args)
wprintw(win, fmt, args)
mvprintw(y, x, fmt, args)
mvwprintw(win, y, x, fmt, args)
These functions correspond to **printf**. The characters which would be output by **printf** are instead output using **waddch** on the given window.

Miscellaneous

box(win, vert, hor)
A box is drawn around the edge of the window. The box is drawn with the **vert** and **hor** characters.

scroll(win)
The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen is scrolled at the same time.

Input from a Window

getwx(win,y,x)
The cursor position of the window is placed in the two integer variables *y* and *x*. Since this is a macro, no ampersand character (&) is necessary.

inch( )
winch(win)
mvinch(y,x)
mvwinch(win,y,x)
The character at the current position in the named window is returned. If any attributes are set for that position, their values will be **or**–ed into the value returned. The predefined constants A_ATTRIBUTES and A_CHARTEXT can be used with the & operator to extract the character to attributes alone.

Input from the Terminal

getch( )
wgetch(win)
mvgetch(y,x)
mvwgetch(win,y,x)
A character is read from the terminal associated with the window. In **nodelay** mode, if there is no input waiting, the value —1 is returned. In *delay* mode, the program hangs until the system passes text through to the program. Depending on the setting of **cbreak**, this is after one character, or after the first newline.

If **keypad** mode is enabled, and a function key is pressed, the code for that function key is returned instead of the raw characters. Possible function keys are defined with integers beginning with 0401, whose names begin with **KEY_**. These are listed in the previous topic *Input*. If a character is received that could be the beginning of a function key (such as <ESC>), **curses** sets a one–second timer. If the remainder of the sequence does not come in within one second, the character is passed through; otherwise the function key value is returned. For this reason, on many terminals, there is a one second delay after you press the <ESC> key. (Use by a programmer of the <ESC> key for a single character function is discouraged.)

getstr(str)
wgetstr(win,str)
mvgetstr(y,x,str)
mvwgetstr(win,y,x,str)
A series of calls to **getch** is made, until a newline is received. The resulting value is placed in the area pointed at by the character pointer **str**. The user's erase and kill characters are interpreted.

scan(fmt, args)
wscanw(win, fmt, args)
mvscanw(y, x, fmt, args)
mvwscanw(win, y, x, fmt, args)
This function corresponds to **scanf**. **Wgetstr** is called on the window, and the resulting line is used as input for the scan.

Video Attributes

attroff(at)
wattroff(win, attrs)
attron(at)
wattron(win, attrs)
attrset(at)
wattrset(win, attrs)
standout( )
standend( )
wstandout(win)
wstandend(win)
These functions set the *current attributes* of the named window. These attributes can be any combination of A_STANDOUT, A_REVERSE, A_BOLD, A_DIM, A_BLINK, and A_UNDERLINE. These constants are defined in <curses.h> and can be combined with the C ¦ (or) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch**. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of characters put on the screen.

attrset(at) sets the current attributes of the given window to **at**. **attroff**(at) turns off the named attributes without affecting any other attributes. **attron**(at) turns on the named attributes without affecting any others. **Standout** is the same as **attron(A_STANDOUT)**. **standend** is the same as **attrset**(0); that is, it turns off all attributes.

Bells and Flashing Lights

beep( )
flash( )
These functions are used to signal the programmer. **Beep** sounds the audible alarm on the terminal, if possible, and if not, flashes the screen (visible bell), if that is possible. **Flash** flashes the screen, and if that is not possible, sounds the audible signal. If neither signal is possible nothing happens. Nearly all terminals have an audible signal (bell or beep), but only some can flash the screen.

Portability Functions

These functions do not directly involve terminal–dependent character output but tend to be needed by programs that use **curses**. Unfortunately, their implementation varies from one version of UNIX * to another. They have been included here to enhance the portability of programs using **curses**.

baudrate( )
baudrate returns the output speed of the terminal. The number returned is the integer baud rate, for example, 9600, rather than a table index such as B9600.

erasechar( )
The erase character chosen by the user is returned. This character is typed by the user to erase the character just entered.

killchar( )
The line kill character chosen by the user is returned. This character is typed by the user to forget the entire line being entered.

flushinp( )
Flushinp throws away any type–ahead that has been typed by the user and has not yet been read by the program.

Delays

These functions are usually not portable, but are often needed by programs that use **curses**, especially real–time response programs. Some of these functions require a particular operating system or a modification to the operating system to work. In all cases, the routine compiles and returns an error status if the requested action is not possible. Programmers should avoid use of these functions if possible.

draino(ms)
The program is suspended until the output queue has drained enough to complete in *ms* additional milliseconds. Thus, **draino**(50) at 1200 baud would pause until there are no more than six characters in the output queue, because it would take 50 milliseconds to output the additional characters. The purpose of this routine is to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the input/output to implement **draino**, the value ERR is returned; otherwise, OK is returned.

napms(ms)
This function suspends the program for *ms* milliseconds. It is similar to **sleep** except with higher resolution. The resolution actually provided varies with the facilities available in the operating system, and often a change to the operating system is necessary to produce good results. If resolution of at least 0.1 second is not possible, the routine rounds to the next higher second, calls **sleep**, and returns ERR. Otherwise, the value OK is returned. Often, the resolution provided is 1/60th second.

Lower Level Functions

These functions are provided for programs not needing the screen optimization capabilities of **curses**. Programs are discouraged from working at this level, since they must handle various glitches in certain terminals. However, a program can be smaller if it only brings in the low-level routines.

Cursor Motion

mvcur(oldrow, oldcol, newrow, newcol)
This routine optimally moves the cursor from (oldrow, oldcol) to (newrow, newcol). The user program is expected to keep track of the current cursor position. Note that unless a full screen image is kept, **curses** has to make pessimistic assumptions, sometimes resulting in less than optimal cursor motion. For example, moving the cursor a few spaces to the right can be done by transmitting the characters being moved over; but if **curses** does not have access to the screen image, it doesn't know what these characters are.

Terminfo Level

These routines are called by low-level programs that need access to specific capabilities of **terminfo**. A program working at this level should include both <curses.h> **and** <**term.h**> in that order. After a call to **setupterm,** the capabilities will be available with macro names defined in <**term.h**>. See *terminfo*(4) in the *Command Reference* for a detailed description of the capabilities.

Boolean-valued capabilities have the value 1 if the capability is present, and 0 if it is not. Numeric capabilities have the value -1 if the capability is missing, and have a value at least 0 if it is present. String capabilities (both those with and without parameters) have the value NULL if the capability is missing, and otherwise have type **char** * and point to a character string containing the capability. The special character codes involving the \ and ^ characters (such as **r** for return, or ^**A** for control A) are translated into the appropriate ASCII characters. Padding information (of the form $<time>) and parameter information (beginning with %) are left uninterpreted at this stage. The routine **tputs** interprets padding information, and **tparm** interprets parameter information.

If the program only needs to handle one terminal, the definition -DSINGLE can be passed to the C compiler, resulting in static references to capabilities instead of dynamic references. This can result in smaller code, but prevents use of more than one terminal at a time. Few programs use more than one terminal, so almost all programs can use this flag.

setupterm(term, filenum, errret)
This routine is called to initialize a terminal. **Term** is the character string representing the name of the terminal being used. **filenum** is the UTek file descriptor of the terminal being used for output. **errret** is a pointer to an integer, in which a success or failure indication is returned. The values returned can be 1 (all is well), 0 (no such terminal), or -1 (some problem locating the **terminfo** database).

The value of **term** can be given as 0, which causes the value of TERM in the environment to be used. The **errret** pointer can also be given as 0, meaning no error code is wanted. If **errret** is defaulted, and something goes wrong, **setupterm** prints an appropriate error message and exits, rather than returning. Thus, a simple program can call **setupterm (0, 1, 0)** and not worry about initialization errors.

If the variable TERMINFO is set in the environment to a pathname, **setupterm** checks for a compiled **terminfo** description of the terminal under that path, before checking */etc/term*. Otherwise, only */etc/term* is checked.

Setupterm checks the tty driver mode bits, using **filenum**, and change any that might prevent the correct operation of other low-level routines. Currently, the mode that expands tabs into spaces is disabled, because the tab character is sometimes used for different functions by different terminals. (Some terminals use it to move right one space. Others use it to address the cursor to row or column nine.) If the system is expanding tabs, **setupterm** will remove the definition of the **tab** and **backtab** functions, making the assumption that since the user is not using hardware tabs, he/she may not be properly set in the terminal. Other system-dependent changes, such as disabling a virtual terminal driver, may be made here.

As a side effect, **setupterm** initializes the global variable **ttytype**, which is an array of characters, to the value of the list of names for the terminal. This list comes from the beginning of the **terminfo** description.

After the call to **setupterm**, the global variable **cur_term** is set to point to the current structure of terminal capabilities. By calling **setupterm** for each terminal, and saving and restoring **cur_term**, a program can use two or more terminals at once.

The mode that turns newlines into CRLF on output is not disabled. Programs that use **cursor_down** or **scroll_forward** should avoid these capabilities if their value is linefeed unless they disable this mode. **Setupterm** calls **reset_prog_mode** after any changes it makes.

reset_prog_mode( )
reset_shell_mode( )
def_prog_mode( )
def_shell_mode( )

These routines can be used to change the tty modes between the two states: *shell* (the mode they were in before the program was started) and *program* (the mode needed by the program). **def_prog_mode** saves the current terminal mode as program mode. **setupterm** and **initscr** call **def_shell_mode** automatically. **reset_prog_mode** puts the terminal into program mode, and **reset_shell_mode** puts the terminal into normal mode.

A typical calling sequence is for a program to call **initscr** (or **setupterm** in a **terminfo** level program), then to set the desired program mode by calling routines such as **cbreak** and **noecho**, then to call **def_prog_mode** to save the current state. Before a shell escape or <CTRL–Z> suspension, the program should call **reset_shell_mode**, to restore normal mode for the shell. Then, when the program resumes, it should call **reset_shell_mode** before they exit. (The higher level routine **endwin** automatically calls **reset_shell_mode**.)

Normal mode is sorted in **cur_term_>Ottyb**, and program mode is in **cur_term_>Nttyb**. These sturctures are both of type **SGTTYB** (which varies depending on the system). Currently the possible types are **struct sgttyb** (on some other systems) and **struct termio** (on this version of the UTek system). **def_prog_mode** should be called to save the current state in **Nttyb**.

vidputs(newmode, putc)
newmode is any combination of attributes, defined in <curses.h>. **Putc** is a putchar–like function. The proper string to put the terminal in the given video mode is output. The previous mode is remembered by this routine. The resulting characters are passed through **putc**.
vidattr(newmode)
The proper string to put the terminal in the given video mode is output to **stdout**.

tparm(instring, p1, p2, p3, p4, p5, p6, p7, p8, p9)
Tparm is used to instantiate a parameterized string. The character string returned has the given parameters supplied, and is suitable for **tputs**. Up to nine parameters can be passed, in addition to the parameterized string.

tputs(cp, affcnt, outc)
A string capability, possibly containing padding information, is processed. Enough padding characters to delay for the specified time replace the padding specification, and the resulting string is passed, one character at a time, to the routine **outc**, which should expect one character parameter. (This routine often just calls **putchar**.) **Cp** is the capability string. **affcnt** is the number of units affected by the capability, which varies with the particular capability. (For example, the **affcnt** for **insert_line** is the number of lines below the inserted line on the screen, that is, the number of lines that will have to be moved by the terminal.) **Affcnt** is used by the padding information of some terminals as a multiplication factor. If the capability does not have a factor, the value 1 should be passed.

putp(str)
This convenient function outputs a capability with no **affcnt**. The string is output to **putchar** with an **affcnt** of 1. It can be used in simple applications that do not need to process the output of **tputs**.

delay_output(ms)
A delay is inserted into the output stream for the given number of milliseconds. The current implementation inserts sufficient pad characters for the delay. This should not be used in place of a high resolution sleep, but rather for delay effects in the output. Due to buffering in the system, it is unlikely that this call results in the process actually sleeping. Since large numbers of pad characters can be output, *ms* should not exceed 500.

Operational Details

These paragraphs describe many of the details of how the **curses** and **terminfo** packages operate.

Insert and Delete Line and Character

The algorithm used by **curses** takes into account insert and delete line and character functions, if available, in the terminal. Calling the routine

idlok(stdscr, TRUE);

enables insert/delete line. By default, **curses** does not use insert/delete line. This was not done for performance reasons, since there is no speed penalty involved. Rather, experience has shown that some programs do not need this facility, and that if **curses** uses insert/delete line, the result on the screen can be visually annoying. Since many simple programs using **curses** do not need this, the default is to avoid insert/delete line. Insert/delete character is always considered.

Additional Terminals

Curses works even if absolute cursor addressing is not possible, as long as the cursor can be moved from any location to any other location. It considers local motions, parameterized motions, home, and carriage return.

Curses is aimed at full duplex, alphanumeric, video terminals. No attempt is made to handle half-duplex, synchronous, hard copy, or bit-mapped terminals. Bit-mapped terminals can be handled by programming the bit-mapped terminal to emulate an ordinary alphanumeric terminal. This does not take advantage of the bit map capabilities, but it is the fundamental nature of **curses** to deal with alphanumeric terminals.

The **curses** handles terminals with the "magic cookie glitch" in their video attributes. The term "magic cookie" means that a change in video attributes is implemented by storing a "magic cookie" in a location on the screen. This "cookie" takes up a space, preventing an exact implementation of what the programmer wanted. **Curses** takes the extra space into account, and moves part of the line to the right, as necessary. In some cases, this will results in losing text from the right edge of the screen. Advantage is taken of existing spaces.

Multiple Terminals

Some applications need to display text on more than one terminal, but control the text by the same process. Even if the terminals are of different types, **curses** can handle this.

All information about the current terminal is kept in the following global variable:

```
struct screen *SP;
```

Although the screen structure is hidden from the user, the C compiler accepts declarations of variables that are pointers. The user program should declare one screen pointer variable for each terminal it wishes to handle. The routine

```
struct screen *
newterm(type, fd)
```

sets up a new terminal of the given terminal type that does output on file descriptor *fd*. A call to **initscr** is essentially **newterm(getenv("TERM"),stdout)**. A program wishing to use more than one terminal should use **newterm** for each terminal and save the value returned as a reference to that terminal.

To switch to a different terminal, call the following:

```
set_term(term)
```

The old value of **SP** now is returned. The programmer should not assign directly to **SP** because certain other global variables must also be changed.

All **curses** routines always affect the current terminal. To handle several terminals, switch to each one in turn with **set_term**, and closed down with **endwin**.

Video Attributes

Video attributes can be displayed in any combination on terminals with this capability. They are treated as an extension of the standout capability, which is still present.

Each character position on the screen has 16 bits of information associated with it. Seven of these bits are the characters to be displayed, leaving separate bits for nine video attributes. These bits are used for the following:

standout	**blank**
underline	**blink**
dim	**reverse video**
bold	**protect**
alternate	

Standout is taken to be whatever highlighting works best on the terminal, and should be used by any program that does not need specific or combined attributes. *Underlining, reverse video, blink, dim*, and *bold* are the usual video attributes. *Blank* means that the character is displayed as a space, for security reasons. *Protected* and *alternate* character sets depend on the particular terminal. The use of these last three bits is subject to change and is not recommended. Note also that not all terminals implement all attributes— in particular, no current terminal implements both dim and bold.

The routines to use these attributes include the following:

attrset(attrs)	**wattrset(win, attrs)**
attron(attrs)	**wattron(win, attrs)**
attroff(attrs)	**wattroff(uwin, attrs)**
standout()	**wstandout(win)**
standend()	**wstandend(win)**

Attributes, if given, can be any combination of A_STANDOUT, A_UNDERLINE, A_REVERSE, A_BLINK, A_DIM, A_BOLD, A_INVIS, A_PROTECT, and A_ALTCHARSET. These constants, defined in <curses.h>, can be combined with the C ¦ (or) operator to get multiple attributes. **Attrset** sets the current attributes to the given **attrs**; **attron** turns on the given **attrs** in addition to any attributes that are already on; **attroff** turns off the given attributes, without affecting any others. **standout** and **standend** are equivalent to **attron(A_STANDOUT)** and **attrset(A_NORMAL)**.

If the particular terminal does not have the particular attribute or combination requested, **curses** will attempt to use some other attribute in its place. If the terminal has no highlighting at all, all attributes are ignored.

Special Keys

Many terminals have special keys, such as arrow keys, keys to erase the screen, insert or delete text, and keys intended for user functions. The particular sequences these terminals send differ from terminal to terminal. **Curses** allows the programmer to handle these keys.

A program using special keys should turn on the keypad by calling the following at initialization:

keypad(stdscr, TRUE)

This causes special characters to be passed through to the program by the function **getch**. These keys have constants that are listed in the topic *Input* earlier in this section. They have values starting at 0401, so they should not be stored in a **char** variable, as significant bits will be lost.

A program using special keys should avoid using the <ESC>, since most sequences start with escape, creating an ambiguity. **Curses** will set a one second alarm to deal with this ambiguity, which will cause delayed response to the <ESC>. You should avoid <ESC> in any case, since there is eventually pressure for nearly *any* screen-oriented program to accept arrow key input.

Scrolling Region

Normally, the programmer-accessible scrolling region is set to the entire window, but the calls

setscrreg(top, bot)
wsetscrreg(win, top, bot)

set the scrolling region for **stdscr** or the given window to any combination of top and bottom margins. When scrolling past the bottom margin of the scrolling region, the lines in the region will move up one line, destroying the top line of the region. If scrolling has been enabled with **scrollok**, scrolling takes place only within that window. Note that the scrolling region is a software feature, and only causes a window data structure to scroll. This may or may not translate to use of the hardware scrolling region feature of a terminal, or insert/delete line.

Minicurses

Curses copies from the current window to an internal screen image for every call to **refresh**. If you are only interested in screen output optimization, and do not want the windowing or input functions, an interface to the lower level routines is available. This makes the program somewhat smaller and faster. The interface is a subset of full **curses**, so that conversion between the levels is not necessary to switch from **minicurses** to full **curses**.

The following functions of **curses** and **terminfo** are available to you with **minicurses**:

addch(ch))
attrset(at)
move(y,x)
refresh( )
addstc(str)
clear( )
mvaddch(y,x,ch)
standend( )
attroff(at)
erase( )
mvaddstr(y,x,str)
standout( )
attron(at)
initscr
newterm

The following functions of **curses** and **terminfo** are *not* available to you with **minicurses**:

box
deleteln
inch
makenew
mvgetstr
mvscanw
mvwdelch
mvwaddch
mvwin
mvwinsch
mvwscanw
overlay
overwrite
putp
scroll
subwin
waddch
wclear
wclrtobot
wdelch
werase
wgetstr
wmove
wrefresh
wscanw
clrtobot
delwin
insch
mvdelch
delch
getstr
longname
mvprintw
mvinch
mvwaddstr
mvwgetch
mvwgetstr
mvwinch
mvinsch
newwin
mvwprintw
printw
scanw
setscrreg
touchwin
waddstr
vidattr
wclrtoeol
wdeleteln
wgetch
winsch
wprintw
winsertln
wsetsrreg
clrtoeol
getch
insertln
mvgetch

The subset mainly requires the programmer to avoid use of more than the one window **stdscr**. Thus, all functions beginning with *w* are generally undefined. Certain high level functions that are convenient but not essential are also not available, including **printw** and **scanw**. Also, the input routine **getch** cannot be used with **minicurses**. Features implemented at a low-level, such as use of hardware insert/delete line and video attributes, are available in both versions. Also, mode setting routines such as **crmode** and **noecho** are allowed.

To access **minicurses**, add -DMINICURSES to the CFLAGS in the makefile. If routines are requested that are not in the subset, the loader prints error messages such as:

Undefined:
m_getch
m_waddch

to tell you that the routines **getch** and **waddch** were used but are not available in the subset. Since the preprocessor is involved in the implementation of mini-curses, the entire program must be recompiled when changing from one version to the other.

TTY Mode Functions

In addition to the save/restore routines **savetty( )** and **resetty( )**, standard routines are available for going into and out of normal tty mode. These routines are **resetterm( )**, which puts the terminal back in the mode it was in when **curses** was started; **fixterm( )**, which undoes the effects of **resetterm**, that is, restores the "current **curses** mode"; and **saveterm( )**, which saves the current state to be used by **fixterm( )**. **Endwin** automatically calls **resetterm,** and the routine to handle <CTRL-Z> (on other systems that have process control) also uses **resetterm** and **fixterm**. You should use these routines before and after shell escapes, and also if they write their own routine to handle <CTRL-Z>. These routines are also available at the **terminfo** level.

Type-ahead Check

If you type something during an update, the update will stop, pending a future update. This is useful when you hit several keys, each of which causes a good deal of output. For example, in a screen editor, if you press the "forward screen" key, which draws the next screen full of text, several times rapidly, rather than drawing several screens of text, the updates are cut short, and only the last screen full is actually displayed. This feature is automatic and cannot be disabled. The feature only works on versions of the UNIX system with the necessary support in the operating system.

getstr

No matter what the setting of *echo* is, strings entered in here are echoed at the current cursor location. The user's erase and kill characters are understood and handled. This makes it unnecessary for an interactive program to deal with erase, kill, and echoing when the user is entering a line of text.

longname

The **longname** function does not need any arguments. It returns a pointer to a static area containing the actual long name of the terminal.

Nodelay Mode

The following call puts the terminal in *nodelay mode*:

nodelay(atdscr, TRUE)

While in this mode, any call to **getch** returns –1 if there is nothing waiting to be read immediately. This is useful for writing programs requiring "real time" behavior where you watch action on the screen and press a key when you want something to happen. For example, the cursor can be moving across the screen in real–time. When it reaches a certain point, you can press an arrow key to change direction at that point.

Portability

Several useful routines are provided to improve portability. The implementation of these routines is different from system to system, and the differences can be isolated from the user program by including them in **curses**.

Functions **erasechar( )** and **killchar( )** return the characters that erase one character, and kill the entire input line, respectively. The function **baudrate( )** will return the current baud rate, as an integer. (For example, at 9600 baud, the integer 9600 is returned, not the value B9600 from <sgtty.h>.) The routine **fluchinp( )** causes all type–ahead to be thrown away.

5D

Curses Examples

The following examples are provided to demonstrate uses of **curses**. They are for illustration purposes only. A good programmer would expand the programs presented here before using them.

Example Program — editor

```
/*
* editor:  A screen-oriented editor.  The user
* interface is similar to a subset of vi.
* The buffer is kept in stdscr itself to simplify
* the program.
*/

#include <curses.h>

#define CTRL(c) (´c´&037)

main(argc,argv)
char**argv;
{
        int i, n, l;
        int c;
        FILE *fd;

        if(argc !=2) {
                fprintf(stderr,"Usage:edit file0);
                exit(1);
        }

        fd=fopen(argv[1],"r");
        if(fd == NULL) {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr,TRUE);
        keypad(stdscr,TRUE);
```

```
        /* Read in the file */
        while ((c=get(fd))!=EOF)
                addch(c);
        fclose(fd);

        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1],"w");
        for (l=0;l<23;l++) {
                n = len(l);
                for (i=0;i<n;i++)
                        putc(mvinch(l,i),fd);
                putc('0,fd);
        }
        fclose(fd);

        endwin();
        exit(0);
}

len(lineno)
int lineno;
{
        int linelen = COLS-1;

        while (linelen >=0 &&mvinch(lineno, linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
        int c;
        for (;;) {
                move(row, col);
                refresh ();
                c = getch();
                switch (c) {  /* Editor commands */
```

```
/* hjkl and arrow keys: move cursor */
/* in direction indicated */
case 'h':
case KEY_LEFT:
        if (col > 0)
                col--;
        break;

case 'j':
case KEY_DOWN:
        if (row < LINES-1)
                row++;
        break;

case 'k':
case KEY_UP:
        if (row > 0)
                row--;
break;

case 'l':
case KEY_RIGHT:
        if(col < COLS-1)
                col++;
        break;

/* i: enter input mode */
case KEY_IC:
case 'i':
        input();
        break;

/* x: delete current character */
case KEY_DC:
case 'x':
        delch();
        break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
        move(++row, col=0);
        insertln();;
        input();
        break;
```

```
/* d: delete current line */
                case KEY_DL:
                case 'd':
                        deleteln();
                        break;

                /* <CTRL-L>: redraw screen */
                case KEY_CLEAR:
                case CTRL(L);
                        clearok(curscr);
                        refresh ();
                        break;

                /* w: write and quit */
                case 'w':
                        return;

                /* q: quit without writing */
                case 'q':
                        endwin   ();
                        exit(1);
                default:
                        flash();
                        break;
                }
        }
}

/*
* Insert mode: accept characters and insert them.
* End with <CTRL-D> or EIC
*/
input()
{
        int c;
standout();
mvaddstr(LINES-1, COLS-20, "INPUT MODE");
standend();
move(row, col);
refresh();
for (;;) {
        c = getch();
        if (c == CTRL(D) || c == KEY_EIC)
                break;
        insch(c);
        move(row, ++col);
        refresh();
```

```
}
move(LINES-1, COLS-20);
clrtoeol();
move(row, col);
refresh();
```

Example Program — highlight

```
/*
* highlight: a program to turn U, B, and
* N sequences into highlighted
* output, allowing works to be
* displayed underlined or in bold.
*/
#include <curses.h>
main(argc, argv)
char **argv;
{
        FILE *fd;
        int c, c2;

        if (argc != 2) {
                fprintf(stderr, "Usage: highlight file0);
                exit(1);
        }

        fd = fopen(argv[1], "r");
        if (fd == NULL) {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        scrollok(stdscr,TRUE);

        for (;;) {
                c = getc(fd);
                if (c == EOF)
                        break;
                if (c == '\') {
                        c2=getc(fd);
                        switch (c2) {
                        case 'B':
                                attrset(A_BOLD);
                                continue;
                        case 'U':
                                attrset(A_UNDERLINE);
                                continue;
                        case 'N':
                                attrset(0);
                                continue;
                        }
```

```
                        addch(c);
                        addch(c2);
                }
                else
                        addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

Example Program — scatter

```
/*
*  SCATTER.  This program takes the first
*  23 lines from the standard
*  input and displays them on the
*  VDU screen, in a random manner.
*/

#include <curses.h>

#define MAXLINES 120
#define MAXCOLS 160
char s[MAXLINES][MAXCOLS];/* Screen Array */

main ()
{
        register int row=0,col=0;
        register char c;
        int char_count=0;
        long t;
        char buf[BUFSIZ];

        initscr();
        for(row=0;row<MAXLINES;row++)
                for(col=0;col<MAXCOLS;col++)
                        s[row][col]=' ';

        row = 0;
        /* Read screen in */
        while((c=getchar()) !=EOF && row < LINES ) {
                if(c != '0') {
                        /* Place char in screen array */
                        s[row][col++] = c;
                        if(c != ' ')
                                char_count++;
                } else {
                        col=0;
                        row++;
                }
        }
```

```
time(&t);/* Seed the random number generator */
srand((int) (t&0177777L));
while(char_count) {
        row=rand() % LINES;
                col=(rand()>>2) % COLS;
                if(s[row][col] != ' ')
                {
                        move(row, col);
                        addch(s[row][col]);
                        s[row][col]=EOF;
                        char_count--;
                        refresh();
                }
        }
        endwin();
        exit(0);
}
```

Example Program — show

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

main (argc, argv)
int argc;
char *argv[];
{
        FILE *fd;
        char linebuf[BUFSIZ];
        int line;
        void done(), perror(), exit();

        if(argc != 2)
        {
                fprintf(stderr,"usage: % s file0, argv[0]);
                exit(1);
        }
        if((fd=fopen(argv[1],"r")) == NULL)
        {
                perror(argv[1]);
                exit(2);
        }
        signal(SIGINT, done);

        initscr();
        noecho();
        cbreak();
        nonl();
        idlok(stdscr, TRUE);

        while(1)
        {
                move(0,0);
                for(line=0; line<LINES; line++)
                {
                        if(fgets(linebuf, sizeof linebuf, fd)
                        == NULL)
                        {
                                clrtobot();
                                done();
                        }
                        move(line, 0);
                        printw(" %s", linebuf);
                        }
```

```
                        refresh();
                if(getch() == 'q')
                        done();
        }
}


void
done()
{
        move(LINES-1,0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

Example Program — termhl

```
/*
* A terminfo level version of highlight.
*/
#include <curses.h>
#include <term.h>

int ulmode = 0;            /* Currently underlining */

main(argc, argv)
char **argv;
{
        FILE *fd;
        int c, c2;
        int outch();

        if (argc > 2) {
                fprintf(stderr,"Usage: termhl [file]0);
                exit(1);
        }
        if (argc == 2) {
                fd = fopen(argv[1],"r");
                if (fd == NULL) {
                        perror(argv[1]);
                        exit(2);
                }
        } else {
                fd = stdin;
        }

        setupterm(0,1,0);

        for (;;) {
                c = getc(fd);
                if (c == EOF)
                        break;
                if (c == '\') {
                        c2 = getc(fd);
                        switch (c2) {
                        case 'B':
                                tputs(enter_bold_mode, 1, outch);
                                continue;
                        case 'U':
                                tputs(enter_underline_mode, 1, outch);
                                ulmode = 1;
                               continue;
```

```
                                case 'N':
                                        tputs(exit_attribute_mode, 1, outch)
                                        ulmode = 0;
                                        continue;
                                }
                                putch(c);
                                putch(c2);
                        }
                        else
                                putch(c);
                }
                fclose(fd);
                fflush(stdout);
                resetterm();
                exit(0);
}

/*
* This function is like putchar, but it checks for underlining.
*/
putch(c)
int c;
{
        outch(c);
        if (ulmode && underline_char) {
                outch('
)
        tputs(underline_char, 1, outch);
;
        }
}

/*
* Outchar is a function version  of putchar that
* can be passed to
* tputs as a routine call.
*/
outch(c)
int c;
{
        putchar(c);
}
```

Example Program — two

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

struct screen *me, *you;
struct screen *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
char **argv;
{
        int done();
        int c;

        if (argc != 4) {
                fprintf(stderr,"Usage: two othertty
                otherttytype inputfile0);
                exit(1);
        }

        fd = fopen(argv[3],"r");
        fdyou = fopen(argv[1],"w+");
        signal(SIGINT, done); /* die gracefully */

        me = newterm(getenv("TERM" ), stdout);
                /* initialize my tty */
        you = newterm(argv[2], fdyou);
                /* initialize his/her terminal */

        set_term(me);     /* Set modes for my terminal */
        noecho();         /* turn off tty echo */
        cbreak();         /* enter cbreak mode */
        nonl();  /* Allow linefeed */
        nodelay(stdscr,TRUE);      /* No hang on input */

        set_term(you);    /* Set modes for other terminal */
        noecho();
        cbreak();
        nonl();
        nodelay(stdscr,TRUE);

        /* Dump first screen full on my terminal */
        dump_page(me);
```

```
        /* Dump second screen full on his/her terminal */
        dump_page(you);

        for (;;) {                  /* for each screen full */
                set_term(me);
                c= getch();
                if (c == 'q')       /* wait for user to read it */
                        done ();
                if (c == ' ')
                        dump_page(me);

                set_term(you);
                c = getch();
                if (c == 'q')       /* wait for user to read it */
                        done();
                if (c == ' ')
                        dump_page(you);
                sleep(1);
        }
}

dump_page(term)
struct screen *term;
{
        int line;

        set_term(term);
        move(0, 0);
        for (line=0; line<LINES-1; line++) {
                if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
                        clrtobot();
                        done();
                }
                mvprintw(line, 0, "%s", linebuf);
        }
        standout();
        mvprintw(LINES-1, 0, "--More--");
        standend();
        refresh();                  /* sync screen */
}
```

```
/*
* Clean up and exit.
/*
done()
{

        /* Clean up first terminal */
        set_term(you);
        move(LINES-1,0);  /* to lower left corner */
        clrtoeol();               /* clear bottom line */
        refresh();                /* flush out everything */
        endwin();                 /* curses cleanup */

        /* Clean up second terminal */
        set_term(me);
        move(LINES-1,0);  /* to lower left corner */
        clrtoeol();               /* clear bottom line */
        refresh();                /* flush out everything */
        endwin();                 /* curses cleanup */

        exit(0);
}
```

Example Program — window

```
#include <curses.h>

WINDOW *cmdwin;

main ()
{
        int i, c;
        char buf[120];

        initscr();
        nonl();
        noecho();
        cbreak();

        cmdwin = newwin(3,COLS,0,0);        /* top 3 lines */
        for (i=0; i<LINES; i++)
                mvprintw(i,0,"This is line %d of stdscr", i);

        for(;;) {
                refresh();
                c = getch();
                switch (c) {
                case 'c':  /* Enter command from keyboard */
                        werase(cmdwin);
                        wprintw(cmdwin, "Enter command:");
                        wmove(cmdwin,2,0);
                        for (i=0; i<COLS; i++)
                                waddch(cmdwin, '-');
                        wmove(cmdwin);
                        touchwin(cmdwin,1,0);
                        wrefresh(cmdwin);
                        wgetstr(cmdwin, buf);
                        touchwin(stdscr);
                        /*
                        * The command is now in buf.
                        * It should be processed here.
                        */
                        break;
                case 'q':
                        endwin( );
                        exit(0);
                }
        }
}
```

The Fortran 77 Compiler

Introduction

The UTek operating system provides the Fortran 77 compiler. Fortran 77 is the official standard for the Fortran programming language, replacing Fortran 66. Fortran 77 includes many of the features of Fortran 66. This section describes the compiled language, interfaces between procedures, and the file formats assumed by the I/O system.

Usage

The command to run the Fortran compiler is:

f77 *[options] file*

The **f77** command is a general–purpose command for compiling and loading Fortran and Fortran–related files. EFL and Ratfor sources files are preprocessed before they are presented to the Fortran compiler. C and assembler source files are compiled by the appropriate programs. The compiler also loads object files. The Fortran 77 compiler recognizes the following filename suffixes:

.f	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C source file
.s	Assembler source file
.o	Object file
.F	Fortran source file, processed by C, EFL or Ratfor first

The Fortran 77 compiler accepts the following options:

-S	Generate assembler output for each source file, but do not assemble it. Assembler output for a source file *x.f, x.e, x.r,* or *x.c* is put in file *x.s*, where *x* is a filename.
-c	Compile but do not load. Output for *x.f, x.e, x.r, x.c,* or *x.s* is put in file *x.o*.
-m	Apply the m4 macro pre-processor to each EFL or Ratfor source file before using the appropriate compiler.
-f	Apply the EFL or Ratfor processor to all relevant files, and leave the output from *x.e* or *x.r* in *x.f*, where *x* is a filename. Do not compile the resulting Fortran program.
-p	Generate code to produce usage profiles for use by the **prof** command.
-gp	Generate code to produce usage profiles for use by the **gprof** command.
-o *f*	Put executable code in file *f*. Default filename is a.out.
-w	Suppresses all warning messages.
-w66	Suppress warnings about Fortran 66 features.
-O	Invoke the C object code optimizer.
-C	Compile code that checks that subscripts are within array bounds.
-onetrip	Compile code that performs every **do** loop at least once.
-U	Do not convert uppercase letters to lowercase. The default converts Fortran programs to lowercase.
-u	Make the default type of a variable *undefined*.
-I2	On machines that support short integers, make the default integer constants and variables short. (**—I4** is the standard value.) All logical quantities are short.
-E*x*	Use the string *x* as an EFL option in processing *.e* files.
-R*x*	Use the string *x* as a Ratfor option in processing Ratfor files.
-F	Ratfor and EFL source programs are preprocessed into Fortran files, but those files are not compiled or removed.

Other options, all library names, and any names not ending with one of the understood suffixes, are passed to the loader.

Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. This approach guarantees that the resulting programs are compatible with C usage. The run–time computational library is complete. The mathematical functions are computed to at least 63–bit precision. The run–time I/O library makes use of the standard I/O package, written in C, to transfer data. With the few exceptions described below, only documented calls are used, so it should be relatively easy to modify programs to run on other operating systems.

Language Extensions

Fortran 77 includes almost all of Fortran 66 as a subset. The most important additions to Fortran 77 are a character string data type, file–oriented input/output statements, and random access I/O. The language is also considerably streamlined.

Double Complex Data Type

The Fortran 77 compiler defines the new type *double complex*. Each piece of data is represented by a pair of double–precision real variables. The compiler provides a double–complex version of every *complex* built–in function. The specific function names begin with **z** instead of **c**.

Internal Files

Fortran 77 has "internal files" (memory arrays), but restricts their use to formatted sequential I/O statements. The I/O system lets you use internal files in direct and unformatted reads and writes.

Implicit Undefined Statement

Fortran 66 has a fixed rule that the type of a variable not appearing in a type statement is *integer*, if its first letter is **i, j, k, l, m**, or **n**, and *real* otherwise. Fortran 77 has an **implicit** statement that overrides this rule. An additional type, *undefined*, is permitted. Consider the following statement:

implicit *undefined(a-z)*

This statement turns off the automatic data–typing mechanism, and the compiler issues a diagnostic for each variable that is used, but does not appear in a type statement. Specifying the **–U** option to the **f77** command is equivalent to beginning each procedure with this **implicit** *undefined* statement.

Recursion

Procedures can call themselves, directly or through a chain of other procedures.

Automatic Storage

The Fortran 77 compiler recognizes two new keywords, **static** and **automatic**. These keywords can appear as types in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the data, and its value is retained between calls. Each invocation of the procedure has one copy of each variable declared **automatic**. Automatic variables cannot appear in **equivalence, data**, or **save** statements.

Source Input Format

To make it easier to type Fortran programs, the Fortran 77 compiler accepts input in variable–length lines. An ampersand character as the first character of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

The Fortran 77 compiler expects lowercase input. By default, the compiler converts all uppercase characters to lowercase except those inside character constants. However, if you specify the **–U** option, uppercase letters are not transformed. Using this option, you can specify external names with uppercase letters in them, and make distinct variables that differ only in case. Whether or not you use the **–U** option, keywords are recognized in lowercase.

Include Statement

Consider the statement:

include *'stuff'*

This statement is replaced by the contents of the file *stuff*. You can nest **include** statements to a depth of ten.

Binary Initialization Constants

A *logical, real,* or *integer* variable can be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**. the string is octal. with digits **0–7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0–9, a–f.** Thus, the following statements initialize all three elements of **a** to ten:

```
integer a(3)
data a/b'1010',o'12',z'a'/
```

Character Strings

For compatibility with C usage, the compiler recognizes the following escape sequences:

\n	newline
\t	tab
\b	backspace
\f	form feed
\0	null
\'	apostrophe (does not terminate a string)
\"	quotation mark (does not terminate a string)
\\	backslash
x	*x*, where *x* is any other character

Fortran 77 has only one quoting character, the apostrophe. The Fortran 77 compiler and I/O system recognize both the apostrophe and the double quote as quoting characters. If a string begins with one kind of quote mark, the other can be embedded within the string without using the repeated quote or escape sequences.

Every scalar local character variable that is not equivalenced, and every character string constant, are aligned on an *integer* word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to make communication with C routines easier.

Hollerith Notation

In the Fortran 77 compiler, Hollerith data can be used in place of character string constants, and can also be used to initialize non–character variables in **data** statements, except for real variables.

Equivalence Statements

As a special case, Fortran 66 permits an element of a multi–dimensional array to be represented by a singly–subscripted reference in **equivalence** statements. Fortran 77 does not permit this usage, since subscript lower bounds can now be different from 1. The Fortran 77 compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message displays for each missing subscript.

One–trip Do Loops

The Fortran 77 standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value. For example:

```
do 10 i = 2,1
```

In Fortran 66 such a statement is undefined, but it was common practice to perform the range of a **do** loop at least once. In order to accommodate older programs, the **–onetrip** option to the Fortran 77 compiler generates non–standard loops.

Commas in Formatted Input

The I/O system attempts to be more lenient than the Fortran 77 Standard when it seems worthwhile. While doing a formatted read of non–character variables, you can use commas as value separators in the input record, overriding the field lengths given in the format statement. Thus this format reads the record **–345,.05e–3,12** correctly:

```
(i10, f20.10, i4)
```

Short Integers

On machines that support half–word integers, the compiler accepts declarations of type *integer*2*. (Ordinary integers are of C type *long init;* half–word integers are of C type *short init*.) An expression involving only objects of type *integer*2* is of type *short*. Generic functions return short or long integers depending on the actual types of their arguments. If you compile a procedure using the **–I2** option, all small integer constants are of type *integer*2*. If the precision of an integer–valued intrinsic function is not determined by the generic function rules, a rule is chosen that returns the prevailing length. When the **–I2** option is in effect, all quantities of type *logical* are short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

Additional Intrinsic Functions

The Fortran 77 compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or, and xor,** and **not**) and for accessing the UTek command arguments (**getarg** and **iargc**).

Violations of the Fortran 77 Standard

There are three ways in which this Fortran implementation violates the Fortran 77 Standard: double precision alignment, dummy procedure arguments, and the tab format codes. Each of these violations is discussed in the following sections.

Double Precision Alignment

The Fortran Standard permits **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary. For example:

```
real a(4)
double precision b,c

equivalence (a(1),b), (a(4),c)
```

All double precision real and complex quantities must be aligned on word boundaries. The system displays a diagnostic if the source code demands a violation of the rule.

Dummy Procedure Arguments

If any argument of a procedure is of type *character*. all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement is a corollary of the way character string arguments are represented, and of the one–pass nature of the compiler. A warning displays if a dummy procedure is not declared **external**.

t and tl Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes let you reread or rewrite part of the record that has already been processed. This implementation uses seeks, so if the terminal where you are running the program does not allow seeks, the program is in error. A benefit of this implementation of tab format codes is that there is no upper limit on the length of a record. Also, you do not have to pre–declare any record lengths, except where specifically required.

Inter–procedure Interface

To write C procedures that call or are called by Fortran procedures, you must know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

Procedure Names

On the UTek system, the name of a common block or a Fortran procedure has an underscore appended to it. This distinguishes it from a C procedure, or an external variable with the same user–assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user–assigned subroutine names.

Data Representations

The following is a table of corresponding Fortran and C declarations:

Fortran	C
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character *6 x	char x[6];

Return Values

A function of type *integer, logical, real,* or *double precision* is declared as a C function that returns the corresponding type. A *complex* or *double complex* function is equivalent to a C routine with an additional initial argument that points to the place where the return value is stored. Consider this example:

```
complex function f(...)
```

It is equivalent to:

```
f_(temp,...)
struct { float r, i; } *temp;
...
```

A character–valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Consider this example:

```
character*15 function g(...)
```

It is equivalent to:

```
g_(result, length,...)
char result[];
long int length;
...
```

This function is invoked in C by:

```
char chars[15];
...
g_(chars, 15L,...);
```

Subroutines are invoked as if they are integer–valued functions whose value specifies what alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the return value is undefined.) Consider the following statement:

```
call nret(*1, *2, *3)
```

It is treated exactly as if it were the computed **goto**:

```
goto (1, 2, 3), nret()
```

Argument Lists

All Fortran arguments are passed by address. In addition, for every argument of type *character* or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are *long int* quantities passed by value.) The order of the arguments is then:

- extra arguments for complex and character functions
- address for each piece of data or function
- a *long int* for each character or procedure argument

Consider this call:

```
external f
character*7 s
integer b(3)
...
call sam(f, b(2), s)
```

It is equivalent to:

```
int f();
char s[7];
long int b[3];
...
sam_(f, &b[1],s, 0L, 7L);
```

Note that the first element of a C array always has subscript 0, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column–major order, C arrays are stored in row–major order.

File Formats

Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UTek systems, these are all implemented as ordinary files that are assumed to have the proper internal structure.

Fortran I/O is based on records. When a direct file is opened in a Fortran program, the record length of the records is given, and this is used by the Fortran I/O system to make the file look as though it is made up of records of the given length. In the special case that the record length is given as 1, the files are not divided into records, but, like normal UTek files, are treated as byte–addressable byte strings.

The particular requirements on sequential unformatted files make it unlikely that they will be read or written except by Fortran I/O statements. Each record is preceded by an integer containing either the record length in bytes or the maximum buffer size. Each record is followed by an integer containing the total length in bytes necessary to write the record.

As it reads, the Fortran I/O system breaks sequential formatted files into records by using each newline as a record separator. Reading off the end of the record makes the I/O system treat the record as though it were extended by blanks. On output, the I/O system writes a newline at the end of each record.

Programs can also write newlines for themselves. This is a program error, but its only effect is that the single record you wrote is treated as more than one record when you backspace over it or read it.

Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, except for two nonstandard features: the I/O system needs to know whether a file can be used for direct I/O, and whether it is possible to backspace. Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** that determines if **fseek** will work. The **inquire** statement also lets you find out if two files are the same. It gets the name of an already open file in a form that enables the program to reopen it.

Preconnected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All units are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named *fort.n*. These files need not exist, nor are they created, unless their units are used without first executing an **open**. The default connection is for sequential formatted I/O.

The Fortran 77 Standard does not specify where a file is initially positioned when it is **open**ed for sequential I/O. In fact, the I/O system attempts to position the file at the end, so a **write** appends the file and a **read** results in an end-of-file. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

Ratfor — A Preprocessor for Rational Fortran

Introduction

Most programmers agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. Fortran is close to a universal programming language, and with care you can write large, portable Fortran programs. Fortran is often the most efficient language available, particularly for programs that require much computation.

Perhaps the worst deficiencies of Fortran are control flow statements — conditional branches and loops — that express the logic of the program. The conditional statements in Fortran are primitive. The arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO statements; it leads to unintelligible code. The logical IF is better, but very restrictive because the statement that follows the IF can be only one restricted Fortran statement.

The Fortran DO statement restricts the user to going forward in an arithmetic progression. The DO statement is useless if a problem is not an arithmetic progression.

When you are faced with a programming language that requires numerous labels and branches, a useful technique is to define a new language that overcomes the deficiencies and use a preprocessor to translate the new language into the original one. This is how **ratfor** is implemented.

Ratfor improves the control flow statements of Fortran by providing:

- statement grouping
- **if–else** and **switch** statements for decision making
- **while, for, do**, and **repeat–until** statements for looping
- **break** and **next** for controlling loop exits

Ratfor improves the syntax of Fortran by providing:

- free–form input
- unobtrusive comments
- translation of >, >= into .GT., .GE., etc.
- **return***(expression)* statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

For more detailed information on **ratfor**, including how to invoke the command, see your *UTek Command Reference, ratfor(1).*

Language Description

Ratfor retains the merits of Fortran — universality, portability, efficiency — and hides the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, **ratfor** provides decent control flow structures. They are sufficient and comfortable for programming without GOTO statements. Second, since the preprocessor examines an entire program to translate the control structure, it cleans up many of the cosmetic deficiencies of Fortran, providing code that is easier to read.

Beyond these two aspects, **ratfor** does nothing about other weaknesses of Fortran. The design principle that determined what should be in **ratfor** is: **ratfor** *doesn't know Fortran*. Any language feature requiring that **ratfor** understand Fortran was omitted. **Ratfor** provides a small set of the most useful constructs, instead of everything that might prove useful.

Statement Grouping

Fortran provides no way to group statements together, short of making them into a subroutine. For example:

```
if (x > 100)
  {call error("x>100"); err = 1; return}
```

This cannot be written directly in Fortran. Instead you must translate this relatively clear thought into Fortran, stating the negative condition and branching around the group of statements:

```
if (x .le. 100) goto 10
   call error(5hx)>100)
   err = 1
   return
10 ...
```

When this program doesn't work, you must translate it back into a clearer form before you know exactly what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in **ratfor**. A group of statements can be treated as a unit by enclosing them in braces. This is true throughout **ratfor**. Wherever you can use a single **ratfor** statement, several can be enclosed in braces.

Cosmetics contribute to the readability of code. The character > is more clear than .GT, so **ratfor** translates it appropriately.

Ratfor is a free-form language: statements can appear anywhere on a line, and several can appear on one line if they are separated by semicolons. The above example could also be written as:

```
if (x > 100) {
   call error("x>100")
   err = 1
   return
}
```

In this case, no semicolon is necessary at the end of each line because **ratfor** assumes that there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** statement is a single statement, no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
  write(6,20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general, **ratfor** continues lines when it seems obvious that they are not yet done.

Although a free-form language permits wide latitude in formatting styles, proper indentation is vital. It makes the logical structure of the program obvious to the reader.

The else Clause

Ratfor provides an **else** statement to handle the construction:

```
if (a <= b)
   { sw = 0; write (6, 1) a, b }
else
   { sw = 1; write (6, 1) b, a }
```

This writes out the smaller of *a* and *b*, then the larger, and sets *sw* appropriately.

The Fortran equivalent of this code is very circuitous:

```
      if (a .gt. b) goto 10
         sw = 0
         write(6, 1) a, b
         goto 20
10    sw = 1
      write(6, 1) b, a
20    ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem of being less understandable than code that is not a translation. To understand the Fortran version, you must scan the entire program to make sure that no other statement branches to statements 10 or 20. With the **ratfor** version, the **if–else** statement is a single unit that can be read and understood.

As before, if the statement following an **if** or **else** is a single statement, no braces are needed:

```
if (a <= b)
   sw = 0
else
   sw = 1
```

The syntax of the **if** statement is:

if (*legal Fortran condition*)
  (*Ratfor statement*)
else
  (*Ratfor statement*)

In this case, the **else** statement is optional. The *legal Fortran condition* is anything that can legally go into a Fortran logical IF. **Ratfor** does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any **ratfor** or Fortran statement, or any collection of them in braces.

Nested if Statements

Since the statement that follows an **if** or an **else** can be any **ratfor** statement, it can be followed by another **if** or **else**. As a useful example, consider this problem: the variable f is set to -1 if x is less than zero, to $+1$ if x is greater than 100, and to 0 otherwise. In **ratfor** enter:

```
if (x < 0)
   f = -1
else if (x > 100)
   f = +1
else
   f = 0
```

Here the statement after the first **else** is another **if–else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran is indirect because Fortran does not let you say what you mean.

Following an **else** statement with an **if** statement is one way to write a multiple–choice branch in **ratfor**. In general this structure provides a way to specify the choice of exactly on of several alternatives:

```
if (...)
   ...
else if
   ...
else if
   ...
...
else
   ...
```

Ratfor also provides a **switch** statement that does the same job in special cases; in more general situations, you have to make do. The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** statement handles the default case, where none of the other conditions apply. If there is no default action, this final **else** statement is omitted:

```
if ( x < 0)
   x = 0
else if (x > 100)
   x = 100
```

Ambiguity in if–else Statements

There is one thing to notice about complicated structures involving nested **if** and **else** statements. Consider this fragment:

```
if (x > 0)
   if (y > 0)
      write(6, 1) x, y
   else
      write(6, 2) y
```

In this fragment there are two **if** statements and only one **else** statement. Which **if** does the **else** go with?

This is a genuine ambiguity in **ratfor**. It is resolved by saying that in such cases the **else** goes with the previous **if** that does not have a corresponding **else** statement. In this case, the **else** goes with the inner **if**, as indicated by the indentation.

You can also resolve this ambiguity using explicit braces. In the case above, enter:

```
if ( x > 0) {
   if ( y > 0)
      write(6, 1) x, y
   else
      write(6, 2) y
}
```

This does not change the meaning, but it leaves no doubt in the reader's mind. If you want the other association write:

```
if ( x > o ) {
   if (y < 0)
      write(6, 1) x, y
}
else
   write(6, 2) y
```

The switch Statement

The **switch** statement provides a clean way to express multiple–choice branches that branch on the value of some integer–valued expression. The syntax is:

switch (*expression*) {

  case *expr1*:
    statements
  case *expr2, expr3*:
    statements
  ...
  default:
    statements
}

Each **case** is followed by a list of comma–separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1, expr2* until one matches. Then the statements following that **case** are executed. If no **cases** match *expression*, and there is a **default** section, the statements there are executed. If there is no **default**, nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately.

NOTE

This behavior is not the same as that of the C **switch** *statement.*

The do Statement

The **do** statement in **ratfor** is similar to the DO statement in Fortran, except that it uses no statement number. The statement number serves only to mark the end of the DO. You can do this easily with braces. Thus:

```
do i = 1, n {
   x(i) = 0.0
   y(i) = 0.0
   z(i) = 0.0
}
```

This is the same as the Fortran:

```
      do 10 = 1, n
         x(i) = 0.0
         y(i) = 0.0
         z(i) = 0.0
10    continue
```

The syntax is:

do *legal Fortran DO text*
  Ratfor statement

The part that follows the keyword **do** can legally go into a Fortran DO statement. So if a local version of Fortran allows DO limits to be expressions, you can use them in a **ratfor do**.

The *Ratfor statement* portion is often enclosed in braces, but like the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
  x(i) = 0.0
```

This slightly more complicated code sets the entire array *m* to 0:

```
do i = 1, n
  do j = 1, n
    m(i, j) = 0
```

This code sets the upper triangle of *m* to –1, the diagonal to 0, and the lower triangle to +1:

```
do i = 1, n
   do j = 1, n
      if (i < j)
         m(i, j) = -1
      else if (i == j)
         m(i, j) = 0
      else
         m(i, j) = +1
```

The operator = = means equals. In each case, the statement that follows the **do** is logically a *single* statement, so it doesn't need braces.

The break and next Statements

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. The **break** causes an immediate exit from the **do**; in effect it is a branch to the statement following the **do**. The **next** statement is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
   if (x(i) < 0.0)
      next
   process positive element
}
```

The **break** and **next** statements also work in **ratfor** looping constructions discussed in the next few topics.

The **break** and **next** statements can be followed by an integer to indicate breaking or iterating that level of enclosing loop. For example:

break 2

This exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. The **next** statement iterates the second enclosing loop.

The while Statement

One of the problems with the Fortran DO statement is that it insists upon being done once regardless of its limits. If a loop begins:

DO I = 2, 1

This typically is done once, with *I* set to 2. Of course a **ratfor do** can easily be preceded by a test:

```
if (j <= k)
   do i = j, k {
}
```

A more serious problem with the DO statement is that it encourages you to write a program in terms of an arithmetic progression with small positive steps, although that may not be the best way to write it.

To overcome these difficulties, **ratfor** provides a **while** statement, which is a simple loop:

while *some condition is true*
  repeat *this group of statements*

For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria:

```
real function sin(x, e)
   #returns sin(x) to accuracy e, by
   #sin(x) = x - x**3/3! + x**5/5! - ...

   sin = x
   term = x

   i = 3
   while (abs(term)>e & i<100) {
      term = -term * x**2 / float(i*(i-1))
      sin = sin + term
      i = i + 2
}

return
end
```

If the routine is entered with *term* already smaller than *e*, the loop is done zero times, and no attempt is made to compute *x**3*, avoiding a potential underflow. Since the text is made at the top of the **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test *i<100* is the other boundary — making sure the routine stops after some maximum number of iterations.)

A pound sign (#) in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line. Blank lines are also permitted anywhere. Use them to emphasize the natural divisions of a program.

The syntax of the **while** statement is:

while (*legal Fortran condition*)
  Ratfor statement

As with the **if**, a *legal Fortran condition* is something that can go into a Fortran logical IF, and *Ratfor statement* is a single statement that can also be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function value and in its argument. Then a loop to find the first non-blank character is:

```
while (nextch(ich) == iblank)
   ;
```

A semicolon by itself is a null statement that marks the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non–blank.

The for Statement

The **for** statement is another **ratfor** loop. It carries the separation of loop–body from reason–for–looping a step further than does the **while** statement. A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is:

for (i = 1; i <= n; i = i + 1) ...

This is equivalent to:

```
i = 1
while (i <= n) {
   ...
   i = i + 1
}
```

The initialization and increment of *i* are moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions are done zero times if *n* is less than 1; this is not true of the **do** statement.

The loop of the **sine** routine in the previous section can be re–written with a **for** statement:

```
for (i=3; abs(term) > e & i < 100; i=i+2) {
   term = -term * x**2 / float(i*I1-1))
   sin = sin + term
}
```

The syntax of the **for** statement is:

for (*init, condition; increment*)
  Ratfor statement

In this statement *init* is any single Fortran statement, and is executed once before the loop begins. The *increment* is any single Fortran statement, and is execute at the end of each pass through the loop, before the test. The *condition* is anything that is legal in a logical IF. You can omit any of *init, condition*, and *increment*, although you must enter each semicolon. A non–existent *condition* is treated as always true, so *for*(;;) is an indefinite repeat.

The **for** statement is useful for backward loops, chaining along lists, loops that might be done zero times, and similar algorithms that are difficult and obscure with DO, IF, and GOTO. For example, here is a backwards DO loop to find the last non–blank character on a card:

```
for (i = 80; i > 0; i = i - 1)
  if (card(i) != blank)
    break
```

The notation != is the same as .NE. The code scans the columns from 80 through 1. If it finds a non–blank, the loop immediately breaks. If *i* reaches 0, the card is all blank. This handles the termination properly for free; *i* is 0 when you fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array *ptr*) until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
  sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the text at the top of a loop instead of the bottom eliminates a potential boundary error.

The repeat–until Statement

Sometimes you need a loop that tests at the bottom after one pass through. This service is provided by the **repeat–until** statement:

repeat
  Ratfor statement
until (*legal Fortran condition*)

The *Ratfor statement* is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a **repeat** by itself is the cleanest way to specify an infinite loop. Of course, such a loop must ultimately be broken by some transfer of control such as **stop, return**, or **break**, or an implicit stop such as running out of input with a READ statement.

The **repeat–until** statement is much less used than the other looping constructions. Be cautious about using it, because loops that test only at the bottom often do not handle null cases well.

More Information on break and next Statements

The **break** statement exits immediately from **do, while, for**, and **repeat–until**. The **next** statement goes to the text part of **do, while** and **repeat–until**, to the increment step of a **for**.

The return Statement

The standard Fortran mechanism to return a value from a function uses the name of the function as a variable that can be assigned to; the last value stored in that variable is the function value upon return. For example, here is a routine that returns 1 if two arrays are identical, and 0 if they differ. The array ends are marked by the special value –1:

```
#equal -- compare str1 to str2
#return 1 if equal, 0 if not
   integer function equal(str1, str2)
   integer str1(100), str2(100)
   integer i

   for (i = 1; str1(i) == str2(i); i = i + 1)
      if (str1(i) == -1) {
         equal = 1
         return

      }
   equal = 0
   return
   end
```

In many languages, you return a value from a function like this:

return *(expression)*

Ratfor provides this kind of **return** statement — in a function *F*, **return***(expression)* is equivalent to:

{ **F** = **expression; return** }

For example, here is **equal** again:

```
#equal -- compare str1 to str2
#return 1 if equal, 0 if not
   integer function equal(str1, str2)
   integer str1(100), str2(100)
   integer i

   for (i = 1; str1(i) == str2(i); i = i + 1)
      if (str1(i) == -1)
         return(1)
   return(0)
   end
```

If there is no parenthesized expression after **return**, a normal RETURN is made.

The Appearance of a Ratfor Program

The visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, **ratfor** provides a number of facilities that make programs more readable, including free-form input and translation services.

Free-form Input

You can put statements anywhere on a line; long statements are continued automatically, as are long conditions in **if, while, for**, and **until**. Blank lines are ignored. You can put multiple statements on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of these characters are assumed to be continued on the next line:

```
= + - * , | & ( _
```

Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all–numeric field is assumed to be a Fortran label, and placed in columns 1–5 on output. Thus:

```
write(6, 100); 100 format("hello")
```

is converted into:

```
      write(6, 100)
  100 format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to the Fortran convention *nH*, but is otherwise unaltered. Within quoted strings, the backslash (\) serves as an escape character; the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"This is a string containing a backslash (\\) and an apostrophe (\'). (This is *not* the standard use of double quotes, but it is easier to use and more general.)"
```

Any line that begins with a percent sign (%) is left absolutely unaltered except for stripping off the percent sign and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be changed a lot. Use the percent sign only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output can come out in an unexpected place.

Ratfor makes the following character translations, except within single or double quotes, or on a line beginning with a percent sign:

==	**.eq.**	!=	**.ne.**
>	**.gt.**	>=	**.ge.**
<	**.lt.**	<=	**.le.**
&	**.and.**	|	**.or.**
!	**.not.**	^	**.not.**

In addition, the following translations are provided for input devices with restricted character sets:

[	{	**]**	}
$(	{	**$)**	}

The define Statement

You can define any string of alphanumeric characters as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics), it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

The **define** statement is typically used to create symbolic parameters:

```
define    ROWS 100
define    COLS 50
dimension a(ROWS), b(ROWS, COLS)
     if (i > ROWS | j > COLS) ...
```

Alternately, you can write definitions as:

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this lets you create multi-line definitions.

Is it a good idea to use symbolic parameters for most constants, because they make clear the function of otherwise mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants:

```
define   YES   1
define   NO    0
define   EOS   -1
define   ARB   100

# equal -- compare str1 to str2;
# return YES if equal, NO if not
   integer function equal(str1, str2)
   integer str1(ARB), str2(ARB)
   integer i

   for (i = 1; str1(i) == str2(i); i = i + 1)
      if (str1(i) == EOS)
         return(YES)
      return(NO)
      end
```

The include Statement

This statement inserts the file found on input stream *file* into the **ratfor** input in place of the **include** statement:

include *file*

The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
   include commonblocks
   ...
   end

subroutine y
   include commonblocks
   ...
   end
```

This ensures that all copies of the COMMON blocks are identical.

Ratfor Difficulties

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make are reported by the Fortran compiler, so you have to relate a Fortran diagnostic back to the **ratfor** source.

Keywords are reserved — using **if, else**, etc. as variable names does not work. Do not leave spaces in keywords or use the arithmetic IF.

The Fortran *nH* convention is not recognized anywhere by **ratfor**; use quotes instead.

The biggest single problem with **ratfor** is that many Fortran syntax errors are not detected by **ratfor**, but by the local Fortran compiler. The compiler prints a message in terms of the generated Fortran, and in some cases this may be difficult to relate back to the offending **ratfor** line. You can deal with this problem by tagging each generated line with some indication of the source line that created it.

Ratfor keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an arithmetic IF. A few standard Fortran constructions are not accepted by **ratfor**. You can use the percent sign to protect lines with those non–recognized Fortran constructions.

Implementation

The **ratfor** grammar is simple and straightforward, being essentially:

```
prog : stat
     ¦ prog   stat
stat : if (...) stat
     ¦ if (...) stat else stat
     ¦ while (...) stat
     ¦ for (...; ...; ...) stat
     ¦ do ... stat
     ¦ repeat stat
     ¦ repeat stat until (...)
     ¦ switch (...) { case ...: prog ...
           default: prog }
     ¦ return
     ¦ break
     ¦ next
     ¦ digits  stat
     ¦ { prog }
     ¦ anything unrecognizable
```

The observation that **ratfor** knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable." In fact, most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if, else**, etc.) the entire statement is simply copied to the output with the appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L + 1 are generated, and the value of L is stacked. The condition is then isolated, and this code is output:

if (.not. (*condition*)**) goto L**

The *statement* part of the **if** is then translated. When **ratfor** encounters the end of the statement this code is generated:

L continue

This code is generated unless there is an **else** clause, in which case the code is:

```
     goto L+1
L    continue
```

In this latter case, this code is produced after the *statement* part of the **else**:

L+1 continue

Code generation for the various loops is equally simple.

5G

Using Pascal on UTek

Introduction

This section discusses how the Pascal programming language works on the UTek operating system. Topics covered include:

- error diagnostics produced by the compiler, **pc**
- input and output
- components and options of the system

Another source of information on Pascal that you receive with your workstation is *The Pascal User Manual and Report* by Kathleen Jensen and Niklaus Wirth. This book actually contains two documents, the *User Manual* and the *Report*. The *User Manual* is a document designed to introduce you to the features of Pascal. The *Report* is a concise reference for use by experienced Pascal users. Many of the example programs in this section are taken from the *User Manual*.

Basic UTek Pascal

This section explains the basics of using UTek Pascal. The UTek Pascal system requires that programs reside in files whose named end with the suffix *.p*, so call the new file *first.p*. Create a file that contains the following program:

```
program first(output)
begin
        writeln('Hello, world!')
end.
.
```

Now you can compile the program using the **pc** command. Enter:

```
pc first.p
Tue Oct 14 21:37 1984 first.p:
        2 begin
e ——^—— Inserted ';'
```

The compiler first printed a syntax error message. The number 2 indicates that the rest of the line is an image of the second line of our program. The compiler expects to find a semicolon (;) before the keyword **begin** on this line. If you look at the Pascal syntax charts in the Jensen–Wirth *User Manual*, you see that the terminating semicolon (;) of the program statement on the first line is omitted.

Another thing to notice about the error message is the letter 'e' at the beginning. It stands for 'error', indicating that the input was not legal Pascal. The fact that it is a lower–case 'e' instead of an upper–case one indicates that the compiler managed to recover from the error to produce an executable a.out file. An executable *a.out* file is produced when no fatal 'E' errors occur during compilation. Other classes of warning messages include 'w', which indicates inconsistencies that are probably due to bugs in the program. Warning messages preceded by 's' violate standard Pascal conventions.

After compiling the program, you can run it by executing the *a.out* file. But first fix the error in the program, by inserting a semicolon at the end of the first line. Now enter **ls** to list the files in the current directory:

```
ls
first.p
a.out
```

The *a.out* file contains the executable binary code. Execute it by entering:

```
a.out
Hello, world!
```

You can rename the program something other than *a.out* using the move command, **mv**. To rename the *a.out* file to program *hello* enter:

```
mv a.out hello
```

To execute the program enter:

```
hello
Hello, world!
```

A Larger Program

This program is similar to program 4.9 on page 30 of the Jensen–Wirth *User Manual*. For the sake of example, a number of problems have been introduced into the program. If the program resides in the file *bigger.p*, you can list it with numbers by entering:

cat –n bigger.p

```
 1  (*
 2   * Graphic representation of a function
 3   *    f(x) = exp(-x) * sin(2 * pi * x)
 4   *)
 5  program graph1(output);
 6  const
 7           d = 0.0625;   (* 1/16, 16 lines for interval [x, x+1] *)
 8           s = 32;       (* 32 character width for interval [x, x+1]
 9           h = 34;       (* Character position of x-axis *)
10           c = 6.28138;  (* 2 * pi *)
11           labeled = 32;
12  var
13           x, y: real;
14           i, n: integer;
15  begin
16           for i := 0 to labeled begin
17                   x := d / i;
18                   y := exp(-x9 * sin(i * x);
19                   n := Round(s * y) + h;
20                   repeat
21                            write(' ');
22                            n := n - 1
23                   writeln('*')
24  end.
```

When you attempt to compile the program using **pc**, the following output displays:

pc bigger.p

```
Wed Aug 15 21:52 1984  bigger.p:
     9   h = 34;        (* Character position of x-axis *)
w ------------------------------^--- (* in a (* ... *) comment
    16   for i := 0 to labeled begin
e ------------------------------^--- Inserted keyword do
    18            y := exp(-x9 * sin(i * x);
E ------------------------------^--- Undefined variable
e ----------------------------------------------^--- Inserted ')'
    19            n := Round(s * y) + h;
E ---------------------------^--- Undefined function
E ----------------------------------------------^--- Undefined variable
    23            writeln('*')
e ---------------------^--- Inserted ';'
    24  end.
E -------^--- Expected keyword until
E ----------^--- Malformed declaration
  ----------^--- Unexpected end-of-file - QUIT
```

Since there were fatal E errors in the program, no executable code was generated. To list the program with its error messages enter:

pc -l bigger.p

```
Wed Aug 15 21:52 1984  bigger.p

     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(-x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7   d = 0.0625;    (* 1/16, 16 lines for interval [x, x+1] *)
     8   s = 32;        (* 32 character width for interval [x, x+1]
w ------------------------------^--- (* in a (* ... *) comment
     9   h = 34;        (* Character position of x-axis *)
    10   c = 6.28138;   (* 2 * pi *)
    11   labeled = 32;
    12  var
    13   x, y: real;
    14   i, n: integer;
    15  begin
    16   for i := 0 to labeled begin
e ------------------------------^--- Inserted keyword do
    17            x := d / i;
    18            y := exp(-x9 * sin(i * x);
E ------------------------------^--- Undefined variable
e ---------------------------------------------^--- Inserted ')'
    19            n := Round(s * y) + h;
E ---------------------------^--- Undefined function
E ---------------------------------------------^--- Undefined variable
    20            repeat
    21                     write(' ');
    22                     n := n - 1
    23            writeln('*')
e ---------------------^--- Inserted ';'
    24  end.
E -------^--- Expected keyword until
E ----------^--- Malformed declaration
  ----------^--- Unexpected end-of-file - QUIT
```

The next few sections work through examples of Pascal by correcting this program.

Correcting the First Errors

Most of the errors were *syntactic* errors, those in the format and structure of the program, rather than its content. In the output, syntax errors are marked by printing the offending line of the program followed by a line that tells where the error was detected. The second line also gives a possible cause of the error, how to recover from the error, a symbol expected at the point of error, or an indication that the input is incorrect. In the last case, the compiler can skip ahead to a point in the program where execution can continue.

In this example, the first error message indicates that the compiler detected a comment within a comment. While this is not an error in standard Pascal, it usually corresponds to an error in the program. In this case, the trailing symbol *) of the comment was omitted on line 8. To correct this problem add **)** to the end of line 8.

The second error message, following line 16, says that the compiler expected the keyword **do** before the keyword **begin** in the **for** statement. Examine the statement syntax chart on page 118 of the *User Manual*. **Do** is a necessary part of the statement. Correct this error by first finding **for** in the file. On that line insert the keyword **do** in front of **begin**.

The next error in the program is easy to pinpoint. On line 18, we did not hit the shift key, and got a 9 instead of a parenthesis. The compiler said that *x9* is an undefined variable and that a parenthesis was missing in the statement. The compiler is not suggesting that you insert a parenthesis before the semicolon (;). The error message only indicates changes that help the program continue to compile. You must determine the correct cause of the error and correct it.

This also illustreates the fact that one error in the input may lead to multiple error messages. **Pc** attempts to give only one message for each error, but a single error in the input can look like more than one error. It is also possible that **pc** does not detect an error when it occurs, but later in the input. In this example, typing **x** instead of **x9** produces an error later in the output.

The next error message, on line 19, says that the function *Round* and the variable *h* are undefined. The compiler does not recognize *Round* because in UTek all keywords and built–in procedure and function names are in lower–case letters. The compiler says that *h* is undefined because its definition was lost in the non–terminated comment on line 9. Terminating the comment takes care of this error.

The next error caused the compiler to insert a semicolon (;) before the statement on line 23 that calls writeln. Look at the program around line 23. The error is that the keyword **until** and an associated expression are omitted. Note that the error message does not indicate the actual error. The compiler corrected the most plausible error, since the omission of a semicolon (;) is a common mistake. The compiler indicates a possible fix here. It laster detected that the keyword **until** is missing, but not until it sees the keyword **end** on line 24. The combination of these two error messages indicate the problem.

The last syntactic error message says that the compiler needs an **end** keyword to match the **begin** at line 15. Since the **end** at line 24 is supposed to match this **begin**, another **begin** must have been mismatched. Before the final **end**, insert another **end** to match the **begin** at line 16.

At the end of each procedure or function, and at the end of the program, the compiler summarizes references to undefined and improperly-used variables. It also warns of potential errors. In this program, the summary warns that *c* is unused, therefore somewhat suspicious. Examining the program, you see that the constant was intended for the expression that is an argument to *sin*f, so you can correct this expression and recompile the program. To correct the expression, replace the "i" in the expression following *sin* with a "c".

The compiler suppresses warning messages for a particular procedure, function, or program when it finds severe syntax errors. This helps prevent confusing and incorrect warning messages. We are now ready to compile the program for the first time. This is what the program looks like after it is corrected:

cat -n bigger.p

```
 1  (*
 2   * Graphic representation of a function
 3   *     f(x) = exp(-x) * sin(2 * pi * x)
 4   *)
 5  program graph1(output);
 6  const
 7          d = 0.0625;    (* 1/16, 16 lines for interval [x, x+1] *)
 8          s = 32;        (* 32 character width for interval [x, x+1] *)
 9          h = 34;        (* Character position of x-axis *)
10          c = 6.28138;   (* 2 * pi *)
11          labeled = 32;
12  var
13          x, y: real;
14          i, n: integer;
15  begin
16          for i := 0 to labeled do begin
17                  x := d / i;
18                  y := exp(-x) * sin(c * x);
19                  n := round(s * y) + h;
20                  repeat
21                          write(' ');
22                          n := n - 1
23                  until n = 0;
24                  writeln('*')
25          end
26  end.
```

Executing the Second Example

You are now ready to execute the second example. The first run of this example produced this output:

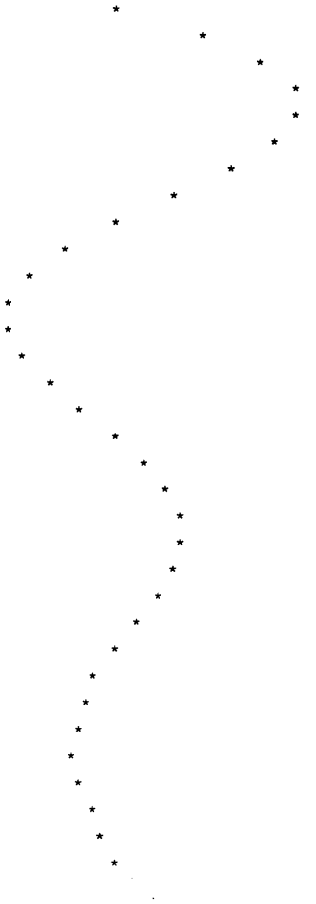
```
pc bigger.p

Floating exception (core dumped)
```

The error consists of a "division by zero" at line 17. We can replace the division sign there with a multiplication sign, then re-run the program.

The corrected program produces the following output:

pc bigger.p
a.out



Formatting the Program Listing

You can use special lines within the source text of a program to format the program listing. A blank line corresponds to a space macro in an assembler, leaving a completely blank line without a line number. A line containing only a <CTRL–L> causes a page eject in the listing, with the corresponding line number suppressed. This corresponds to an eject pseudo–instruction. See the *Options of Pc* discussion for details on the **n** and **i** options to **pc**.

Execution Profiling

An execution profile consists of a structured listing of a program, that details the number of times each statement was executed in a particular run of the program. In a program that terminates abnormally due to excessive looping or recursion, or by a program fault, the counts help find the error. Zero counts mark portions of the program that were not executed. In preliminary debugging these portions should prompt you to use new test data or re–examine the program logic. The profile is most valuable in showing the portions of the program that dominate execution time. This information is useful for source–level optimization.

An Example of Execution Profiling

A prime number is divisible only by itself and the number one. The program *primes*, written by Niklaus Wirth, determines the first few prime numbers. In compiling the program, the **p** option to **pc** is used. This option causes the compiler to generate code that determines the number of times each procedure or function in the program was executed, and the percentage of total time spent in each. When execution of the *a.out* file completes, this data is written to the file *mon.out* in the current directory. It is possible to prepare an execution profile by running the profiler **prof** on the file that you profiled. The following example illustrates this:

pc -l -p primes.p

```
Wed Aug 15 21:52 1984  primes.p

 1  program primes(output);
 2  const n = 50; n1 = 7; (*n1 = sqrt(n)*)
 3  var i,k,x,inc,labeled,square,l: integer;
 4      prim: boolean;
 5      p,v: array[1..n1] of integer;
 6  begin
 7     write(2:6, 3:6); l := 2;
 8     x := 1; inc := 4; labeled := 1; square := 9;
 9     for i := 3 to n do
10     begin (*find next prime*)
11        repeat x := x + inc; inc := 6-inc;
12           if square <= x then
13              begin labeled := labeled+1;
14                 v[labeled] := square; square := sqr(p[labeled+1])
15              end ;
16           k := 2; prim := true;
17           while prim and (k<labeled) do
18           begin k := k+1;
19              if v[k] < x then v[k] := v[k] + 2*p[k];
20              prim := x <> v[k]
21           end
22        until prim;
23        if i <= n1 then p[i] := x;
24        write(x:6); l := l+1;
25        if l = 10 then
26           begin writeln; l := 0
27           end
28     end ;
29     writeln;
30  end .
```

a.out

```
  2   3   5   7  11  13  17  19  23  29
 31  37  41  43  47  53  59  61  67  71
 73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
```

Discussion

After you compile the program, use the **prof** command to profile the data that was gathered:

prof primes.p

```
%time  cumsecs  #call  ms/call  name
 33.3     0.03     50     0.57  __doprnt
 22.2     0.05      6     3.18  _write
 11.1     0.06    306     0.03  __flsbuf
 11.1     0.07      1     9.53  _fstat
 11.1     0.08            _main
 11.1     0.09      1     9.53  _program
  0.0     0.09      1     0.00  _PCEXIT
  0.0     0.09      1     0.00  _PCLOSE
  0.0     0.09      1     0.00  _PCSTART
  0.0     0.09      1     0.00  _PFLUSH
  0.0     0.09      2     0.00  _fflush
  0.0     0.09     50     0.00  _fprintf
  0.0     0.09      1     0.00  _gtty
  0.0     0.09      1     0.00  _ioctl
  0.0     0.09      1     0.00  _isatty
  0.0     0.09      1     0.00  _nargs
  0.0     0.09      1     0.00  _profil
```

Because the program primes.p was compiled using the **-p** option to **pc**, the profiler **prof** is able to give information about how the program executed. The first field in the output gives the percentage of time spent between one symbol (statement) name and the next, and the second field represents that time in cumulative seconds. The third field gives the number of calls to a symbol, and the field labeled ms/call gives the number of milliseconds per call.

Error Messages

This section discusses the error messages displayed by the compiler, **pc**.

Compiler Syntax Errors

A few comments on the nature of syntax errors frequently made by Pascal programmers, and on the recover mechanisms of the compiler, can help you use Pascal better.

Illegal Characters

Characters such as dollar sign and exclamation mark are not part of Pascal. In the source program they are considered illegal characters, unless they are part of a constant string, a constant character, or a comment. This can happen if you leave off an opening string quote. Note that the double quote character, although used in English to quote strings, is not used to quote strings in Pascal. Most non–printing characters in the input are also illegal, except in character constants and character strings. Except for the tab and form–feed characters, non–printing characters in the input file print as question marks, so that they show in your listing.

String Errors

There is no character string of length 0 in Pascal.' Consequently the input '' (right double quote) is not acceptable. Similarly, encountering an end–of–line after an opening string quote ', without encountering the matching closing quote ', results in the error message *Unmatched for string*. You can use the character # instead of ' to delimit character and constant strings. For this reason, a misplaced # sometimes causes an error about unbalanced quotes. Similarly, a # in column one prepares programs that are kept in multiple files.

Comments Within a Comment, Non-terminated Comments

As you saw above, these errors are usually caused by leaving off a comment delimiter. You can convert parts of your program to comments without causing this error message, since there are two different kinds of comments — those delimited by braces and those delimited by asterisks.

If a comment does not terminate before the end of the input file, the compiler points to the beginning of the comment, indicating that the comment is not terminated. In this case, processing stops immediately.

Digits in Numbers

This part of the language is a minor irritation. Pascal requires digits in real numbers both before and after the decimal point. So the following statements, which would seem reasonable to Fortran users, generate error messages in Pascal:

```
Wed Aug 15 21:53 1984  digits.p:
     4  r := 0.;
e ------------^--- Digits required after decimal point
     5  r := .0;
e -----------^--- Digits required before decimal point
     6  r := 1.e10;
e ------------^--- Digits required after decimal point
     7  r := .05e-10;
e -----------^--- Digits required before decimal point
```

Replacements, Insertions, and Deletions

When the compiler encounters a syntax error in the input, the parser invokes an error recovery procedure. This procedure examines the input text immediately after the point of error and considers some simple corrections that might allow the analysis to continue. These corrections replace an input token with a different token or insert a token. Most of these changes do not cause fatal syntax errors. The exception is insertion or replacement with a symbol such as an identifier or a number. In this case the recover makes no attempt to determine which identifier or what number to insertr, and these are considered fatal syntax errors.

Consider the following example:

pc –l synerr.p

```
Wed Aug 15 21:53 1984  synerr.p

     1  program syn(output);
     2  var i, j are integer;
e ---------------^--- Replaced identifier with a ':'
     3  begin
     4   for j :* 1 to 20 begin
e ---------------------^--- Replaced '*' with a '='
e ------------------------------^--- Inserted keyword do
     5            write(j);
     6            i = 2 ** j;
e ------------------------^--- Inserted ':'
E ------------------------------^--- Inserted identifier
     7            writeln(i))
E ---------------------------------^--- Deleted ')'
     8   end
     9  end.
```

The output is as expected, except the complaint about **. This occurs because Pascal does not have an exponentiation operator. This error illustrates that you must not assume that the language has a particular feature. The compiler is unlikely to recognize the construct that you enter.

Undefined or Improper Identifiers

If the compiler encounters an identifier in the output that is undefined, the error recovery replaces it with an identifier of the appropriate class. Further references to this identifier are summarized at the end of the containing procedure or function, or at the end of the program. Similarly, if you use an identifier inappropriately an error message displays and an identifier of the appropriate type is inserted. Further incorrect references to this identifier are noted only if they involve incorrect use in a different way. All incorrect uses are summarized in the same way as undefined variable uses.

Expected Symbols and Malformed Constructs

If none of the corrections mentioned above seem reasonable, the error recovery examines the input to the left of the point of error to see if there is only one symbol that can follow this input. If this is the case, the recovery prints a message indicating that the given symbol is "expected".

In cases where none of these corrections resolve the problems in the input, the recovery can issue a diagnostic indicating that the input is "malformed". If necessary, the compiler can then skip forward in the input to a place where compilation can continue. This process can miss some errors in the text.

Consider the following example:

pc -l synerr2.p

```
Wed Aug 15 21:53 1984  synerr2.p

     1  program synerr2(input,outpu);
     2  integer a(10)
E ------^--- Malformed declaration
     3  begin
     4   read(b);
E ------------------^--- Undefined variable
     5   for c := 1 to 10 do
E ------------------^--- Undefined variable
     6            a(c) := b * c;
E ----------------------^--- Undefined procedure
E --------------------------^--- Malformed statement
     7  end.
E 1 - File outpu listed in program statement but not declared
In program synerr2:
  E - a undefined on line 6
  E - b undefined on line 4
  E - c undefined on lines 5 6
```

The word "output" is misspelled, giving a Fortran-style variable that the compiler diagnoses as a "malformed declaration". On line 6, parentheses instead of brackets were used for subscripting. The compiler notes that *a* is not defined as a procedure. This occurred because procedure and function argument lists are not delimited by parentheses in Pascal. You cannot assign to procedure calls, so the compiler diagnosed a "malformed statement".

Expected and Unexpected End-of-File

If the compiler finds a complete program, but there is more non-comment text in the input file, it indicates that an end-of-file is expected. This situation can occur after a bracketing error, or if too many ends are present in the input. Following recovery, a message may appear saying that a period was expected, since the period is the symbol that terminates a program.

If severe errors in the input prohibit further processing the compiler produces an error message followed by "QUIT". Consider the following example:

pc -l mism.p

```
Wed Aug 15 21:53 1984  mism.p

     1  program mismatch(output)
     2  begin
e ------^--- Inserted ';'
     3    writeln('***');
     4    { The next line is the last line in the file }
     5    writeln
E ----------------------^--- Malformed declaration
  ----------------------^--- Unexpected end-of-file - QUIT
```

Compiler Semantic Errors

The extremely large number of semantic error messages produced by the compiler make it impossible to discuss each message or group of messages in detail. This section explains the typical formats and the terminology used in error messages, so that you can interpret them. If you do not understand a particular error message, refer to the *User Manual* by Jensen and Wirth for examples.

Error Message Format

As we saw in the last example program, the error messages from the Pascal compiler include the number of a line in the text of the program, as well as the text of the error message. Occasionally, the error occurs on the line number containing a bracketing keyword like **end** or **null**. In this case, the error message might refer to the previous statement. This happens because of the method the compiler uses to sample line numbers. The absence of a trailing semicolon in the previous statement causes the line number corresponding to the **end** or **null** to become associated with the statement. As Pascal is a free-format language, the line number associations can only be approximate, and they may seem arbitrary.

Incompatible Types

Since Pascal is very much a language of types, many semantic errors manifest themselves as type errors. These are called type clashes by the compiler. The types allowed for various operators in the language are summarized on page 108 of the Jensen–Wirth *User Manual.* It is important to know that the Pascal compiler, in its diagnostics, distinguishes between the following type classes:

- array
- Boolean
- char
- file
- integer
- pointer
- real
- record
- scalar
- string

These words display in many error messages. So if you try to assign an integer value to a char variable, an error message displays:

Sat Aug 15 14:50 1984 clash.p:
E7 — Type clash; integer is incompatible with char
. . . Type of expression clashed with variable in assignment

In this case, one error produced a two–line error message. If the same error occurs more than once, the same diagnostic displays each time.

Scalar

The only class whose meaning is not self–explanatory is scalar. Scalar has a precise meaning in the Jensen–Wirth *User Manual*, where it refers to char, integer, real, and Boolean types, as well as the enumerated types. For the purposes of the Pascal compiler, scalar in an error message refers to a user–defined, enumerated type. For example, color in this example:

```
type color = (red, green, blue)
```

For integers, the more explicit denotation integer is used. Although it is correct to refer to an integer variable as a scalar variable, **pc** prefers the more specific indication.

Function and Procedure Type Errors

For built–in procedures and functions, two kinds of errors occur. If the routines are called with the wrong number of arguments a message like this displays:

Sat Aug 15 14:55 1984 sin2.p:
E12 – sin's argument must be integer or real, not char

Non-readable and Non-writable Scalars

The messages stating that scalar (user–defined) types cannot be written to and read from files are often mysterious. For example, if you define:

type color = (red, green, blue)

With this definition standard Pascal does not associate these constants with the strings "red", "green", and "blue". UTek Pascal has a feature that allows enumerated types to be read an written. However, if the program is supposed to be portable, you have to write your own routines to perform these functions. Standard Pascal allows only the reading of characters, integers, and real numbers from text files. You cannot read strings or Booleans.

Expression Diagnostics

The error messages for semantically incorrect expressions are very explicit. Consider this sample compilation:

pc –l expr.p

```
Wed Aug 15 21:53 1984  expr.p

    1  program x(output);
    2  var
    3   a: set of char;
    4   b: Boolean;
    5   c: (red, green, blue);
    6   p: ^ integer;
    7   A: alfa;
    8   B: packed array [1..5] of char;
    9  begin
   10   b := true;
   11   c := red;
   12   new(p);
   13   a := [];
   14   A := 'Hello, yellow';
   15   b := a and b;
   16   a := a * 3;
   17   if input < 2 then writeln('boo');
   18   if p <= 2 then writeln('sure nuff');
   19   if A = B then writeln('same');
   20   if c = true then writeln('hue''s and color''s')
   21  end.
E 14 - Constant string too long
E 15 - Left operand of and must be Boolean, not set
E 16 - Cannot mix sets with integers and reals as operands of *
E 17 - files may not participate in comparisons
E 18 - pointers and integers cannot be compared - operator was <=
E 19 - Strings not same length in = comparison
E 20 - scalars and Booleans cannot be compared - operator was =
e 21 - Input is used but not defined in the program statement
In program x:
  w - constant green is never used
  w - constant blue is never used
  w - variable B is used but never set
```

This example is completely artificial, but it illustrates the clarity of the error messages about expressions.

Type Equivalence

Several messages produced by the Pascal compiler complain about non-equivalent types. In general, Pascal considers variables to have the same type only if they were declared with the same constructed type, or with the same type identifier. Thus, the variables *x* and *y* in this example do not have the same type:

```
var
        x: ^ integer;
        y: ^ integer
```

So the assignment **x :=y** results in an error message.

So you must always declare a type, then use it to declare a variable. For example:

```
type insert = ^ integer;
var x: insert;y: insert;
```

Since the parameter to a procedure or function must be declared with a type identifier, instead of a constructed type, you must declare any type that is used in this way.

Unreachable Statements

Pascal prints error messages about unreachable statements. Such statements usually correspond to errors in the program logic. A statement is considered reachable if there is a potential path of control, even if that path is never taken. So this statement does not result in an error message:

```
if false then
        writeln('impossible!')
```

Goto Directed to Structured Statements

The compiler complains about **goto** statements that transfer control into structured statements. It does not allow such jumps, nor does it allow branching from the **then** part of an **if** statement into the **else** part. The compiler checks for this transfer of control only within a single procedure or function.

Unused and Unset Variables

Pc does not clear variables to 0 at procedure and function entry, unless run-time checking is enabled using the **C** option. It is not good programming practice to rely on the initialization of the **C** option. To discourage this practice, and to help detect errors in program logic, **pc** gives a "w" warning error for:

- use of a variable that is never assigned a value
- a variable that is declared, but never used, distinguishing between variables whose values are computed and those completely unused

In fact, these messages apply to all declared items. So a **const** or **procedure** that is declared but never used is noted. You can use the **w** option of **pc** to suppress these warnings.

Compiler Panics

One kind of error that rarely happens, but causes termination of all processing, is called a panic. A panic indicates a compiler-detected internal inconsistency. A typical panic message is:

```
snark (rvalue) line=110 yyline=109
Snark in pc
```

If this kind of message displays, compilation terminates immediately. Contact the Tek representative in charge of your system software.

Input/Output Errors

Other errors that you may encounter when you run **pc** relate to input and output. If **pc** cannot open the file you specify, or if the file is empty, an error message displays.

Run-time Errors

The example program *bigger.p* had a run-time error. In this section we attempt to give general descriptions of run-time errors. Use the **C** option to activate run-time checking.

As an example of a run-time error, suppose that you accidentally declared the constant *nl* to be 6, instead of 7. This error is on line 2 of the program *primes*, used as an example above. If we run this program the following response displays:

```
pc -C -g primes.p
a.out

  2   3   5   7   11   13   17   19   23   29
 31  37  41  43  47   53   59   61   67   71
 73  79  83  89  97  101  103  107  109  113
127 131 137 139 149  151  157  163  167

a.out: Subscript value of 7 is out of range
Trace/BPT trap (core dumped)
```

Now you can run the symbolic debugger, **sdb**, on your program to find the error. Enter:

```
sdb
*t
```

After you enter the debugger, the asterisk displays as the **sdb** prompt. The **t** command give you a backtrace of the stack to tell you where the program died.

Interrupts

If the program is interrupted during execution a core image is produced. You can use **sdb** to examine this core image and look at a stack backtrace of the program.

Input/Output Interaction Errors

The final class of compiler errors results from inappropriate interactions with files, including the user's terminal. This includes bad formats for integers and real numbers that become evident when the file is read.

Input/Output

This section describes features of the Pascal input/output environment on UTek. The most basic aspects of input and output using the UTek shell are described in other documents. For information on file redirection and the redirection of output into other UTek commands, see section 2B, *Introduction to the Shell.*

End–of–file and End–of–line

An extremely common problem encountered by new users of Pascal, especially in the UTek environment, relates to the definitions of end–of–file (*eof*) and end–of–line (*eoln*). These functions are defined at the beginning of execution, indicating whether the input device is at the end of a line or at the end of a file. Setting *eof* or *eoln* corresponds to an implicit read where the input is inspected, but it is not used up. In fact, the system cannot know whether the input is at the end–of–file or at the end–of–line, unless it attempts to read a line from it. If the input is from a previously–created file, the reading can take place without your entering something during execution. However, if the input is from a terminal, then the user must type it.

Pascal is designed so that an initial read is not necessary. At any given time, the Pascal system may or may not know whether the end–of–file or end–of–line conditions are true. Internally, these functions can have three values — true, false, and an indeterminate state. The indeterminate state says "I don't know yet; if you ask me I will have to find out." All files remain in this indeterminate state until the Pascal program requires a value for *eof* or *eoln* either explicitly or implicitly, for example, in a call to *read*. If you force the Pascal system to determine whether the input is at the end–of–file or the end–of–line, the program must try to read from the input. Consider the following example code:

```
while not eof do begin
        write('number, please ?');
        read(i);
        writeln(;that was a ',i:2);
        write('number, please ?)
end
```

At first glance this appears to be a correct program that requests, reads, and echos numbers. Notice, however, that the **while** loop asks whether *eof* is true before the request is printed. This forces the Pascal system to decide whether the input is at the end–of–file. The system simply waits for the user to type a line. By inserting the prompt before testing *eof*, this code avoids the problem:

```
write('number, please ?');
while not eof do begin
        read(i);
        writeln('that was a',i:2);
        write('number, please ?')
end
```

The user must still type a line before the **while** test is completed, but the prompt asks for it. This example, however, is still not correct. To understand why, it is necessary to know that there is a blank character at the end of each line in a Pascal text file. The *read* procedure, when reading integers or readl numbers, is defined to return a zero value and set the end–of–file condition only if there are blanks left in the file. If, however, there is a number remaining in the file, the end–of–file condition is not set, even if it is the last number. *Read* never reads the blanks after the number, and there is always at least one blank. So the modified code outputs the spurious line:

```
that was a 0
```

This displays at the end of a session when the end–of–file is reached. The simplest way to correct this problem is to use the procedure *readln* instead of *read*. In general, unless you test the end–of–file condition both before and after calls to *read* or *readln*, there are inputs where the program tries to read past the end–of–file.

More About End-of-Line

To have a good understanding of when *eoln* is true, recall that in any file there is a special character indicating end–of–line. In effect, the Pascal system always reads one character ahead of the Pascal *read* commands. For instance, in response to read(ch), the system sets *ch* to the current input characters and gets the next input character. If the current input character is the last character of the line, the next input character from the file is the new–line character, the UTek line separator. When the read routine gets the new–line character, it replaces that character with a blank. This ends every line with a blank, and sets *eoln* to true. *Eoln* is true as soon

as the program reads the last character of the line, and before it reads the blank character corresponding to the end-of-line. So it is almost always a mistake to write a program that deals with input in the following way:

read(ch);
if eoln then
done with line
else
normal processing

It is almost certain that this program will ignore the last character in the line. The read(ch) belongs with the normal processing.

Given this framework, the *readln* call becomes very useful. This call is defined as:

```
while not eoln do
        get(input);
get(input);
```

This advances the file until the blank that corresponds to the end-of-line is the current input character, then discards the blank. The next character available from *read* is the first character of the next line.

Output Buffering

It is extremely inefficient for the Pascal system to send each character to the user's terminal as the program generates it for output; even less efficient if the output is the input of another program. To gain efficiency, the Pascal system buffers the output characters. That is, it saves them in memory until the buffer is full and then emits the entire buffer in one interaction. However, so that interactive prompting works, this prompt displays before the Pascal system waits for a response. Because of this, Pascal normally prints all the output that has been generated for a particular file whenever:

- *a writeln* occurs
- the program reads from the terminal
- the procedure *message* or *flush* is called

So in this code, the output integers do not display until the *writeln* occurs:

for i:=1 to 5 do begin
        write(i:2);
        processing
end
writeln

By setting the **b** option to 0 before the program statement, the output is completely unbuffered, with a corresponding degradation in program efficiency. For example:

```
(*$b*0)
```

See the *Options* section for more information on this feature.

Files, Reset, and Rewrite

You can use extended forms of the built-in functions *reset* and *rewrite* to associate UTek file names with Pascal file variables. When a file other than *input* or *output* is read or written, the reading and writing must be preceded by a *reset* or *rewrite* call. In general, if the Pascal file variable has never been used before, no UTek filename is associated with it. By mentioning the file in the **program** statement, you can associate a UTek file with the same name as the Pascal variable. If you do not mention a file in the **program** statement, and use it for the first time with the **reset** or **rewrite** statements, a temporary file is generated. The name of the temporary file is *#tmp.x* for some number *x*. This temporary UTek filename is associated with the Pascal file. The advantage of using temporary files is that they are automatically removed by the Pascal system as soon as they become inaccessible. They are not removed if a runtime error causes termination while they are accessible.

To associate a UTek pathname with a Pascal file variable, give that name in the **reset** or **rewrite** call. For example, you can associate the Pascal file *data* with the file *primes* in our earlier example. Enter:

```
reset(data, 'primes')
```

It is not necessary to mention *data* in the program statement, but it helps document the program. The second parameter to **reset** and **rewrite** can be any string value, including a variable. So the names of UTek files associated with Pascal file variables are read when the program executes.

Argc and Argv

Each UTek process receives a variable-length sequence of arguments. Each argument is a variable-length character string. The built-in function *argc* and the built-in procedure *argv* can be used to access and process these arguments. The value of the function *argc* is the number of arguments to the process. By convention, the arguments are treated as an array, and indexed from 0 to *argc*–1, where the zeroth argument is the name of the program being executed. The rest of the arguments are those passed to the command on the command line. The following command invokes the program in the file *obj*, where *argc* has a value of 4.

```
obj /etc/motd /usr/s/words hello
```

The zeroth element accessed by *argv* is *obj*, the first */etc/motd*, and so on.

Pascal does not provide variable–size arrays, nor does it allow character strings of varying length. For this reason, *argv* is a procedure:

argv(i,a)

In the procedure *i* is an integer and *a* is a string variable. This procedure call assigns the *i*'th argument of the current process to the string variable *a*. (The *i*'th argument can be truncated or blank padded). The file manipulation routines *reset* and *rewrite* strip trailing blanks from their optional second arguments. So blank padding is not a problem in the case where the arguments are filenames.

Now we can execute a Pascal program called *kat*. This program has the same syntax as the UTek system program **cat**.

cat kat.p

```
program kat(input, output);
var
    ch: char;
    i: integer;
    name: packed array [1..100] of char;
begin
    i := 1;
    repeat
        if i < argc then begin
            argv(i, name);
            reset(input, name);
            i := i + 1
        end;
        while not eof do begin
            while not eoln do begin
                read(ch);
                write(ch)
            end;
            readln;
            writeln
        end
    until i >= argc
end { kat }.
```

If this program is in the file *kat.p*, enter:

```
pc kat.p
mv a.out kat
kat < primes

  2   3   5   7  11  13  17  19  23  29
 31  37  41  43  47  53  59  61  67  71
 73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
```

Given standard input, **kat** copies each line.

Details on the Components of the System

Options

Pc takes a number of options.

Except for the **b** option, which takes a single–digit value, each option is set on or off. When you enter an on/off valued option on the **pc** command line it inverts the default setting of that option. This command enables option **I**, since the default value of option **I** is off:

```
pc -I foo.p
```

The **–I** option enables the run–time tests option.

In addition to inverting the default settings of **pc** options on the command line, you can also control the **pc** options within the body of the program. Use special comment delimiters to control the options. For example:

```
{$I-}
```

The right brace comment delimiter is immediately followed by the dollar sign character. The dollar sign signals the start of the option list, and after it enter a sequence of letters and option controls, separated by commas. To enable options enter:

```
{$I+ option +}
```

To clear the options enter:

{*option* –}

Notice that the addition sign always enables an option and the minus sign always disables it, not matter what the default is. So the minus sign has a different meaning in an option comment than on the command line.

Options to pr

The following options apply to the **pc** Pascal compiler. This section discusses each option, its default setting, its setting on the command line, and a sample command using the options. Most options have on/off values, with the **b** option taking a single–digit value.

Buffering the File Output —b

The **b** option controls the buffering of the file *output*. The default is line buffering, and the buffer is flushed at each reference to the file *input*. Entering **b** on the command line causes the standard output to be block buffered. For example, enter:

pc —b *filename*

The **–b** option can also be controlled in comments. It takes a single–digit value rather than an on/off setting. For example, a value of 0 causes the file output to be unbuffered:

{**$b0**}

Any value 2 or greater causes block buffering and is equivalent to entering the option on the command line. The option control comment setting **b** must precede the **program** statement.

Make a Listing —l

The **–l** option enables a listing of the program, and its default value is off. When specified on the command line, it displays a header line that identifies the version of the compiler. The header also gives the modification time of the file.

Standard Pascal Only —s

The **–s** option causes features of the UTek implementation that are not found in standard Pascal to be diagnosed as "s" warning errors. This option defaults off and is enabled when mentioned on the command line. Some of the features diagnosed include: non–standard procedures and functions, extensions to the write procedure, and the padding of constant strings with blanks. In addition, all letters are mapped to lowercase except in strings and characters, so keywords and identifiers are effectively ignored. This option is most useful when a program is transported to another system.

Runtime Tests —C

This option generates tests to verify that subrange variable values are within bounds when the program executes. Enabling runtime tests also verifies **assert** statements.

Suppress Warning Diagnostics —w

The **-w** option, which defaults on, lets the compiler print a number of warnings about inconsistencies in the input program. Turn this option off with a comment:

```
{$w-}
```

Or you can turn it off on the command line:

pc -w *filename*

Generate Assembly Language —S

The program compiles and the assembly language output creates a file with a *.s* suffix. For example, this command creates the file foo.s:

```
pc —S foo.p
```

No executable file is created.

Symbolic Debugger Information —go

The **-g** option generates information needed by **sdb**, the symbolic debugger. See section 5L for details on **sdb**.

Redirect the Output File —o

The *name* argument following the **-o** option specifies a name other than *a.out* for the output file. Its typical use is to name the compiled program using the root of the filename. For example, this command names the compiled program *myprog*:

```
pc -o myprog myprog.p
```

Generate Counters for a prof Execution Profile —p

The compiler produces code that counts the number of times each routine is called. The profiling is based on a periodic sample taken by the system. This results in faster execution, at somewhat of a loss in accuracy. See *prof(1)* in your *UTek Command Reference* for a complete description.

Run the Object Code Optimizer —O

When you specify this option the output of the compiler is run through the object-code optimizer. This increases compile time, in exchange for a decrease in the size of the compiled code and a decrease in execution time.

Pxref

You can use the cross-reference program **pxref** to make cross-referenced listings of Pascal programs. To produce a cross-reference of the program in the file *foo.p* enter:

```
pxref foo.p
```

The cross-reference is, unfortunately, not block structured. See *pxref(1)* in your *UTek Command Reference* for details.

Multi-file Programs

A text inclusion facility is available in Pascal. This facility interpolates source text from other files into the source stream of the compiler. You can use it to divide large programs into more manageable pieces for ease in editing, listing, and maintenance.

The **include** facility is based on that of the UTek C compiler. To trigger it, place the character # in the first portion of the line. Then, after an arbitrary number of blanks or tabs, enter the word include, followed by a filename enclosed in single or double quotes. The filename can be followed by a semicolon to treat it as a pseudo-Pascal statement. The filenames of included files must end in *.i*. This is an example of the use of included files in a main program:

```
program compiler(input, output, obj);
#include "globals.i"
#include "scanner.i"
#include "parser.i"
#include "semantics.i"
begin
  {main program}
end.
```

When the compiler encounters the **include** pseudo-statement in the input, lines from the included file are interpolated into the input stream. For the purposes of compilation and run-time diagnostics, and statement numbers in the listings and post-mortem backtraces, the lines in the included file are numbered from 1. You can nest includes up to 10 deep.

See the descriptions of the **i** option to **pc**. You can use it to control listing when **Include** files are present.

When the compiler encounters a non-trivial line in the source text after an **include** finishes, the "popped" filename is printed, in the same manner as above.

When you are not making a listing, you can print the "popped" filename in error messages. If the current filename has changed since the last filename was printed, the filename prints.

Separate Compilation with pc

A separate compilation facility is provided with the **pc** compiler. This facility lets you divide programs into a number of files. The pieces can be compiled individually, then linked together later. This is especially useful for large programs, where small changes would otherwise require time-consuming recompilation of the entire program.

Normally, **pc** expects entire Pascal programs. However, with the **-c** option on the command line, it accepts a sequence of definitions and declarations and compiled them into a *.o* file. The *.o* file is linked with a Pascal program later. So that procedures and functions are available across separately compiles files, you must declare them with the directive **external**. This directive is similar to the directive **forward** in that it must precede the resolution of the function or procedure. You must specify format parameters and function result types at the **external** declaration and not at the resolution.

Pc performs type checking across separately-compiled files. Since Pascal type definitions define unique types, any types that are shared between separately-compiled files must be the same definition.

This problem is solved using a facility similar to the **include** facility discussed above. You can place definitions in files with the *.h* suffix, and in the files included by separately-compiled files. Each definition from a *.h* file defines a unique type, and all uses of a definition from the *.h* file define the same type.

Similarly, the facility allows the definition of **consts** and the declaration of **labels, vars** and **external** functions and procedures. Thus procedures and functions that are used between separately-compiled files must be declared **external**. They must be declared external in a *.h* included by any file that calls or resolves the function or procedure. Conversely, functions and procedures declared **external** can only be declared **external** in *.h* files. These files can be included only at the outermost level, so they define or declare global objects. Note that since only **external** function and procedure declarations (not resolutions) are allowed in *.h* files, statically nested functions and procedures cannot be declared **external**.

This example shows the use of included *.h* files in a program:

```
program compiler(input, output, obj);
#include "globals.h"
#include "scanner.h"
#include "parser.h"
#include "semantics.h"
begin
  {main program}
end.
```

In the main program this might include the definitions and declarations of all the global **labels, consts, types vars** from the file *globals.h*, and the **external** function and procedure declarations for each of the separately–compiled files for the scanner, parser, and semantics. The header file *scanner.h* contains declarations of the form:

```
type
  token = record
     { token fields }
  end;

function scan (var inclusion: text): token;
  external;
```

Then the scanner might be in a separately–compiled file containing:

```
#include "globals.h"
#include "scanner.h"

function scan;
begin
   { scanner code}
end;
```

This file includes the same global definitions and declarations, and it resolves the scanner functions and procedures declared **external** in the file *scanner.h*.

Appendix to Wirth's Pascal Report

This section is an appendix to the definition of the Pascal language in Niklaus Wirth's *Pascal Report*, so this section precisely defines this UTek implementation. It includes a summary of extensions to the language, shows how the undefined specifications were resolved, gives limitations and restrictions of the current implementation, and lists the added functions and procedures available.

Extensions to the Pascal Language

This section defines non-standard language constructs available in this implementation of Pascal. The **s** option of **pc** detects these extensions.

String Padding

UTek Pascal pads constant strings with blanks in expressions and as value parameters, to make them as long as required. The following is a legal UTek Pascal program:

```
program x(output);
var z : packed array [1 .. 13] of char
begin
  z := 'red';
  writeln(z)
end;
```

The padded blanks are added on the right. So the assignment above is equivalent to Standard Pascal:

```
z := 'red
```

Octal Constants, Octal and Hexadecimal Write

You can give octal constants as a sequence of octal digits, followed by the character 'b' or 'B'. For example:

```
write(a:n oct)
write(a:n hex)
```

These forms cause the internal representation of expression *a*, which must be Boolean, character, integer, pointer, or a user-defined enumerated type, to be written in octal or hexadecimal.

Assert Statement

An **assert** statement evaluates a Boolean expression each time the statement is executed. A run–time error results if any of the expressions is evaluated false. The **assert** statement is treated as a comment if run–time tests are disabled. The syntax for **assert** is:

assert *<expression>*

Enumerated Type Input/Output

Enumerated types can be read and written. On output, the stringname associated with the enumerated value is output. If the value is out of range, a run–time error occurs. On input an identifier is read and looked up in a table of names associated with the type of the variable, and the appropriate internal value is assigned to the variable being read. If the name is not found in the table, a run–time error occurs.

Structure Returning Functions:

Another extension allows function to return arbitrary–size structures, rather than just scalars.

Separate Compilation

The **pc** compiler allows separate compilation of programs. You can compile procedures and functions declared at the global level. When you load the program, the compiler type checks calls to separately–compiled routines to ensure that the entire program is consistent.

Resolution of Undefined Specifications

Each Pascal file variable is associated with a UTek filename. Except for *input* and *output* files, which do not conform to some of the rules, a filename becomes associated with a file in three ways:

- a global Pascal file variable in the **program** statement is associated with a file of the same name
- a file reset or rewritten using the two–argument form of **reset** or **rewrite** is associated with a file of the same name
- a file reset or rewritten without specifying a name in the second argument has the temporary name #tmp.x. Temporary files are automatically removed when their scope is exited.

The Program Statement

The syntax of the program statement is:

program <**id**>**(** <**file id**>{ **,** <**file id**>} **)** **;**

The file identifiers (except *input* and *output*) must be declared as variables of **file** type in the global declaration.

The Files Input and Output

The formal parameters *input* and *output* are associated with the standard input and output. The following rules apply to these files:

- The program heading **must** contains the formal parameter output. If you use *input*, explicitly or implicitly, you must also declare it there.
- Unlike other files, *input* and *output* must not be defined in a declaration. Their declaration is automatically: **var input, output: text**.
- You can use the **reset** procedure on *input*. If no UTek filename is associated with *input*, and no filename is given, the compiler tries to 'rewind' input. If the 'rewind' fails, a run-time error occurs. **Rewrite** calls to output initially do not have an associated file. So this simple statement associates a temporary name with *output*: rewrite(output).

Details for Files

To read a file other than *input*, the reading must be initiated by a call to the **reset** procedure. This causes the Pascal system to try to open the associated UTek file for reading. If this fails, a run-time error occurs. To write to a file other than *output*, the write request must be initiated by a **rewrite** call. This causes the Pascal system to create the associated UTek file and to then open the file for writing.

Buffering

The value of the **b** option determines the buffering for *output* at the end of the **program** statement. If its default value is 1, *output* is buffered in blocks of up to 1024 characters. The buffer is flushed whenever a writeln occurs, and at each reference to the file *input*. If the value of the **b** option is 0, output is unbuffered. A value of 2 or more gives block buffering without line or *input* reference flushing. All other output files are buffered in blocks of 1024 characters. Output buffers are flushed when the files are closed at scope exit, and when the procedure **message** is called. Output buffers can also be flushed using the built-in procedure **flush**.

The Character Set

UTek uses the seven-bit ASCII character set. ASCII recognizes the standard Pascal symbols **and, or, not**, <=, >=, <>, and ^. Less portable are the synonyms for and, or and not:

&	**and**
|	**or**
~	**not**

Uppercase and lowercase characters are considered distinct. Keywords and built-in procedure and function names are in lowercase letters. So the identifiers GOTO and GOto are distinct from each other and from the keyword **goto**. The standard type boolean is available as Boolean.

The single quote or the pound sign (#) delimits character strings and constants. The pound sign character has no special meaning when it is the first character on a line.

The Standard Types

The standard type **integer** is defined as:

```
type integer = minint .. maxint;
```

Integer is implemented with 32-bit twos-complement arithmetic. Pre-defined constants of type **integer** are:

```
const maxint = 2147483647; minint = —21747483648;
```

The standard type **char** is defined as:

```
type char = minchar .. maxchar;
```

Built-in character constants are minchar and maxchar, bell and tab, ord(minchar) = 0, and ord(maxchar) = 127.

The type **real** is implemented using 64-bit floating point arithmetic. The floating point arithmetic is done in rounded mode, and provides approximately 16 digits of precision with numbers as small as 10 to the negative 308th power, and as large as 10 to the 308th power.

Comments

You can delimit comments by right and left braces or by (* and *). If a left brace appears in a comment delimited by a right and left brace, a warning message displays. A similar warning prints if the sequence (* appears in a comment delimited by (* and *).

Option Control

You can control compiler options in two separate ways. You can enter many of the options on the command line when you invoke **pc**. Enter these options as one or more letters preceded by —. This is the most common way to change options from their default setting in UTek.

If you want more control over portions of the program where options are required, place the option in comments. For example, to specify the l and s options enter the following as the first line of the program:

```
{$l+,s+ listing on, standard Pascal}
```

This example consists of the dollar sign character as the first character of the comment, and a comma–separated list of directives. Directives consist of a letter designating the option, followed by an addition sign to turn the option on, or a subtraction sign to turn the option off.

Notes on the Listings:

The first page of a listing includes a banner line indicating the version and date of generation of **pc**. It also includes the UTek pathname of the source file and its date of last modification.

In the body of the listing, lines are numbered consecutively, and they correspond to the line numbers of the editor. There are two characters you can use to format the listing: <CTRL–L> and a blank. Both should be placed on a line by themselves. <CTRL–L> causes a page eject in the listing, and the blank line causes the line number to be suppressed in the listing. These correspond to the eject and space macros found in many assemblers. Non–printing characters print as a question mark in the listing.

The Standard Procedure Write

If no minimum field–length parameter is specified for a **write**, the following default values are assumed:

integer	10
real	22
Boolean	length of true or false
char	1
string	length of the string
oct	11
hex	8

Specifically indicate the end of each line in a text file by **writeln(f)**, where **writeln(output)** can be written as **writeln**. For UTek, the built-in function **page(f)** puts a single ASCII form-feed character on the output file.

Restrictions and Limitations

Files

Files cannot be members of files or members of dynamically-allocated structures.

Arrays, Sets, and Strings

The calculations involving array subscripts and set elements are done with 16-bit arithmetic. This restricts the types over which arrays and sets can be defined. The lower bound of such a range must be greater than or equal to -32768, and the upper bound less than 32768. In particular, strings can have any length from 1 to 65535 characters, and sets can contain no more than 65535 elements.

Line and Symbol Length

There is no intrinsic limit on the length of identifiers. Identifiers are considered distinct if they differ in any single position over their entire length. The maximum input line length is over 160 characters.

Procedure and Function Nesting, Program Size

You can nest procedures and functions up to 20 levels. There is not a fundamental, compiler-defined limit on the size of the program that you can compile.

There is no limit on the number of variables, and a definite limit of 65535 bytes per variable.

Overflow

The hardware of your workstation performs overflow checking.

Added Types, Operators, Procedures, and Functions

Additional Predefined Types

The type **alfa** is predefined as:

```
type alfa = packed array [1..10] of char
```

The type **intset** is predefined as:

```
type intset = set of 0..127
```

In most cases the context of an expression involving a constant set lets the compiler determine the type of the set, even though the constant set itself may not uniquely determine this type. In the cases where you cannot determine the type of the set from context, the expression type defaults to a set over the entire base type unless the base type is integer. In the latter case, the type defaults to the current binding of **intset**. **Intset** must be "type set of integer" at that point, where a "set of integer" means "a subrange of integer."

Additional Predefined Operators

The greater than and less than signs of proper set illustrates are available. With sets a and b note that:

```
(not (a < b)) <> (a >= b)
```

Non-standard Procedures

argv(i,a)	where *i* is an integer and *a* is a string variable. This assigns the (possibly truncated or blank– padded) *i*'th argument to the invocation of the current UTek process to the variable *a*. The range of valid *i* is 0 to argc–1.
date(a)	assigns the current date to the alfa variable *a* in the format *dd mmm yy*, where *mmm* is the first three characters of the month.
flush(f)	writes the output buffered for Pascal file *f* into the associated UTek file.
halt	terminates the execution of the program with a control flow backtrace.
linelimit(f,x)	with *f* a text file and *x* an integer expression, causes the program to terminate abnormally if more than *x* lines are written on file *f*. If *x* is less than 0 then no limit is imposed.

message(x,..)	causes the parameters to be written unbuffered on the diagnostic unit 2, usually the user's terminal.
remove	where *a* is a string, this removes the UTek file whose name is *a*
reset(f,a)	associates the file *a* with *f*, in addition to the normal function of *reset*.
rewrite(f,a)	analogous to *reset* above.
stlimit(i)	where *i* is an integer, the statement limit is *i* statements. Specifying the **—p** option to **pc** disables this limit.
time(a)	the current time, in the format *hh:mm:ss* is assigned to the alfa variable *a*.

Non-standard Functions

argc	returns the count of arguments when the Pascal program was invoked.
card(x)	returns the number of elements (cardinality) in the set *x*.
clock	returns the number of milliseconds the CPU used by this process.
expo(x)	yields the integer-valued exponent of the floating point-representation of *x*.
random(x)	in this case *x* is a real parameter that is evaluated, but otherwise ignored. This function invokes a linear congruential random number generator. Successive seeds are generated as (seed*a + c)mod m. The new random number is a normalization of the seed to the range 0.0 to 1.0; a is 62605, c is 113218009, and m is 53670912. The initial seed is 7774755.
seed(i)	sets the random number generator seed to *i* and returns the previous seed.
sysclock	returns the number of CPU milliseconds used by this process.
undefined(x)	a Boolean function whose argument is a real number and always returns false.
wallclock	returns the time in seconds since 00:00:00 Greenwich Mean Time on January 1, 1970.

Features Not Available in UTek Pascal

The following features are not available with UTek Pascal:

- segmented files, and associated functions and procedures
- the function *trunc* with two arguments
- arrays whose indexes exceed the capacity of 16 bit arithmetic

Lexical Analyzer Generator (lex)

Introduction

Lex is a program generator that produces a program in a general purpose language that recognizes regular expressions. It is designed for *lexical* processing of character input streams. It accepts a high–level, problem–oriented specification for character string matching. The regular expressions are specified by you (the user) in the source specifications given to **lex**. The **lex** program generator source is a table of regular expressions and corresponding program fragments. The table is translated to a program that reads an input stream, copies the input stream to an output stream, and partitions the input into strings that 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 procedure generated by **lex**. The program fragments written by you are executed in the order in which the corresponding regular expressions occur in the input stream.

You supply the additional code beyond the expression matching needed to complete the tasks, possibly including codes written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for your program fragments. Thus, a high–level expression language is provided to write the string expressions to be matched, while your freedom to write actions is unimpaired.

The **lex** written code is not a complete language, but rather a generator representing a new language feature that 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 you. Compatible run–time libraries 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, your background, and the properties of local implementations. At present, the only supported host language is the C language, although FORTRAN (in the form of Ratfor) has been available in the past. The **lex** generator exists on the UTek operating system, but the codes generated by **lex** may be taken anywhere the appropriate compilers exist. appropriate compilers exist.

The **lex** program generator turns your expressions and actions (called *source*) into the host general purpose language; the generated program is named **yylex**. The **yylex** program recognizes expressions in a stream (called *input*) and performs the specified actions for each expression as it is detected. See Figure 5H–1.

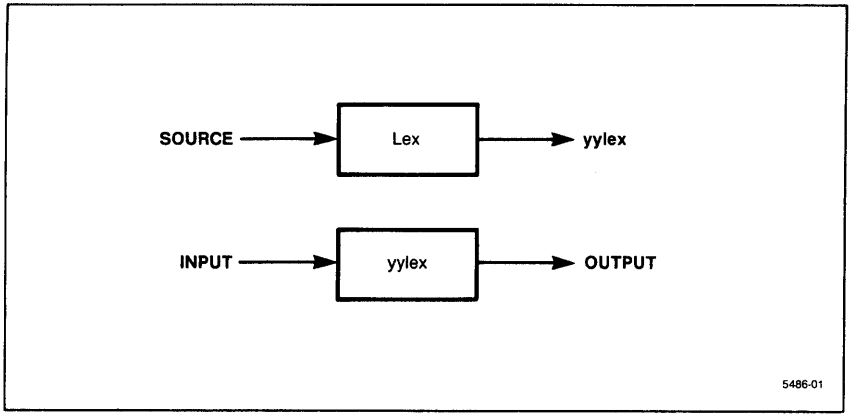


Figure 5H-1. Overview of Lex.

For example, consider a program to delete from the input all spaces or tabs at the ends of lines:

```
% %
[ \t]+$    ;
```

This input is all that is required. The program contains a **%%** delimiter to mark the beginning of the *rules*. Each rule is on a line. This rule contains a regular expression that matches one or more instances of the space or tab characters (written for visibility, in accordance with the C language convention) and occurs prior to the end of a line. The brackets indicate the character class made of space and tab; the + indicates "one or more . . . "; and the **$** indicates "end of line." No action is specified, so the **yylex** program generated by **lex** ignores these characters. Everything else is copied. To change any remaining string of spaces or tabs to a single space, add another rule.

```
%%
[ \t]+$   ;
[ \t]+    printf(" ");
```

the coded instructions (generated for this source) scan for both rules at once, observe (at the termination of the string of spaces or tabs) whether or not there is a newline character, and then executes the desired rule action. The first rule matches all strings of spaces or tabs at the end of lines, and the second rule matches all remaining strings of spaces or tabs.

The **lex** program generator can be used alone for simple transformations or for analysis and statistics gathering on a lexical level. The **lex** generator can also be used with a parser generator to perform the lexical analysis phase; **lex** and **yacc** are particularly compatible. The **lex** program recognizes only regular expressions; **yacc** writes parsers that accept a large class of context–free grammars but requires a low–level analyzer to recognize input tokens. 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 scan is shown in Figure 5H–2. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex**. The name **yylex** is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **lex** simplifies interfacing.

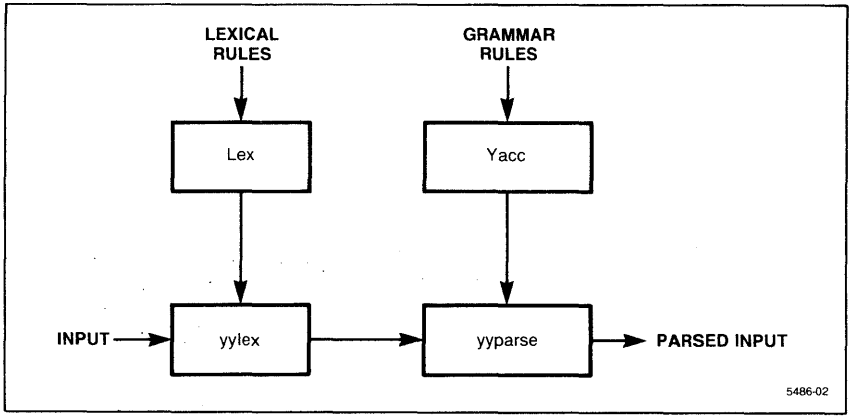


Figure 5H–2. Lex with Yacc.

The program written by **lex**, your fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch. The program interpreter directs the control flow. Opportunity is provided for you to insert either declarations or additional statements in the routine containing the actions or to add subroutines outside this action routine.

The **lex** program generator is not limited to a source that 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** recognizes *ab* and leaves the input pointer just before *cdefh*. Such backup is more costly than the processing of simpler languages.

Lex Source

The general format of **lex** source is:

{definitions}
%%
{rules}
%%
{user subroutines}

where the definitions and the user subroutines are often omitted. The first **%%** is required to mark the beginning of the rules, but the second **%%** is optional. The minimum **lex** program is:

```
%%
```

(no definitions, no rules), which translates into a program that copies the input to the output unchanged.

In the outline of **lex** programs shown above, the rules represent your control decisions. They are in a table containing:

- A left column with regular expressions
- A right column with actions and program fragments to be executed when the expressions are recognized.

Thus, an individual rule might be:

```
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 language 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 language 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 { }.

For example, suppose you want to change a number of words from British to American spelling. The **lex** rules, such as:

```
colour          printf("color" );
mechanize               printf("mechanize" );
petrol          printf("gas" );
```

would be a start. These rules are not sufficient since the word *petroleum* would become *gaseum*.

Lex Regular Expressions

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; 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, they should be indicated by quotation marks (" "). These marks indicate that whatever is contained between a pair of them is to be taken as text characters. Thus:

xyz"++"

matched 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 equivalent to the previous one. Thus, by quoting every nonalphanumeric character being used as a text character, you can avoid remembering the list of current operator characters.

An operator character may also be turned into a text character by preceding it with a backslash (\), as in:

xyz\+\+

which is another, less readable, equivalent of the previous expressions. Another use of quotation marks is to get a space into an expression; normally, as explained above, spaces or tabs end a rule. Any space character not contained within **[ ]** (see below) must be quoted. Several normal C language 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 use a backslash before a tab or backspace. Every character except space, tab, newline, and any of the operator characters is always a text character.

Character Classes

Classes of characters can be specified using the paired square brackets **[ ]**. The construction **[abc]** matches a single character which may be **a, b,** or **c**. Withing 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 lowercase letters, the digits, the angle brackets, and underline. Ranges can be given in either order. Using - between any pair of characters that are not both uppercase letters, both lowercase letters, or both digits is implementation-dependent and gets a warning message (for example, **[0-z]** in ASCII is many more characters than is in EBCDIC). If you want in include the character - in a character class, it should be first or last; thus:

[-+0-9]

matches all digits and the two signs.

In character classes, the caret operator (^) must appear as the first character after the left bracket to indicate that the resulting string is 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 that is not a letter. The \ character provides the usual escapes (as it did in the previous examples with \) within character class brackets.

Arbitrary Characters

To match almost any character, the dot or period operator character (.) is used for all characters except newline. Escaping into octal is possible although nonportable.

The following example matches all printable ASCII characters from octal 40 (space) to octal 176 (tilde):

[\40-\176]

Optional Expressions

The operator **?** precedes 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 +. For example:

a*

is any number of consecutive **a** character, including none. At the same time,

a+

is one or more instances of **a**. For example:

[a-z]+

is all strings of lowercase 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.

Alternations and Grouping

The operator ¦ indicates alternation. The following example matches either **ab** or **cd**:

(ab¦cd)

Note that parentheses are used for grouping, although they are not necessary on the outside level.

This next example says and works like the previous expression:

ab¦cd

Parentheses should be used for more complex expressions:

(ab¦cd+)?(ef)*

This matches such strings as **abefef, efefef, cedf**, or **cddd**; it does not match **abc, abcd**, or **abcdef**.

Context Sensitivity

The **lex** program recognizes a small amount of surrounding context. The two simplest operators for this are ^ and **$**. If the first character of an expression is ^, the expression is only matched at the beginning of a line (after a newline character or at the beginning of the input stream). This never conflicts with the other meaning of ^ (complementation of character classes), since that only applies within the **[ ]** operators. If the very last character is **$**, the expression is only matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character that 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 later. If a rule is only to be executed when the **lex** procedure 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 used only as the separator for **lex** source segments.

Lex Actions

When an expression is matched, **lex** executes the corresponding action. This section describes some features of **lex** that aid in writing actions. Note that there is a default action that 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. You may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule that merely copies can be omitted. Also, a character combination that is omitted from the rules and that 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 language null statement using the semicolon (;) as an action causes this result. A frequent rule is:

```
[\t\n]    ;
```

which causes the three spacing characters (space, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character (¦), the pipe, 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. The quotation marks around \n and \t are not required.

In more complex actions, you may often want to know the actual text that matched an expression such as **[a–z]+**. The **lex** program leaves this text in an external character array. Thus, to print the name found, a rule like:

```
[a–z]+   printf(" %s", yytext);
```

prints the string in *yytext[ ]*. The C language 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[ ]*. This places the matched string on the output. This action is so common that it is abbreviated as **ECHO**. It is the same as the preceding example:

```
[a–z]+  ECHO;
```

Since the default action is just to print the characters found, you 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 that is not desired. For example, if there is a rule that matches *read*, it normally matches the instances of *read* contained in *break* or *readjust*. To avoid this, a rule of the form **[a–z]+** is needed. This is explained later.

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, 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 onto the end of this input. Normally, the next input string would overwrite the current entry in **yytext**. Second, **yyless** (*n*) can 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 that 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 match is somewhat confusing, so that it might be preferable to write:

```
\"[^"]*  {
         if(yytext[yyleng-1] == '\\')
                 yymore( );
         else
                 ...normal user processing
         }
```

Lex, when faced with a string such as *"abc\def"*, first matches the five characters *"abc*; then the call to **yymore ( )** causes 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 labeled *normal processing.*

The function **yyless** might be used to reprocess text in various circumstances. Consider the C language problem of distinguishing the ambiguity of **=-a** (note that there is a space after the **a**). Suppose you want to treat this as **=-a** (without the space after the **a**), but also to print a message. A rule might be:

```
=-[a-zA-Z]   {
             printf("Operator (=-) 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 **=-** . (Note the space after the minus sign.) Alternatively, it might be desired to treat this as **=-a** (with a space after the **a**). To do this, just return the minus sign as well as the letter to the input.

The following example performs the other interpretation:

```
=-[a-zA-Z]  {
             printf("Operator (=-) ambiguous\n");
             yyless(yyleng-2);
             ...action for =...
             }
```

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 is required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of **=-3**, however,makes the following a still better rule:

=-/[^ \t\n]

In addition to these routines, **lex** also permits access to the I/O routines it uses. They are as follows:

1. **input( )** returns the next input character.
2. **output** (*c*) writes the character *c* on the output.
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 you 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 can be redefined to cause input or output to be transmitted to or from strange places including other programs or internal memory. The character set used must be consistent in all routines and a value of 0 returned by **input** must mean *end of file*. The relationship between **unput** and **input** must be retained or the **lex** look–ahead does not work. The **lex** program 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. The standard **lex** library imposes a 100–character limit on backup.

Another **lex** library routine that you may 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 wrap–up on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, you should provide a **yywrap** that 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, and so on, at the end of a program. Note that it is not possible to write a normal rule that 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.

Ambiguous Source Rules

The **lex** program 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 that matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules:

```
integer   keyword action . . . ;
[a–z]+            identifier action . . . ;
```

are to be given in that order. If the input is **integers**, it is taken as an identifier because:

```
"[a–z]+"
```

matches eight characters while **integer** matches only seven. If the input is **integer**, both rules match seven characters; and the keyword rule is selected because it was given first. Anything shorter (such as **int**) does 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 appear to be a good way of recognizing a string in single quotation marks. However, 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 matches the following:

'first'quoted string here,'second'

This is probably not what was wanted. A better rule is of the form:

```
'[^'\n]*'
```

which, on the above input, stops after **'first'**. The consequences of errors like this are reduced because the dot (.) operator does not match newline. Thus, expressions like **.*** stop on the current line.

NOTE

Do not try to defeat this with expressions like **[.\n]+** *or equivalents; the* **lex** *generated program tries to read the entire input file causing internal buffer overflows.*

Also note that **lex** normally is 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 dot **(.)** does not include newline. Since **she** includes **he**, **lex** normally does not recognize the instances of **he** included in **she** since once it has passed a **she**, those characters are gone.

Sometimes you might want 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 you really want to count the included instances of **he**. Use the following rules to change the previous example to accomplish the task.

```
she     {s++;REJECT;}
he      {h++;REJECT;}
\n      |
.       ;
```

After counting each expression, it is rejected; whenever appropriate, the other expression is then counted. In this example, you could note that **she** includes **he** but not vice versa and omit the **REJECT** action on **he**. In other cases, it is not possible to state which input characters are in both classes.

Consider these 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 **accd** 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.

The action **REJECT** does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found and **REJECT** executed, you must not have used **unput** to change the characters forthcoming from the input stream. This is the only restriction on your ability to manipulate the not-yet-processed input.

Lex Source Definitions

Let's review the format of the **lex** source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far, only the rules have been described. You need additional options to define variables for use in the program and for use by **lex**.

Remember, **lex** is generating 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 not part of a **lex** rule or action that begins with a space or tab is copied into the lex-generated program. Such source input prior to the first **%%** delimiter is 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** that contains the actions. This material must look like program fragments and should precede the first **lex** rule.

   Lines that begin with a space or tab and that 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 described previously. The delimiters are discarded. This format permits entering text-like preprocessor statements that must begin in column one or copying lines that do not look like programs.

3. Anything after the third **%%** delimiter, regardless of formats or whatever, 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 one 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 space 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, you can 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. 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 could be used in addition to the normal rule for integers. For example:

```
[0-9]+/"."EQ                printf("integer");
```

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 later.

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 in a file named *lex.yy.c*. The I/O library is defined in terms of the C language standard library.

On the UTek operating system, the library is accessed by the loader flag **-ll**. So an appropriate set of commends 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 the next topic, *Lex and Yacc*. Although the default **lex** I/O routines use the C language standard library, the **lex** automatons themselves do not do so. If private versions on **input, output**, and **unput** are given, the library is avoided.

Using Lex with Yacc

To use **lex** with **yacc**, observe that **lex** writes 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** calls **yylex**. In this case, each **lex** rule ends 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. If the grammar is to be named **good** and the lexical rules are to be named **better**, the UTek software command sequence could 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 that invokes the **yacc** parser. The generations of **lex** and **yacc** programs can be done in either order.

Examples

As a problem, consider copying an input file while adding 3 to every positive number divisible by 7. A suitable **lex** source program follows:

```
%%
        int k;
[0-9]+  {
        k=atoi(yytext);
        if(k%7==0)
           printf("%d´´ ,k+3);
        else
          printf("%d´´ ,k);
        }
```

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, *k* is incremented by 3 as it is written out. You might object that this program alters such input items as **49.63** or **X7**. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, 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 dot (.) or preceded by a letter are picked up by one of the last two rules and not changed. The **if–else** has been replaced by a C language 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 that determines 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 data while producing no output. At the end of the input, it prints the table. The final statement **return(1);** indicates that **lex** performs wrap–up. If **yywrap** returns zero (false), it implies that further input is available and the program is to continue reading and processing. Providing a **yywrap** (that never returns true (1)) causes an infinite loop.

Left Context Sensitivity

Sometimes you want 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 ^ (caret) operator, for example, is a prior–context operator recognizing the 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 part 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
- the possibility of making multiple lexical analyzers all run together

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed and that set a parameter to reflect the change. This may be a flag explicitly tested by your action code; this 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. The rule is only recognized when **lex** is in that start condition. The current start condition can be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, write several distinct lexical analyzers and switch from one to another as desired.

Consider the following problem: the input to the output, changing the word **magic** to **first** on every line that began with the letter **a**, changing **magic** to **second** on every line that 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 the job is with a flag. For example:

```
    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;
  }
}
```

To handle the same problem with start conditions, each start condition must be introduced to **lex** in the definition section with a line reading:

%Start *name1 name2*

where the conditions can be named in any order. The word **Start** can be abbreviated to **s** or **S**. The conditions can be referenced at the head of a rule with <angle brackets>:

<name1>expression

is a rule that 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** interpreter. A rule can be active in several start conditions. The following is a legal prefix:

<name1,name2,name3>

Any rule

The sam

```
%STAR
%%
^a
^b
^c
\n
<AA>m
<BB>m
<CC>m
```

where th
problem,

Cha

The prog
input, ou
routines i
character
value eq
host com
character
translate
section a
contains

{integ

Sun

The gene

```
{defin
%%
{rules}
%%
{user
```

Regular expressions in **lex** use the operators shown in Table 5H–1.

Table 5H–1
OPERATORS AND DESCRIPTIONS

OPERATOR	DESCRIPTION
x	the character *x*
"x"	"*x*", even if *x* is an operator
\x	"*x*", even if *x* is an operator
[xy]	the character *x* or *y*
[x-z]	the character *x*, *y*, or *z*
[^x]	any character but *x*
.	any character but newline
^x	*x* at the beginning of a line
<y>x	*x* when **lex** is in start condition *y*
x$	*x* at the end of a line
x?	optional *x*
x*	0,1,2,...instances of *x*
x+	1,2,3,...instances of *x*
x|y	*x* or *y*
(x)	*x*
x/y	*x* but only if followed by *y*
{xx}	translation of *xx* from the definitions section
x{m,n}	*m* through *n* occurrences of *x*

Problems and Bugs

There are pathological expressions that produce exponential grow
when converted to deterministic machines; fortunately, they are ra

REJECT does not rescan the input; instead it remembers the resu
scan. This means that if a rule with trailing context is found and **F**
you must not have used **unput** to change the characters coming fi
stream. This is the only restriction on your ability to manipulate th
processed input.

Consider the following problem: the input to the output, changing the word **magic** to **first** on every line that began with the letter **a**, changing **magic** to **second** on every line that 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 the job is with a flag. For example:

```
    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;
  }
}
```

To handle the same problem with start conditions, each start condition must be introduced to **lex** in the definition section with a line reading:

%Start *name1 name2*

where the conditions can be named in any order. The word **Start** can be abbreviated to **s** or **S**. The conditions can be referenced at the head of a rule with <angle brackets>:

<name1>expression

is a rule that 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** interpreter. A rule can be active in several start conditions. The following is a legal prefix:

<name1,name2,name3>

Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written as follows:

```
%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 your code.

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 used to return values in **yytext**. For internal use, a character is represented as a small integer that, 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 in the same form as the character constant **a**. If this interpretation is changed by providing I/O routines that translate the character, **lex** must be given a translation table that is in the definitions section and must be bracketed by lines containing only **%T**. The translation table contains lines in the form:

{*integer*} {*character string*}

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 translation.*
2. Included code in the form " code" (where the blank space preceding the code is necessary).
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 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:

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 can be continued on succeeding lines by using brackets to delimit it.

Regular expressions in **lex** use the operators shown in Table 5H–1.

Table 5H–1
OPERATORS AND DESCRIPTIONS

OPERATOR	DESCRIPTION
x	the character *x*
"x"	"*x*", even if *x* is an operator
\x	"*x*", even if *x* is an operator
[xy]	the character *x* or *y*
[x–z]	the character *x*, *y*, or *z*
[^x]	any character but *x*
.	any character but newline
^x	*x* at the beginning of a line
<y>x	*x* when **lex** is in start condition *y*
x$	*x* at the end of a line
x?	optional *x*
x*	0,1,2,...instances of *x*
x+	1,2,3,...instances of *x*
x|y	*x* or *y*
(x)	*x*
x/y	*x* but only if followed by *y*
{xx}	translation of *xx* from the definitions section
x{m,n}	*m* through *n* occurrences of *x*

Problems and Bugs

There are pathological expressions that produce exponential growth of the table when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found and **REJECT** executed, you must not have used **unput** to change the characters coming from the input stream. This is the only restriction on your ability to manipulate the not–yet–processed input.

The M4 Macro Processor

Introduction

A macro processor enhances a programming language by making it more palatable or more readable, or tailoring it to a particular application. The basic facility provided by any macro processor is replacement of text by other text.

M4 is a preprocessor for Ratfor, useful in those cases where macros without parameters are not powerful enough. **M4** is particularly suited for functional languages like FORTRAN, PL/I, and C, since macros are specified in a functional notation.

M4 is a suitable preprocessor for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string-processing functions.

M4 basically copies its input to its output. As the input is read, each alphanumeric *token* (string of letters and digits) is checked. If the token is the name of a macro, the name of the macro is replaced by the macro definition. The resulting macro definition is pushed back into the input to be rescanned. You can call macros with arguments, and the arguments are substituted into the macro definitions text before they are rescanned.

M4 has about twenty built-in macros that perform various useful operations; and the user can define new macros. Built-in macros, and user-defined macros work exactly the same way, except that some of the built-in macros affect the state of the process.

Invoking M4

To invoke **m4**, enter:

m4 [*filename1*] [*filename2*]

Each *filename* is processed in order. In this command, *filename* is optional. If you enter **m4** without arguments, or with the argument -, input for the command comes from the standard input. The processed text is written to the standard output.

Defining Macros

The most important built–in function of **m4** is **define**, and it defines new macros. This input defines the string *name* as *stuff*:

define(*name, stuff***)**

All subsequent occurrences of *name* are replaced by *stuff* . In this example, *name* consists of letters and numbers, but must begin with a letter (the underscore (_) counts as a letter). The string *stuff* is any text that contains balanced parentheses; it can stretch over multiple lines.

Following is an example of the **define** function:

```
define(N, 100)
...
if (i > N)
```

This example defines *N* to be 100, and uses it in a later **if** statement.

The left parenthesis immediately follows the word **define** , to signal that **define** has arguments. If a macro or built–in name is not followed immediately by a left parenthesis (, **m4** assumes it has no arguments.

A macro name is only recognized as such if it does not have a letter or a number either before or after it. For example:

```
define(N, 100)
...
if (NNN > 100)
```

The variable *NNN* is unrelated to the defined macro **N** .

You can take a macro definition and extend it to another level:

```
define(N, 100)
define(M, N)
```

This example defines both *M* and *N* to be 100.

What happens if *N* is redefined? Or, to say it another way, is *M* defined as *N* or as 100? In **m4**, the latter is true; *M* is 100, so even if *N* subsequently changes, *M* does not.

This happens because **m4** expands macro names into their defining text as soon as it possibly can. So when **define** (*N, 100*) is read, *N* is immediately replaced by 100. The second statement of the example is equivalent to:

define *(M, 100)*

If you wanted to avoid permanent definitions that extend to all levels, you can do two things. The first, is to interchange the order of the definitions:

define *(M, N)*
define *(N, 100)*

Now *M* is defined as *N* , so when you ask for *M* later, you get the *current* value of *N*, before *N* is redefined as 100.

Quoting

The more general solution to the problem of permanent redefinition delays expansion of the arguments of **define** by *quoting* them. Any text surrounded by single quotes (' and ') is not expanded immediately, but the quotes are stripped off. For example:

define(*N, 100***)**
define(*M, 'N'* **)**

In this example, the quotes around *N* are removed, and *M* is defined as the string *N* , not 100. In general, **m4** always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word **define** to display in the output, quote it in the input:

'define' = 1;

You can also redefine *N*.

The *N* in this second definition is immediately 100. So it is equivalent to this illegal **m4** statement. To redefine *N* , you must delay the evaluation by quoting:

define(*N***, 100)**
...
define('*N***', 200)**

If it is inconvenient to use single quotes, the quote characters can be changed with the built-in macro **changequote**:

changequote([,])

This changes makes the quote characters left and right brackets. You can restore the original characters by entering:

changequote

There are two additional built-in macros related to **define** . The **undefine** macro removes the definition of some other macro:

undefine(*'N'***)**

This removes the definition of *N* . You can also remove the definitions of built-in macros:

undefine(*'define'***)**

NOTE

Once you remove a built-in macro definition, you can never get it back.

The built-in macro **ifdef** lets you determine if a macro is currently defined. In particular, **m4** has pre-defined the names *unix* and *gcos* on the corresponding systems, so you can tell which one you're using:

ifdef('unix', 'define(wordsize,16)')
ifdef('gcos', 'define(wordsize,36)')

makes a definition appropriate for the particular machine.

The macro **ifdef** accepts three arguments; if the name is undefined, the value of **ifdef** is then the third argument. For example:

ifdef('unix', on UNIX, not on UNIX)

Arguments

So far we have discussed the simplest form of macro processing— replacing one string by another string. User-defined macros can also have arguments, so different invocations can have different results. In the replacement text for a macro (the second argument of **define**) any occurrence of **$***n* is replaced by the *n* th argument. So, the macro **bump**, defined as:

define(*bump, $1 = $1 + 1)*

generates code to increment its argument by 1:

bump(*x***)**

is equivalent to $x = x + 1$.

A macro can have as many arguments as you want, but only the first nine are accessible as *$1* to *$9*. (The macro name itself is *$0*, although that is less commonly used.) Arguments you do not enter are replaced by null strings, so you can define a macro **cat** that concatenates its arguments:

define(*cat, $1$2$3$4$5$6$7$8$9***)**

So:

cat(*x, y, z***)**

is equivalent to *xyz*.

The arguments *$4* through *$9* are null, because you did not enter them.

White space (blanks, tabs, or newlines) at the beginning of arguments are discarded. All other white space is retained. So this macro defines *a* to be *b c*:

define(*a, b c***)**

Arguments are separated by commas, but in **m4**, only the outside set of parentheses are counted properly. So a comma enclosed in secondary parentheses does not terminate an argument:

define(*a, (b,c)***)**

In this example there are only two arguments; the second is literally *(b,c)* . And of course you can include a single comma or parenthesis in an argument by quoting it.

Built–in Arithmetic Macro

M4 provides two built–in macros for doing arithmetic on integers. The simplest is **inca**, that increments its numeric argument by 1. So to define a variable as "one more than *N*", enter:

define(*N, 100***)**
define(*N1, 'incr(N)'***)**

Then *N1* is defined as one more than the current value of *N* .

The more general mechanism for arithmetic is a built–in macro called **eval**, that is, it performs arbitrary arithmetic on integers. It provides the following operators, in decreasing order of precedence:

1. unary + and –
2. ** or ^ (exponentiation)
3. * / % (modulus)
4. + –
5. == != < <= > >=
6. ! (not)
7. & or && (logical and)
8. ¦ or ¦¦ (logical or)

You can use parentheses to group operations. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and a false relation is 0. The precision in **eval** is 32 bits on UTek.

As a simple example, suppose you want *M* to be *2**N+1*. Enter:

define(*N, 3***)**
define(*M, 'eval(2**N+1)'***)**

It is a good idea to quote the defining text for a macro unless it is very simple.

File Manipulation

You can include a new file in the input at any time by using the built-in macro **include**:

include(*filename***)**

This replaces the **include** macro with the contents of *filename*. The contents of the file are often a set of definitions. The value of **include** (that is, its replacement text) is the contents of *filename*.

It is a fatal error if the file named in **include** cannot be accessed. Another macro, **sinclude** ("silent **include**"), says nothing and continues if it cannot access the file.

You can also divert the output of **m4** to temporary files during processing, and look at the collected material later. **M4** maintains nine of these diversions, numbered 1 through 9. Enter:

divert(*n***)**

All subsequent output is appended to the end of a temporary file, referred to as *n*. Stop this file diversion by entering **divert** or **divert(0)**.

Diverted text is normally displayed all at once at the end of processing, with the diversions output in numeric order. You can bring back diversions at any time to append them to the current diversion. Enter:

undivert

This command brings back all diversions in numeric order, and **undivert**, with arguments, brings back the selected diversions in the order specified. The **undivert** command discards the diverted material, as does diverting to a number other than 0 to 9.

The value of **undivert** is *not* the diverted material. Furthermore, the diverted material is *not* rescanned for macros.

The built-in macro **divnum** returns the number of the currently active diversion. This is zero during normal processing.

System Command

You can run any UTek program with the **syscmd** built-in macro. For example:

syscmd(*date***)**

This runs the UTek **date** command.

To facilitate making unique filenames, the built-in macro **maketemp** is provided. It works like the UTek command **mktemp**: a string of XXXXX in the argument is replaced by the process id of the current process.

Conditionals

There is a built-in macro called **ifelse** that enables you to perform arbitrary conditional testing. In the simplest form, the following example compares the two strings *a* and *b*:

ifelse(*a, b, c, d***)**

If *a* and *b* are identical, **ifelse** returns the string *c* ; otherwise it returns *d* . So you might define a macro called **compare** that compares two strings and returns *yes* if they are the same, or *no* if they are different. For example:

define(*compare, 'ifelse($1, $2, yes, no)'***)**

Note the quotes; they prevent premature evaluation of **ifelse** .

If the fourth argument is missing, it is treated as empty.

The macro **ifelse** can have any number of arguments, so it provides a limited form of multi-choice decision capability:

ifelse(*a, b, c, d, e, f, g***)**

If the string *a* matches the string *b* , the result is *c* . Otherwise, if *d* is the same as *e*, the result is *f* . Otherwise the result is *g* . If the final argument is omitted, the result is null.

String Manipulation

The built-in macro **len** returns the length of the string that makes up its argument. For example:

len(*abcdef***)**

This returns a value of 6.

The built-in macro **substr** can be used to produce substrings of strings. It has the general form:

substr(*s, i, n***)**

This returns the substring of *s* that starts at the *i* th position and is *n* characters long. If *n* is omitted, the rest of the string is returned.

The macro **index** (*s1, s2*) returns the index (position) in string *s1* where the string *s2* occurs, or —1 if *s2* is not present in *s1*. As with **substr** , the origin for strings is 0.

The built–in macro **translit** performs character transliteration:

translit(*s, f, t*)

This modifies *s* by replacing any character found in *f* by the corresponding character of *t*. For example:

translit(*s, aeiou, 12345*)

This replaces the vowels by the corresponding digits. If *t* is shorter than *f* , characters without an entry in *t* are deleted; as a limiting case, if *t* is not present at all, characters from *f* are deleted from *s* . So

translit(*s, aeiou*)

This deletes vowels from *s* .

There is also a built–in macro called **dnl** that deletes all characters following it including the next newline character. It is useful for throwing away empty lines that otherwise tend to clutter up **m4** output. For example, if you enter:

define(*N, 100*)
define(*M, 200*)
define(*L, 300*)

In this example the newline at the end of each line is not part of the definition, but it is copied into the output, where you might not want it. If you add **dnl** to each of these lines, the newline characters disappear.

Another way to achieve this follows:

```
divert(-1)
        define(...)
        ...
divert
```

Printing

The built–in macro **errprint** writes its arguments out on the standard error file. So you can enter:

errprint('fatal error')

The **dumpdef** macro is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get all the definitions; otherwise you get the defined terms you name as arguments. Be sure to quote the defined terms you use as arguments.

The Programming Language EFL

Introduction

Purpose

EFL is a general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. You can translate EFL programs into efficient FORTRAN code, so you can take advantage of the availability of FORTRAN, the valuable software libraries in FORTRAN, and the portability of a standardized language. EFL works especially well for numeric programs. EFL lets you express complicated ideas in a comprehensible way, yet retain the power of the FORTRAN environment.

History

EFL is a descendant of Ratfor. The current EFL compiler is written in C. It is much more than a simple preprocessor: it tries to diagnose syntax errors, to provide readable FORTRAN output, and to avoid a number of restrictions.

Character Set

The following characters are legal in an EFL program:

letters	a b c d e f g h i j k l m
	n o p q r s t u v w x y z
digits	0 1 2 3 4 5 6 7 8 9
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )

other	, ; : . + - * / = < > & ~ ¦ $

Upper- or lower-case letters are ignored except within strings, so **a** and **A** are treated as the same character. All of the examples below are printed in lower case. An exclamation mark (**!**) may be used in place of a tilde (~). You can use square brackets (**[ ]**) in place of braces ({ }).

Lines

EFL is a line-oriented language. Except in special cases discussed below, the end of a line marks the end of a token and the end of a statement. You can use the trailing portion of a line for a comment. EFL can divert input from one source file to another, so you can replace a single line in the program with a number of lines from the other file. Diagnostic messages are labeled with the line number of the file where they are detected.

White Space

Any sequence of one or more spaces or tab characters acts as a single space, except outside of a character string or comment. Such a space terminates a token.

Comments

A comment can appear at the end of any line. It is introduced by a sharp character (#), and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) EFL discards the sharp and succeeding characters on the line; they have no effect on execution. A blank line is also a comment.

Include Files

You can insert the contents of a file somewhere in the source text, by referencing it in a line. For example:

```
include joe
```

Do not put a statement or comment following an **include** line. The **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. You can nest **include**s at least ten deep.

Continuation

You can continue lines by using the underscore character (_). If the last character of a line is an underscore, the EFL ignores the end of a line and the initial blanks on the next line. Underscores are ignored in other contexts (except inside of quoted strings). So:

```
1_000_000_
  000
```

equals 109.

EFL also contains rules for continuing lines automatically: the end of line is ignored when it is obvious that the statement is not complete. Specifically, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. Some compound statements are continued automatically; these instances are noted in the sections on executable statements.

Multiple Statements on a Line

A semicolon (;) terminates the current statement, so you can write more than one statement on a line. A line consisting of one or more semicolons forms a null statement.

Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless an underscore continues the line.

Identifiers

An identifier is a letter or a letter followed by letters or digits. Table 5J—1 contains the reserved words that have special meaning in EFL. They are discussed later.

Table 5J-1
RESERVED WORDS WITH SPECIAL MEANING

array	exit	precision
automatic	external	procedure
break	false	read
call	field	readbin
case	for	real
character	function	repeat
common	go	return
complex	goto	select
continue	if	short
debug	implicit	sizeof
default	include	static
define	initial	struct
dimension	integer	subroutine
do	internal	true
double	lengthof	until
doubleprecision	logical	value
else	long	while
end	next	write
equivalence	option	writebin

Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single–quote marks ('), it may contain double quote marks ("), and vice versa. You cannot break a quoted string across a line boundary.

Integer Constants

An integer constant is a sequence of one or more digits. For example:

0
57
123456

Floating Point Constants

A floating point constant contains a dot and/or an *exponent field*. An exponent field consists of the letters "d" or "e" followed by an integer constant with an optional sign. If **I** and **J** are integer constants and **E** is an exponent field, then a floating constant has one of the following forms:

.I
I.
I.J
IE
I.E
.IE
I.JE

Punctuation

Certain characters group or separate objects in the language. These are

parentheses ( )
braces { }
comma ,
semicolon ;
colon :
end–of–line

The end–of–line is a token when the line is neither blank nor continued.

Operators

The EFL operators are written as sequences of one or more special characters.

```
+   -   *   /   **
<   <=   >   >=   ==   ~=
&&   ||   &   |
+=   -=   *=   /=   **=
&&=   ||=   &=   |=
->   .   $
```

A dot (.) is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode in which some of the operators can be represented by a string consisting of a dot, an identifier, and a dot.

Macros

EFL has a simple macro substitution facility. You can define an identifier to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement. For example:

define *count* **n += 1**

Any time the name *count* appears in the program, it is replaced by the statement

n += 1

You must put a macro definition alone on a line; the form is

define *name*

Trailing comments are part of the string.

Program Form

Files

A *file* is a sequence of lines. EFL compiles each file as a single unit. The file can contain one or more procedures. Declarations and options outside of a procedure affect the succeeding procedures in that file.

Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed later.

Blocks

You can form statements into groups inside a procedure. To describe the scope of names, the ideas of *block* and of *nesting level* are introduced. The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations in the file are also at nesting level zero. The text immediately following a **procedure** statement is at nesting level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. An **end** statement marks the end of the procedure, level 1, and the return to level 0. A variable or macro name that is defined at a particular level is defined throughout that block. It is also defined in all deeper levels, if the name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#   block 0
procedure george
real x
x = 2
 . . .
if(x > 2)
        {       # new block
        integer x       # a different variable
        do x = 1,7
                write(,x)
        . . .
        }       # end of block
end     # end of procedure, return to block 0
```

Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define
Procedure
End
Declarative
Executable

All these kinds of statements are discussed later in this section. Each procedure begins with a **procedure** statements and finishes with an **end** statement. Declarations describe types and values of variables and procedures. Executable statements cause EFL to take specific actions. A block is an example of an executable statement; it is made up of declarative and executable statements.

Labels

An executable statement can have a *label* that is used in a branch statement. A label is an identifier followed by a colon, as in

```
        read(, x)
        if(x < 3) goto error
        . . .
error:  fatal("bad input")
```

Data Types and Variables

EFL supports a small number of basic (scalar) types. You can define objects made up of variables of basic type; you can then define other aggregates in terms of previously–defined aggregates.

Basic Types

The basic types are:

logical
integer
field(*m*:*n*)
real
complex
long real
character(*n*)

A logical quantity can acquire the values **true** and **false**. An integer can acquire any whole number value in some machine–dependent range. A field quantity is an integer restricted to a particular closed interval ([*m*:*n*]). A real quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers. A complex quantity is an approximation to a complex number. It is represented as a pair of reals. A character quantity is a fixed–length string of *n* characters.

Constants

A notation exists for a constant of each basic type.

A logical type can acquire the two values:

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign. For example:

17
–94
+6
0

A long real (double precision) constant is a floating point constant containing an exponent field that begins with the letter **d**. A real (single precision) constant is any other floating point constant. You can precede real or long real constant with a plus or minus sign. The following are valid real constants:

17.3
–.4
7.9e–6 (= 7.9 x 10 –6)
14e9 (= 1.4 x 1010)

The following are valid long real constants:

7.9d–6 (= 7.9 x 10 6)
5d3

A character constant is a quoted string.

Variables

A variable is a quantity with a name and a location. At any particular time the variable can also have a value. (A variable is *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has some of the following attributes.

Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent.

Scope of Names

The names of common areas are global, as are procedure names. You can use these names anywhere in the program. All other names are local to the block where they are declared.

Precision

Floating point variables are either of normal or long precision. You can state this attribute independently of the basic type.

Arrays

You can declare rectangular arrays of values of the same type. (The arrays can have any dimension.) The index set is always a cross–product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or common. A formal argument array can have intervals that have the same length as one of the other formal arguments. An element of an array is denoted by the array name, followed by a parenthesized list of integer values. The list is separated by commas and enclosed in parentheses. Each value must lie within the corresponding interval. The intervals can include negative numbers. EFL can pass entire arrays as procedure arguments or in input/output lists, or initialize them. All other array references must be to individual elements.

Structures

You can define new types that are made up of elements of other types. The compound object is known as a *structure*; its constituents are called *members* of the structure. You can give the structure a name. The name acts as a type name in the remaining statements, within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they can be arrays of such objects. EFL can pass entire structures to procedures or use them in input/output lists. It also references individual elements of structures. The uses of structures are detailed below. The following structure might represent a symbol table:

```
struct tableentry
          {
          character(8) name
          integer hashvalue
          integer numberofelements
          field(0:1) initialized, used, set
          field(0:10) type
          }
```

Expressions

Expressions are syntactic forms that yield a value. An expression can have any of the following forms, recursively applied:

primary
(expression)
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. These operators are described later.

```
->  .
**
*  /       unary +  -  ++  --
+  -
<  <=  >  >=  ==  ~=
&  &&
|  ||
$
=  +=  -=  *=  /=  **=    &=  |=  &&=  ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

Primaries

Primaries are the basic elements of expressions, as follows:

Constants

Constants are described earlier.

Variables

Scalar variable names are primaries. They can appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) can only appear as procedure arguments and in input/output lists.

Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, with one integer value for each declared dimension:

```
a(5)
b(6,-3,4)
```

Structure Members

A structure name, followed by a dot and the name of a member of that structure, reference that element. If that element is itself a structure, the reference can be further qualified.

a.b
x(3).y(4).z(5)

Procedure Invocations

You can invoke a procedure by an expression of one of the forms

procedurename **(** **)**
procedurename **(** *expression* **)**
procedurename **(** *expression-1*, . . . ,
expression-n **)**

The *procedurename* is the name of a variable declared external, or the name of a function known to the EFL compiler, or the actual name of a procedure from a **procedure** statement. If a *procedurename* is declared external and is an argument of the current procedure, it is associated with the procedure name passed as actual argument. Otherwise it is the actual name of a procedure. Each *expression* is called an *actual argument*. Examples of procedure invocations are:

f(x)
work( )
g(x, y+3, 'xx')

When EFL performs a procedure invocation, it first evaluates each of the actual argument expressions. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, EFL lets the called procedure use the corresponding formal argument as the left side of an assignment or in an input list; otherwise the actual argument can only use the value. After the formal and actual arguments are associated, control passes to the first executable statement of the procedure. When that procedure executes a **return** statement, or when control reaches the **end** statement of that procedure, EFL makes available the function value. The function value is the value of the procedure invocation. The attributes of the *procedurename* that are declared or implied in the calling procedure, determine the type of the value. The type of the value must agree with the attributes the procedure declares for the function. In the special case of a generic function, the type of the result is also affected by the type of the argument.

Input/Output Expressions

You can use the EFL input/output syntactic forms as integer primaries. If an error occurs during input or output, the integer primaries have a nonzero value.

Coercions

You can convert an expression of one precision or type to another. Use an expression of the form

attributes **(** *expression* **)**

Attributes are precision and basic types. White spaces separate special attributes. You can coerce an arithmetic value of one type to any other arithmetic type; a character expression of one length can be coerced to a character expression of another length; logical expressions cannot be coerced to a nonlogical type. As a special case, you can construct a quantity of complex or long complex type from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = **5**
long real(5) = **5.0d0**
complex(5,3) = **5+3i**

EFL does most conversions implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are useful when you must convert the type of an actual argument to match the type of the corresponding formal parameter in a procedure call.

Sizes

There is a notation that gives the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside* **)**
sizeof (*attributes* **)**

In the first case, *leftside* stands for a variable, array, array element, or structure member.

Sizeof is an integer that gives the size in arbitrary units. If you need the size in specific units, compute it by division:

sizeof(*x***) / sizeof(***integer***)**

This division gives the size of the variable x in integer words.

The distance between consecutive elements of an array cannot equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives larger distance, again in arbitrary units. The form is:

lengthof (*leftside* **)**
lengthof (*attributes* **)**

Parentheses

An expression in parentheses is itself an expression. It must be evaluated before the expression it is part of is evaluated.

Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

Arithmetic

Unary + has no effect. A unary – yields the negative of its operand. The value of either expression results from the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

Logical

The only logical unary operator is complement (~). These equations define the complement operator:

~ **true** = **false**
~ **false** = **true**

Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, strings of two or three special characters denote some operators. All binary operators except exponentiation are left associative.

Arithmetic

The binary arithmetic operators are

+	addition
–	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = a (b c) The operations have the conventional meanings:

8+2 = 10, 8–2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 82 = 64.

The type of its operands determines the type of the result of a binary operation A operator B. This is shown in Tables 5J–2 and 5J–3.

Table 5J-2
RELATION BETWEEN BINARY OPERATION A AND B

	TYPE OF B	
	integer	*real*
TYPE OF A		
integer	*integer*	*real*
real	*real*	*real*
long real	*long real*	*long real*
complex	*complex*	*complex*
long complex	*long complex*	*long complex*

Table 5J-3
RELATION BETWEEN BINARY OPERATION A AND B

	TYPE OF B		
	long real	**complex**	*long complex*
TYPE OF A			
integer	*long real*	*complex*	*long complex*
real	*long real*	*complex*	*long complex*
long real	*long real*	*long complex*	*long complex*
complex	*long complex*	*complex*	*long complex*
long complex	*long complex*	*long complex*	*long complex*

If the type of the operand differs from the type of the result, EFL does the calculation as if the operand were first coerced to the type of the result. If both operands are integers, the result an integer type, and it is computed exactly. (Quotients are truncated toward zero, so 8/3 = 2.)

Logical

The two binary logical operations in EFL, *and* and *or*, are defined by the truth tables:

Table 5J–4
TRUTH TABLES

A	**B**	**A** *and* **B**	*A or B*
false	*false*	*false*	*false*
false	*true*	*false*	*true*
true	*false*	*false*	*true*
true	*true*	*true*	*true*

Each of these operators comes in two forms. In one form, the order of evaluation is specified. EFL evaluates the expression

a && b

by evaluating **a** first; if **a** is false then the expression is false and **b** is not evaluated. If **a** is true, the the expression has the value of **b**. EFL evaluates the expression

a ¦ ¦ b

by evaluating **a** first; if **a** is true then the expression is true and **b** is not evaluated. If **a** if false the expression has the value of **b**. The other forms of the operators (**&** for *and* and **¦** for *or*) do not imply an order of evaluation. With the **&** and **¦** operators, the compiler can speed up the code by evaluating the operands in any order.

Relational Operators

There are six relations between arithmetic quantities. These operators are not associative. See Table 5J–5.

Table 5J–5
RELATION BETWEEN ARITHMETIC QUANTITIES

EFL Operator		**Meaning**
<	<	*less than*
<=	≤	*less than or equal to*
= =	=	*equal to*
~=	!=	*not equal to*
>	>	*greater than*
>=	≥	*greater than or equal*

Since complex numbers are not ordered, the only relational operators that can have complex operands are == and ~= . The character–collating sequence is not defined.

Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = expression

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side and stores that value in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

An assignment operator corresponds to each binary arithmetic and logical operator. In each case, *a op = b* is equivalent to *a = a op b*. (The operator and equal sign must not be separated by blanks.) So, *n*+=**2** adds 2 to *n*. The location of the left side is evaluated only once.

Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures:

leftside –> structurename

This expression is a structure with the shape implied by *structurename*, but it starts at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted using the dot operator. Thus,

place(i) –> st.elt

refers to the *elt* member of the **st** structure starting at the *ith* element of the array **place**.

Repetition Operator

Inside of a list, an element of the form

integer-constant-expression **$** *constant-expression*

is equivalent to *expression* a number of times equal to the first expression. For example:

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

Constant Expressions

If an expression is built of operators (other than functions) and constants, the value of the expression is a constant. You can use the value of the expression anywhere a constant is required.

Declarations

The declarations statement describes the meaning, shape, and size of named objects in the EFL language.

Syntax

A declaration statement is made up of attributes and variables. Declaration statements have two possible forms:

attributes variable-list
attributes { *declarations* }

In the first example, each name in the *variable-list* has the attributes you specify. This is also true for the second example. You can put a variable name in more than one variable list, so long as its attributes do not contradict each other. You can specify an initial value to accompany each name of a nonargument variable. The *declarations* inside braces consist of one or more declaration statements. Examples of declarations are:

```
integer k=2
long real b(7,3)
common(cname)
      {
      integer i
      long real array(5,0:3) x, y
      character(7) ch
      }
```

Attributes

Basic Types

Basic types in declarations include:

logical
integer
field(*m:n***)**
character(*k***)**
real
complex

The quantities *k*, *m*, and *n* are integer constant expressions with the properties $k>0$ and $n>m$.

Arrays

You can declare dimensionality by an **array** attribute of the form:

array(b1 , . . . , bn)

Each of the **b***i* can be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to *n*, where *n* is the number of bounds. All the integer expressions must be constants. EFL permits integer expressions that are not constants if all of the variables associated with an array declarator are formal arguments of the procedure. In this case, each bound must have the property that **upper – lower + 1** is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it recognizes important cases such as **(0:n–1)**. You can mark the upper bound for the last dimension (**b***n*) with an asterisk (*) if the size of the array is unknown. The following are legal **array** attributes:

array(5)
array(5, 1:5, –3:0)
array(5, *)
array(0:*m***–1,** *m***)**

Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

Structname is optional; if it is present, it acts like the name of a type in the rest of its scope. Each name that appears inside the *declaration statements* is a member of the structure. It has a special meaning when you use it to qualify any variable declared with the structure type. A name can be a member of any number of structures, and may also be the name of an ordinary variable, because you use a structure member name only in contexts where the parent type is known. The following are valid structure attributes:

```
struct xx
      {
      integer a, b
      real x(5)
      }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx** 's and a character string.

Precision

You can declare variables of long floating point (real or complex) type to ensure higher precision than ordinary floating point variables. The default precision is short.

Common

Certain objects called *common areas* have external scope. You can reference them with any procedure that has a declaration for the name using the attribute:

common (*commonareaname* **)**

All the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in different procedures must put variables in the same order, and use the same types, precision, and shapes. These variables do not have to use the same names.

External

If you use a name as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is passed as an argument, you must declare it in a statement of the form:

external [*name* **]**

If *name* has the external attribute and is a formal argument of the procedure, it is associated with a procedure identifier, that is passed as an actual argument at each call. If the name is not a formal argument, it is the actual name of a procedure. The name corresponds to that in the procedure statement.

Variable List

A variable list in a declaration consists of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification has the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign can be a parenthesized list of constant expressions, or repeated elements or lists. The total number of elements in the list must not exceed the number of elements of the array. The elements of the array are filled in column-major order.

The Initial Statement

You can specify an initial value for a simple variable, array, array element, or member of a structure. Use a statement of the form

initial [*var* = *val* **]**

Var can be a variable name, array element specification, or member of structure. *Val* follows the same rules as other declaration statements for an initial value specification.

Executable Statements

Every EFL program contains executable statements. Statements are frequently a composite of other statements. Blocks are the most obvious case, but many other forms are made up of statements.

To make EFL programs more readable, you can break some of the statement forms without an explicit continuation. A pair of brackets ([ ]) in the syntax represents a point where the end of a line is ignored.

Expression Statements

Subroutine Call

A subroutine call is a procedure invocation that returns no value. Such an invocation is a statement. Examples are

```
work(in, out)
run( ¦ )
```

Input/output statements resemble procedure invocations, but do not yield a value. If an error occurs, the program stops.

Assignment Statements

An expression that is a simple assignment (for example, =) or a compound assignment (for example, +=) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, and executable statements, followed by a right brace. You can use a block anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is:

```
{
integer i     # this variable is unknown outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

If Statement

The simplest of the test statements is the **if** statement. It has the form:

if (*logical-expression* **) [ ]** *statement*

The *logical expression* is evaluated; if it is true, *statement* is executed.

If-Else

A more general statement has the form:

if (*logical-expression* **) [ ]** *statement-1* **[ ] else [ ]**
statement-2

If *logical-expression* is true, *statement-1* is executed. Otherwise *statement-2* is executed. The next statement can itself be an **if–else**, so a completely nested test sequence is possible:

```
if(x<y)
    if(a<b)
        k = 1
    else
        k = 2
else
    if(a<b)
        m = 1
    else
        m = 2
```

An **else** applies to the nearest preceding un–**else**d **if**. A more common use is as a sequential test:

```
if(x==1)
    k = 1
else if(x==3  ¦  x==5)
    k = 2
else
    k = 3
```

Select Statement

You can succinctly state a multi–way test on the value of a quantity as a **select** statement, with the general form:

select(*expression* **) [ ]** *block*

Inside the block two special types of labels are recognized. A prefix of the following form marks the statement to which control is passed if the *expression* in the select statement has a value equal to one of the case constants:

case [*constant*] :

If *expression* equals none of these constants, but a label **default** is present in the select statement, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until it encounters the next **case** or **default**. The previous **else-if** example is better written as:

```
select(x)
    {
    case 1:
        k = 1
    case 3,5:
        k = 2
    default:
        k = 3
    }
```

Note that control does not fall through to the next case.

Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest form (**while**) is theoretically sufficient. But it is very convenient to have the loops of a general form available, since each illustrates a kind of control that programmer's frequently use.

While Statement

The **while** statement takes the following form:

while (*logical-expression*) [ ] *statement*

The expression is evaluated; if it is true, the statement executes, and the test is performed again. If the expression is false, execution proceeds to the next statement.

For Statement

The **for** statement is a more elaborate looping construct. It takes the following form:

for (*initial-statement* , [ ] *logical-expression* ,
[ ] *interation-statement*) [ ] *body-statement*

Except for the behavior of the **next** statement, this construct is equivalent to:

initial-statement
while (*logical-expression*)
{
body-statement
iteration-statement
}

This form is useful for general arithmetic iterations, and for various pointer operations. The sum of the integers from 1 to 100 can be computed by the fragment:

```
n = 0
for(i = 1, i <= 100, i += 1)
        n += i
```

Alternatively, you could use the single statement:

```
for( { n = 0 ; i = 1 } ,
i<=100, { n += i ; ++i } )
        ;
```

Note that the body of the **for** loop is a null statement in this case.

Repeat Statement

The **repeat** statement takes the following form:

repeat [ ] *statement*

This command executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is necessary to stop the loop.

Repeat...Until Statement

The **while** loop performs a test before each iteration. The following statement executes the *statement*, then evaluates *logical-expression*:

repeat [ ] *statement* **[ ] until (** *logical-expression* **)**

In this command, if *logical-expression* is true, the loop is complete; otherwise control returns to the *statement*. Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that is not paired with an **until**. In practice, this is a less frequently used looping construct.

Do Loops

The simple arithmetic progression is common in numerical programs. EFL has a special loop form for ranging over an ascending arithmetic sequence:

do *variable* = *expression-1, expression-2, expression-3*
    statement

The variable is first given the value *expression-1*. The statement is executed, then *expression-3* is added to the variable. The loop repeats until the variable exceeds *expression-2*. If you omit *expression-3* and the preceding comma, the assumed increment is 1. The previous loop is equivalent to:

t2 = *expression-2*
t3 = *expression-3*
for(*variable* = *expression-1* , *variable* <= **t2** , *variable* += **t3)**
    statement

(The compiler translates EFL **do** statements into FORTRAN **DO** statements, that are compiled into excellent code.) You cannot change the **do** *variable* inside the loop, and *expression-1* must not exceed *expression-2*. For example, you can compute the sum of the first hundred positive integers as follows:

```
n = 0
do i = 1, 100
        n += i
```

Branch Statements

Usually you can avoid branch statements in programs by using the **loop** and **test** constructs, but in some programs they are very useful.

Goto Statement

The most general, and most error prone, branching statement is the simple, unconditional **goto** statement:

goto *label*

After this statement executes, the next statement following the *label* is executed. Inside a **select** statement you can use the **case** labels of that block as *label*. For example:

```
select(k)
        {
        case 1:
            error(7)

        case 2:
            k = 2
            goto case 4

        case 3:
            k = 5
            goto case 4

        case 4:
            fixup(k)
            goto default

        default:
            prmsg("ouch")
        }
```

(If you nest two **select** statements, the case labels of the outer **select** are not accessible from the inner one.)

Break Statement

The more frequently used **break** transfers control to the statement following the current **select** or **loop** construct. A **repeat** loop usually requires this kind of statement:

```
repeat
    {
    do a computation
    if ¦ (finished)
        break
    }
```

More general forms let you control a branch using more than one construct:

```
break 3
```

This form transfers control to the statement following the third **loop** and/or **select** surrounding the statement. You can specify what type of construct (**for**, **while**, **repeat**, **do**, or **select**) to count. This statement breaks out of the first surrounding **while** statement:

break while

Either of the following statements transfers to the statement after the third enclosing for **loop**:

break 3 for
break for 3

Next Statement

The **next** statement makes the first surrounding loop statement go on to the next iteration. The next operation performed is the test of a **while**, the *iteration-statement* of a **for**, the body of a **repeat**, the test of a **repeat . . . until**, or the increment of a **do**. Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

Return

The last statement of a procedure returns control to the caller. If you want to return control to the caller from any other point in the procedure, use the **return** statement:

return

Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression*)

Input/Output Statements

The input/output part of EFL very strongly reflects the facilities of FORTRAN. EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). You can use these forms as a primary with an integer value or as a statement. If an exception occurs when you use one of these forms as a statement, the result is undefined, but is usually treated as a fatal error. If you use them in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end–of–file and a positive value an error.

Input/Output Units

Each I/O statement refers to a *unit*, identified by a small positive integer. EFL defines two special units, the *standard input unit* and the *standard output unit*. These particular units are assumed if no unit is specified in an I/O transmission statement.

Information about the unit is organized into *records*. These records can be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file*.

Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form:

writebin(*unit , binary-output-list* **)**
readbin(*unit , binary-input-list* **)**

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see the following topic, *Iolists*) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers where each of the expressions is a variable name, array element, or structure member.

Formatted Input/Output

The **read** and **write** statements transmit data as lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, provided, implicitly or explicitly, in the statement. The syntax of the statements is:

write(*unit, formatted-output-list* **)**
read(*unit, formatted-input-list* **)**

The lists have the same form as binary I/O, except that the lists can include format specifications. If the *unit* is omitted, the standard input or output unit is used.

Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form:

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* can also have these forms:

ioexpression :*format-specifier*
: *format-specifier*

A *do-specification* has the syntax of a **do** statement, and it has a similar effect. The values in the braces are transmitted repeatedly until the **do** execution completes.

Formats

The following are *format-specifiers* recognized by EFL. The quantities *w*, *d*, and *k* must be integer constant expressions.

i(*w*)	integer with *w* digits
f(*w*,*d*)	floating point number of *w* characters, *d* of them to the right of the decimal point.
e(*w*,*d*)	floating point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(*w*)	logical field of width *w* characters, the first of which is **true** or **false** (the rest are blank on output, ignored on input)
c	character string of width equal to the length of piece of data
c(*w*)	character string of width *w*
s(*k*)	skip *k* lines
x(*k*)	skip *k* spaces
" ... "	use the characters inside the string as a FORTRAN format

If no format is specified for an item in a formatted input/output statement, EFL chooses a default form.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

Manipulation Statements

These three input/output statements look like ordinary procedure calls:

backspace(*unit*)
rewind(*unit*)
endfile(*unit*)

But you can use them either as statements, or as integer expressions that returns a nonzero value if an error is detected. The **backspace** statement causes the specified unit to back up, so that the next **read** reads the previous record again, and the next write over-writes it. The **rewind** statement moves the device to its beginning, so that the next input statement reads the first record. The **endfile** statement marks the file so that the record most recently written is the last record on the file, and any attempt to read past that point is an error.

Procedures

Procedures are the basic unit of an EFL program, providing the means of segmenting a program into parts that can be named and complied separately.

Procedure Statement

Each procedure begins with a statement of one of these forms, preceded by **procedure** on a line by itself:

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* **( )**
attributes **procedure** *procedurename* **(** [*name*] **)**

The first form specifies the main procedure, where execution begins. In the other two forms, the *attributes* can specify precision and type, or you can omit them. You can declare the precision and type of the procedure in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and it cannot return a value. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside parentheses, as in the last form, is called a *formal argument* of the procedure.

End Statement

Each procedure terminates with the following statement:

end

Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument. The procedure can reference the values in the object, and assign values to it. Otherwise, the value of the actual argument is associated with the formal argument, but the procedure may not try to change the value of that formal argument.

If the procedure changes the value of one of the arguments, the corresponding actual argument cannot be associated with another formal argument, or with a common element that is referenced in the procedure.

Execution and Return Values

After actual and formal arguments are associated, control passes to the first executable statement of the procedure. Control returns to the invoker when the **end** statement of the procedure is reached, or when a **return** statement is executed. If the procedure is a function (has a declared type), and a return value is executed, the value is coerced to the correct type and precision and returned.

Known Functions

EFL knows about a number of functions, and it is not necessary to declare them. The compiler knows the types of these functions. Some of them are *generic*, that is, they name a family of functions that differ in the types of their arguments and return values. The compiler chooses what element of the set to invoke based upon the attributes of the actual arguments.

Minimum and Maximum Functions

The generic functions are **min** and **max**. The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that can take different numbers of arguments in different calls. If any of the arguments are long real, the result is long real. Otherwise, if any of the arguments are real, the result is real; otherwise all the arguments and the result must be integer. For example:

```
min(5, x, -3.20)
max(i, z)
```

Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments, the type of the result is identical to the type of the argument; for complex arguments the type of the result is a real of the same precision.

Elementary Functions

The following generic functions take arguments of real, long real, or complex type and return a result of the same type:

sin	sine function
cos	cosine function
exp	exponential function (*ex*)

log	natural (base *e*) logarithm
log10	common (base 10) logarithm
sqrt	square root function

In addition, the following functions accept only real or long real arguments:

atan	atan(x) = tan−1 x
atan2	atan2(x,y) = tan−1 x over y

Other Generic Functions

The **sign** functions take two arguments of identical type; **sign** (x,y) = **sgn**(y) $|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

Converting Older Programs

Certain facilities are included in the EFL language to ease the conversion of old FORTRAN or Ratfor programs to EFL.

Escape Lines

To make use of nonstandard features of the local FORTRAN compiler, you occasionally need to pass a particular line through to the EFL compiler output. A line that begins with a percent sign (**%**) is copied through to the output without the percent sign. Inside procedure, each escape line is treated as an executable statement. If a sequence of lines constitutes a continued FORTRAN statement, they are enclosed in braces.

Call Statement

A subroutine call may be preceded by the keyword **call**:

```
call joe
call work(17)
```

Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Table 5J–6
KEYWORDS FOR FORTRAN AND EFL

FORTRAN	EFL
double precision	*long real*
function	*procedure*
subroutine	*procedure (untyped)*

Numeric Labels

Standard statement labels are identifiers. You can also have a numeric (positive integer constant) label; the colon is optional following a numeric label.

Implicit Declarations

If you use a name but it is not in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If you use the name in the context of a procedure invocation, EFL assumes it is a procedure name; otherwise it assumes a local variable defined at nesting level 1 in the current procedure. The first letter of the name determines the assumed type. The association of letters and types can be given in an **implicit** statement:

implicit (*letter-list*) *type*

In this statement *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement is present, the following rules are assumed:

implicit (a–h, o–z) real
implicit (i–n) integer

Computed goto

FORTRAN contains an indexed multi–choice branch; You can use this facility in EFL by the computed **goto**:

goto ([*label*]), *expression*

The expression must be of type integer and be positive, but it cannot be larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

Go to Statement

In unconditional and computed **goto** statements, you can separate the words **go** and **to**:

go to *xyz*

Dot Names

FORTRAN uses a restricted character set, so it represents certain operators by multi-character sequences. There is a *dots=on* option; that forces the compiler to recognize the forms in the second column:

<	**.lt.**
<=	**.le.**
>	**.gt.**
>=	**.ge.**
==	**.eq.**
~=	**.ne.**
&	**.and.**
|	**.or.**
&&	**.andand.**
||	**.oror.**
~	**.not.**
true	**.true.**
false	**.false.**

In this mode, no structure element may be named **lt**, **le**, etc. The readable forms in the left column are always recognized.

Complex Constants

You can write a complex constant as a parenthesized list of real quantities. For example:

(1.5, 3.0)

The preferred notation is by a type coercion:

complex*(1.5, 3.0)*

Function Values

The preferred way to return a value from a function in EFL is the **return** (value) construct. However, the name of the function acts as a variable to which you can assign values. An ordinary **return** statement returns the last value assigned to that name as the function value.

Equivalence

An equivalence statement has the form:

equivalence *v1* , *v2* , . . . , *vn*

This statement declares that each *v* starts at the same memory location, where *v* is a variable name, array element name, or structure member.

Minimum and Maximum Functions

This category contains a number of nongeneric functions that differ in the required types of the arguments, and in the type of the return value. They can also have variable numbers of arguments, but all the arguments must have the same type. See Table 5J–7.

Table 5J–7
MINIMUM AND MAXIMUM FUNCTIONS

Function	Argument Type	Result Type
amin0	*integer*	*real*
amin1	*real*	*real*
min0	*integer*	*integer*
min1	*real*	*integer*
dmin1	*long real*	*long real*
amax0	*integer*	*real*
amax1	*real*	*real*
max0	*integer*	*integer*
max1	*real*	*integer*
dmax1	*long real*	*long real*

Compiler Options

You can specify a number of options to control the output of the compiler, and to tailor it for various compilers and systems. The default options are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Set options with statements of the form:

option [*opt*]

where each *opt* is of one of the following:

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

Default Options

Each option has a default setting. You can change the whole set of defaults by using the *system* option. At present, the only valid values are *system* = *unix* and *system* = *gcos*.

Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no**.

Input/Output Error Handling

The **ioerror** option accepts three values, but none of the values let you use the I/O statements in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses **ERR=** and **END=** clauses. The implementation of the Fortran 77 form uses **IOSTAT=** clauses.

Continuation Conventions

By default, continued FORTRAN statements are indicated by a character in column 6 (Standard FORTRAN). The option **continue=column1** puts an ampersand (**&**) in the first column of the continued lines instead.

Default Formats

If no format is specified for for a **read** or **write** statement and datum in an iolist, a default is provided. You can change the default formats by setting the options shown in Table 5J–8:

Table 5J–8
OPTIONS FOR SETTING DEFAULT FORMATS

Option	Type
iformat	*integer*
rformat	*real*
dformat	*long real*
zformat	*complex*
zdformat	*long complex*
lformat	*logical*

The associated value must be a FORTRAN format:

option rformat=f22.6

Alignments and Sizes

To implement character variables, structures, and the **sizeof** and **lengthof** operators, you must know how much space various FORTRAN data types require, and what boundary alignment properties they demand. The relevant options include those shown in Table 5J–9:

Table 5J–9
SIZE AND ALIGNMENT OPTIONS FOR FORTRAN TYPE

FORTRAN Type	Size Option	Alignment Option
integer	*isize*	*ialign*
real	*rsize*	*ralign*
long real	*dsize*	*dalign*
complex	*zsize*	*zalign*
logical	*lsize*	*lalign*

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option *charperint* gives the number of characters per integer variable.

Default Input/Output Units

The options *ftnin* and *ftnout* are the numbers of the standard input and output units. The default values are *ftnin = 5* and *ftnout = 6*.

Miscellaneous Output Control Options

Each FORTRAN procedure generated by the compiler is preceded by the value of the *procheader* option.

No Hollerith strings are passed as subroutine arguments if you specify *hollincall = no*.

The FORTRAN statement numbers normally start at 1 and increase by 1. You can change the increment value by using the *deltastno* option.

Examples

To show the flavor or programming in EFL, this section presents a few examples. They are short, but they show the convenience of using EFL.

Copying Files

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters:

```
procedure  # main program
character(100) line

while( read( , line) == 0 )
        write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

Matrix Multiplication

The following procedure multiplies the m x n matrix **a** by the n x p matrix **b** to give the m x p matrix **c**. The calculation obeys the formula **cij** = **aikbkj**:

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
          {
          c(i,j) = 0
          do k = 1,n
                    c(i,j) += a(i,k) * b(k,j)
          }
end
```

Searching a Linked List

Assume you have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value:

```
define LAST 0
define NOTFOUND       -1

integer procedure val(list, first, x)

#   list is an array of structures.
#   Each structure contains a thread index value,
#   an x, and a y value.
struct
          {
          integer nextindex
          integer x, y
          } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x ,
          p = list(p).nextindex)
          if(list(p).x == x).
                    return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until the list is exhausted **(p==LAST)** or until it is known that the specified value is not on the list **(list(p).x > x).** The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

Walking a Tree

As an example of a more complicated problem, imagine an expression tree stored in a common area, and that you want to print out an **infix** form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, you could implement this "tree walk" with the following simple pseudocode:

```
if this node is a leaf
            print its value
otherwise
            print a left parenthesis
            print the left node
            print the operator
            print the right node
            print a right parenthesis
```

In a nonrecursive language like EFL, you must maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value:

```
procedure walk(first)
            # print out an expression tree
integer first
            # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
            {
            character(1) op
            integer leftp, rightp
            real val
            } tree(100)                 # array of structures
struct
            {
            integer nextstate
            integer nodep
            } stackframe(100)
```

```
define NOI
define STA
#   nextst
define DOW
define LEF
define RIG
#   initia
stackdepth
STACK.next
STACK.node

while( sta
```


Relation Between EFL and Ratfor

There are a number of differences between Ratfor and EFL, since language, and Ratfor is the union of the special control structures accepted by the underlying FORTRAN compiler. Ratfor running c FORTRAN is almost a subset of EFL. Most of the features descri *Converting Older Programs* are present to ease the conversion of EFL.

A few incompatibilities remain: the syntax of the **for** statement is sl the two languages, and the three clauses are separated by semico by commas in EFL. (The initial and iteration statements can be co statements in EFL because of this change.) The input/output synta in the two languages, and there is no **FORMAT** statement in EFL. **ASSIGN** or assigned **GOTO** statements in EFL.

The major additions in EFL are character data, factored declaratio structure, assignment and sequential test operators, generic functi structures. EFL permits more general forms for expressions and p uniform syntax.

Compiler

Current Version

The current version of the EFL compiler is a two-pass translator w C. It implements all of the features described previously, except f numbers.

Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line an known) where the error was detected. The compiler gives warning that are used but not explicitly declared.

Quality of FORTRAN Produced

The FORTRAN produced by EFL is quite clean and readable. To possible, the variable names that appear in the EFL program are FORTRAN code. The bodies of loops and test constructs are inde numbers are consecutive. Few unnecessary **GOTO** and **CONTINU** used. It is considered a compiler bug if incorrect FORTRAN is pr escaped lines).

```
end
```

Constraints on the Design of the EFL Language

Although FORTRAN can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into FORTRAN. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by FORTRAN.

External Names

External names (procedure and COMMON block names) must be no longer than six characters in FORTRAN. Further, an external name is global to the entire program. So, to compile EFL procedures separately you can have only one level of an external name.

Procedure Interface

The FORTRAN standards permit arguments to be passed between FORTRAN procedures either by reference or by copy–in/copy–out. This vague specification is evident in EFL. A program that depends on the method of argument transmission is illegal in either language.

FORTRAN has no procedure–valued variables. A procedure name can only be passed as an argument or be invoked; it cannot be stored. FORTRAN (and EFL) would be simpler if a procedure variable mechanism were available. The most serious problem with FORTRAN is its lack of a pointer–like data type. The implementation of the compiler would be easier, and the language would be more simple if you could simulate pointers by using subscripts; but they founder on the problems of external variables and initialization.

Recursion

FORTRAN procedures are not recursive, so EFL procedures cannot be recursive.

Storage Allocation

The definition of FORTRAN does not specify the lifetime of variables. It is difficult, but you can implement stack or heap storage disciplines by using COMMON blocks.

The search is a single **for** loop that begins with the head of the list and examines items until the list is exhausted **(p==LAST)** or until it is known that the specified value is not on the list **(list(p).x > x)**. The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

Walking a Tree

As an example of a more complicated problem, imagine an expression tree stored in a common area, and that you want to print out an **infix** form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, you could implement this "tree walk" with the following simple pseudocode:

```
if this node is a leaf
            print its value
otherwise
            print a left parenthesis
            print the left node
            print the operator
            print the right node
            print a right parenthesis
```

In a nonrecursive language like EFL, you must maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value:

```
procedure walk(first)
            # print out an expression tree
integer first
            # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
            {
            character(1) op
            integer leftp, rightp
            real val
            } tree(100)                 # array of structures
struct
            {
            integer nextstate
            integer nodep
            } stackframe(100)
```

```
define NODE             tree(currentnode)
define STACK            stackframe(stackdepth)
#   nextstate values
define DOWN             1
define LEFT             2
define RIGHT            3
#   initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first

while( stackdepth > 0 )
          {
          currentnode = STACK.nodep
          select(STACK.nextstate)
                    {
                    case DOWN:
                                        if(NODE.op == '' '')
                                        # a leaf
                                        {
                                        outval( NODE.val )
                                        stackdepth -= 1
                                        }
                                        else{
                                        # a binary operator node
                                        outch ( '' ('' )
                                        STACK.nextstate = LEFT
                                        stackdepth += 1
                                        STACK.nextstate = DOWN
                                        STACK.nodep = NODE.leftp
                                        }
                    case LEFT:
                                        outch( NODE.op )
                                        STACK.nextstate = RIGHT
                                        stackdepth += 1
                                        STACK.nextstate = DOWN
                                        STACK.nodep = NODE.rightp
                    case RIGHT:
                                        outch( ")'' )
                                        stackdepth -= 1
                    }
          }
end
```

Portability

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard FORTRAN compiler (unless you specify *Fortran77* option).

Primitives

Certain EFL operations cannot be implemented in portable FORTRAN, so each environment has a few machine–dependent procedures.

Copying a Character String

The subroutine **ef1asc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is:

```
subroutine ef1asc(a, la, b, lb)
integer a(*), la, b(*), lb
```

It must copy the first *lb* characters from *b* to the first *la* characters of *a*.

Character String Comparisons

The function **ef1cmc** is invoked to determine the order of two character strings. The declaration is:

```
integer function ef1cmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string *a* of length *la* precedes the string *b* of length *lb*. It returns zero if the strings are equal, and a positive value otherwise. If the strings are of different length, the comparison is carried out as if the end of the shorter string were padded with blanks.

EFL Design Considerations

This section details more completely the design considerations involved in the development of EFL, such as differences between EFL and Ratfor, compiler design, and constraints on the language.

Relation Between EFL and Ratfor

There are a number of differences between Ratfor and EFL, since EFL is a defined language, and Ratfor is the union of the special control structures and the language accepted by the underlying FORTRAN compiler. Ratfor running over Standard FORTRAN is almost a subset of EFL. Most of the features described in the Section *Converting Older Programs* are present to ease the conversion of Ratfor programs to EFL.

A few incompatibilities remain: the syntax of the **for** statement is slightly different in the two languages, and the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements can be compound statements in EFL because of this change.) The input/output syntax is quite different in the two languages, and there is no **FORMAT** statement in EFL. There are no **ASSIGN** or assigned **GOTO** statements in EFL.

The major additions in EFL are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions and provides a more uniform syntax.

Compiler

Current Version

The current version of the EFL compiler is a two–pass translator written in portable C. It implements all of the features described previously, except for long complex numbers.

Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and filename (if known) where the error was detected. The compiler gives warnings for variables that are used but not explicitly declared.

Quality of FORTRAN Produced

The FORTRAN produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the FORTRAN code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unnecessary **GOTO** and **CONTINUE** statements are used. It is considered a compiler bug if incorrect FORTRAN is produced (except for escaped lines).

Introduction to Debugging

Overview

Debugging tools let the user access the inner workings of programs, to see that they are running correctly. UTek provides two debuggers — **adb** and **sdb** — to debug programs written in C, Pascal, Fortran 77, or assembly language.

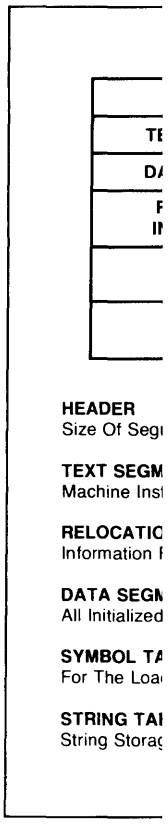
This section presents some introductory concepts about how debuggers work. It also provides information about the structure of object files, which is useful when you are using a debugger.

In most cases, **sdb** is most convenient to use because it allows you to examine the source code of the program, as well as the core image file produced when an object file does not execute. **Sdb** also allows you to call procedures explicitly.

Adb examines programs on a lower level than does **sdb**. It has facilities to examine object and core image files, as well as run programs interactively. **Adb** is especially useful for debugging programs written in assembly language, and for debugging the kernel. **Adb** also provides a primitive command file facility.

Object Files

Although **sdb** allows you to display and manipulate the program using source level constructs, debuggers examine the contents of an executable object file and the core image file created when the program halts and dumps to core. To understand how debuggers operate on the object file, the following figure is useful.



Accessing Variables

Sdb and **adb** let you do three things with variables: display t
their addresses, and change their values.

Backtraces

Adb and **sdb** let you print a C program *backtrace.* A backtrac
sequence of functions that brought you to the current point in
you check to see that the parameters were passed on correct
the next. **Sdb** prints the backtrace with source line numbers.

Termination

You can quit the debugger at any point where you can enter a
necessary to finish executing the program being debugged.

Introduction to Debugging

Overview

Debugging tools let the user access the inner workings of programs, to see that they are running correctly. UTek provides two debuggers — **adb** and **sdb** — to debug programs written in C, Pascal, Fortran 77, or assembly language.

This section presents some introductory concepts about how debuggers work. It also provides information about the structure of object files, which is useful when you are using a debugger.

In most cases, **sdb** is most convenient to use because it allows you to examine the source code of the program, as well as the core image file produced when an object file does not execute. **Sdb** also allows you to call procedures explicitly.

Adb examines programs on a lower level than does **sdb**. It has facilities to examine object and core image files, as well as run programs interactively. **Adb** is especially useful for debugging programs written in assembly language, and for debugging the kernel. **Adb** also provides a primitive command file facility.

Object Files

Although **sdb** allows you to display and manipulate the program using source level constructs, debuggers examine the contents of an executable object file and the core image file created when the program halts and dumps to core. To understand how debuggers operate on the object file, the following figure is useful.

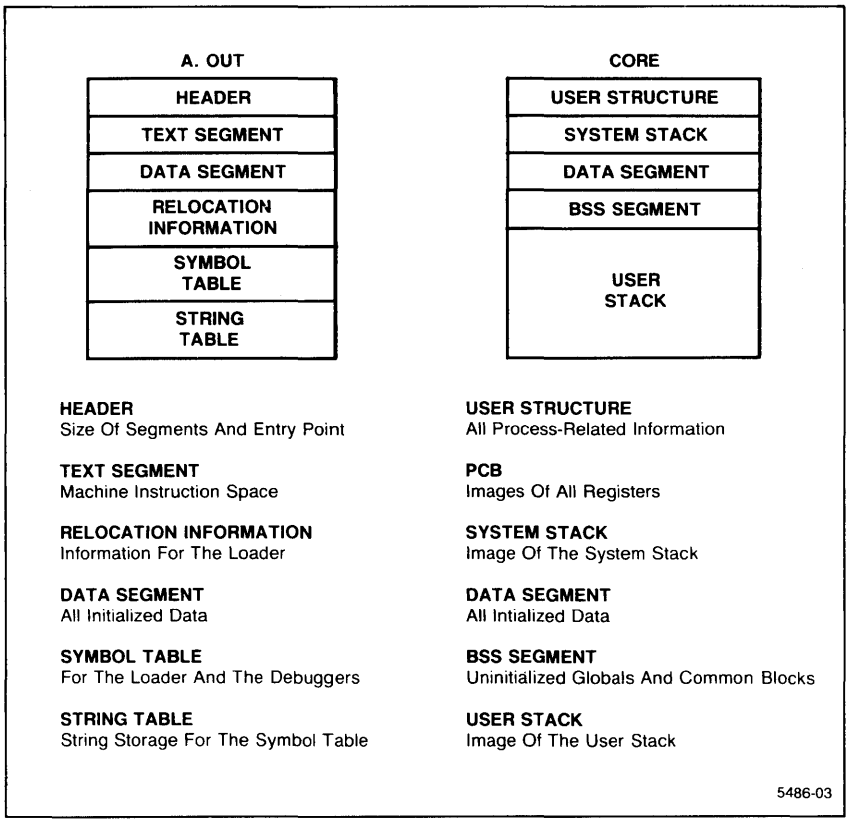


Figure 5K–1. Object Files.

Debugger Features

Debuggers control the execution of a program and examine variables. Following are some features of **adb** and **sdb** that help accomplish these tasks.

Breakpoints

The debugger itself is a program that runs as a separate process from the program being debugged. Because the debugger runs as a separate process, it can control the execution of the program, selectively starting and stopping it at different points.

To stop temporarily the execution of the program, set a *breakpoint.* It is often useful to set a breakpoint just before a procedure that causes the program to fail, so that you can examine values at that point. **Adb** and **sdb** allow you to set breakpoints, delete them, and display all breakpoints that are currently set.

Execution Control

Three basic types of commands control the execution of the program: run commands, single-step commands, and continue commands.

Run commands cause the program being debugged to execute from scratch. Execution continues until a breakpoint is reached, an error occurs, or the program terminates, normally or abnormally.

Single-step commands continue execution for a single instruction or a single line in the source file.

Continue commands continue execution from a breakpoint until another breakpoint is reached, an error occurs, or the program terminates.

Accessing Variables

Sdb and **adb** let you do three things with variables: display their values, display their addresses, and change their values.

Backtraces

Adb and **sdb** let you print a C program *backtrace*. A backtrace recreates the calling sequence of functions that brought you to the current point in the program. This lets you check to see that the parameters were passed on correctly from one function to the next. **Sdb** prints the backtrace with source line numbers.

Termination

You can quit the debugger at any point where you can enter a command. It is not necessary to finish executing the program being debugged.

Using the adb Debugger

Introduction

The UTek debugger **adb** allows you to debug programs by examining the core image produced when the program halts or by interactively executing a program. **Adb** prints results in a variety of formats, including octal, hexadecimal, and decimal. **Adb** also allows you to embed breakpoints in a program and patch errors in the object file.

Adb is a complex debugger, and this section provides an introduction to its use. For more details on **adb**, see the *UTek Command Reference, adb(1).*

Overview

Adb is especially useful for debugging programs written in assembly language and for debugging the kernel. Another useful feature of **adb** is its ability to read complex, multiple commands in from another file and then execute them.

Call the **adb** command by typing:

adb *objfil corfil*

Normally, **objfil** contains a program ready for execution. The default value for **objfil** is **a.out**. **Corfil** contains the core image of the objfile that aborted during execution. The default value for **corfil** is **core**.

Command Requests

Once you have invoked the **adb** utility, requests to **adb** take the general form:

[address] [,count] [command] [;]

where **addresses** is a location and **command** is a request you ask adb to perform. **Count** specifies how many times the command is executed. The default value of count is 1. The semicolon (;) separates multiple commands.

Leaving adb

Adb intercepts signals, so a quit signal does not let you exit from **adb**. To exit from **adb**, you must type **$q** or **$Q**.

Addresses and Their Formats

Adb has parameters for examining an address in either the object file or the core file. The **?** request examines the contents of objfile, and the / request examines the contents of corfil. These requests take the general form:

address?format

or:

address/format

Adb maintains a current address, called *dot*, similar in function to *dot* in the UTek editor. Each time you enter an address, the current address is set to that location. When using the **?** or / requests, you can advance the current address by typing <RETURN>, or decrement it by typing ^. Typing " prints the last address you typed.

The format that you request allows you to print the instruction present at the specified address in many different formats. For example,

0126?i

sets dot to hexadecimal 126 and prints the machine instruction at that address in integer format. The request:

.,10/d

prints sixteen decimal numbers starting at dot, where dot is the address of the previous item printed.

Input integer formats in **adb** include the following formats. The first value is the default:

nnnn	default radix (hexadecimal)
0Onnnn	octal
0xnnn	hexadecimal
0tnnnn	decimal

It is important to use a zero instead of the letter O in the last three formats. Otherwise, **adb** thinks that the integer is a symbol.

Formats, like their associated address, are remembered. Typing a request without a format causes the new printout to default to the previous format. For a complete list of formats, see the *UTek Commands Dictionary* **adb**(1). The following are the most commonly–used formats and their sizes in bytes.

o 2 print 2 bytes in octal

O 4 print 4 bytes in octal

d 2 print 2 bytes in decimal

D 4 print 4 bytes in long decimal

x 2 print 2 bytes in hexadecimal

X 4 print 4 bytes in hexadecimal

S n print a string using the ^X convention (where control characters are printed as ^X and and the delete character is printed as ^?)

i n print as 16032 instructions. N is the number of bytes occupied by the instruction.

Commands

We have already introduced the requests ?, /, and ;. Following are some other commonly–used **adb** commands.

= print the value of the current address

: subprocess control

$ miscellaneous requests

! escape to shell

Command Modifiers

Most commands have modifiers that direct their functions. The commands most commonly used with a modifier are $ (miscellaneous requests) and : (manage a subprocess). Recall the general form of a request to **adb**:

[address] [count] [command] [;]

Modifiers in these two lists follow either the **$** or the **:** command.

$*modifier*

- **<f** read commands from the file *f*
- **?** print process identification
- **r** print the general registers and the instruction addressed by the program counter. Dot is set to the point counter.
- **b** print all breakpoints and their associated counts and commands
- **c** C stack backtrace. If an address is given, then it as taken as the address of the current frame (instead of fp).
- **m** prints the address map

:*modifier*

- **b** sets a breakpoint. The breakpoint is executed **count –1** times before causing a stop.
- **d** delete a breakpoint at the given address.
- **r** run *objfil* as a subprocess. If an address is specified, the program is entered at this point; otherwise it is entered at its standard entry point. **Count** specifies how many breakpoints to ignore before stopping.
- **c** continue the subprocess. If an address is specified, the subprocess is continued at that address. Count specifies how many breakpoints to ignore before stopping.
- **s** the subprocess is single–stepped count times. If there is currently no subprocess, then *objfil* is run as a subprocess.
- **k** the subprocess is terminated.
- **p** causes **adb** to consider a process active.

Debugging a Core Image

Consider the following C program, example 5L–1.

```
char *cp;
main ()
{
     strcpy (cp, "This will blow up!");
}
```

Example 5L–1. C Program with Allocation Error.

This program illustrates a common error made by C programmers. No space has been allocated in which to write the string "This will blow up!". When you try to compile and execute the program, it results in a core dump, which you can examine using **adb**.

The following steps illustrate how to use **adb** to debug the example program.

To invoke **adb**, type:

adb a.out core

By default, **adb** does not have a prompt, so nothing displays on your screen after you type this request. **Adb** awaits a request. When you invoke **adb** you can use the **—p** option to specify a prompt. For example, to give **adb** an asterisk for a prompt, invoke it like this:

adb –p * ...

The first debugging request:

$c

```
_strcpy(0x10,0x800) from 0x129
_main(0x1,0x7fff8c,0x7fff94) from 0x5b
```

This gives a C backtrace through the subroutines called. The function *strcpy* was called from main.

The next request:

```
$r
r0    0xffffffff
r1    0x800
r2    0x17        start+0x17
r3    0x10cdc
r4    0x0         start
r5    0x10cf9
r6    0x10d70
r7    0x10cf0
pc    0x125       _strcpy+0xd
fp    0x7fff64
sp    0x7fff60
mo    0x0
ps    0xb40
f0    0x0         start
f1    0x0         start
f2    0x0         start
f3    0x0         start
f4    0x0         start
f5    0x0         start
f6    0x0         start
f7    0x0         start
fs    0x0         start
msr   0x70000
bpr0  0x44000000
bpr1  0x40000000
bcnt  0x100144
sc    0x31a3
_strcpy+0xd:    movsb   []
```

prints out registers including the program counter and an interpretation of the instruction at that location.

The request **$e** displays the values of all external variables:

```
$e
_environ:   0x7fff94
__pgmname:  0x7fffac
__last_err: 0x0
_errno:     0x0
__iob:      0x0
__sobuf:    0x0
__lastbuf:  0xd20
_realloc_srchlen:       0xffffffff
_end:       0x0
curbrk:     0x4da4
minbrk:     0x4da4
__sibuf:    0x0
```

The map shows the beginning and ending addresses of the text, data, and stack segments. A map exists for each file handled by **adb**. The map for the *a.out* file is referenced by **?** and the map for the *core* file is referenced by **/**. To print out information about the maps, type:

```
$m
? map       `a.out'
b1 = 0x0              e1 = 0x800            f1 = 0x400
b2 = 0x800            e2 = 0x1000           f2 = 0xc00
/ map       `core'
b1 = 0x800            e1 = 0x5000           f1 = 0x1000
b2 = 0x7ff400         e2 = 0x800000         f2 = 0x5800
```

This output displays the contents of the maps. Maps are discussed in more detail later in this section.

For this example program, it is useful to see the contents of the string pointed to by *cp*. Enter:

```
*cp/x
0x0:
data address not found
```

This command uses *strcopy* as a pointer to the *core* file and prints the information as a character string. It prints the address at the start of the string. The address is 0, which is obviously incorrect.

When you have identified the location of an error, it is useful to examine the address. The current address is now set to the address of the first argument. The request:

```
.=X
```

prints the current address (not its contents) in hexadecimal.

Setting Breakpoints

The following C program illustrates the use of breakpoints. This program changes tabs into blanks.

```
/* decodetabs.c - C program to decode tabs */

#include <stdio.h>

#define MAXLINE   80
#define YES       1
#define NO        0
#define TABSP     8

char *input = "data";
FILE *ibuf;
int tabs[MAXLINE];

main()
{
        int col;
        int c;              /* (because getc returns an int) */
        char ch;

        settab(tabs);       /* set initial tab stops   */
        col = 0;

        if ((ibuf = fopen(input, "r")) == NULL) {
                perror(input);    /* file not found */
                exit(1);
        }

        while ((c = getc(ibuf)) != EOF) {
                ch = (char) c;

                switch(ch) {
                case ' ':
                        do {
                                putchar(' ');
                                ++col;
                        } while (!tabpos(col));
                        break;
```

```
                case '0:
                        putchar('0);
                        col = 0;
                        break;
                default:
                        putchar(ch);
                        ++col;
                        break;
                }
        }
        exit(0);
}

int             /* returns true if col is a tab stop        */
tabpos(col)
int col;
{
        if (col >= MAXLINE) {
                return(YES);
        } else {
                return(tabs[col]);
        }
}

settab(t)       /* sets initial tab stops           */
int *t;
{
        int i;

        for (i = 0; i < MAXLINE; ++i) {
                t[i] = ((i % TABSP) == 0);
        }
}
```

To run this program under the control of **adb**, enter:

adb a.out -

This program does not produce a core image when it is executed, so the hyphen indicates an argument to **adb** that is not present. Again, **adb** does not display a prompt. The next thing you type is a request to the debugger.

To set breakpoints, use the command:

address:**b**

Enter the following requests to set breakpoints at the start of these functions:

settab:b
fopen:b
tabpos:b

C does not generate statement labels. Therefore you cannot set breakpoints at locations other than function entry points, unless you know the code generated by the C compiler. Note that some of the functions are from the C library.

To print the location of breakpoints, use the **$b** request:

$b

```
breakpoints
count   bkpt            command
1       _tabpos
1       _fopen
1       _settab

memory breakpoints
```

The display indicates a *count* field. **adb** bypasses a breakpoint *count -1* times before stopping. The *command* field indicates the **adb** requests to be executed each time the breakpoint is encountered. In our example, no *command* fields are present.

By displaying the original instructions at the function *settab* we can see that the breakpoint is set after the enter instruction. We can display the instructions using the **adb** request:

settab,5?ia

```
_settab:        enter   [], 0x4
_settab+0x6:    movqd   0x0, -0x4(fp)
_settab+0x9:    cmpd    0x50, -0x4(fp)
_settab+0x10:   ble     _settab+0x33
_settab+0x12:   movd    -0x4(fp), r0
_settab+0x15:
```

This request displays five instructions starting at *settab* and displays the address of each instruction. Another variation of this command displays the instructions with only the starting address:

settab,5?i

```
_settab:        enter   [], 0x4
                movqd   0x0, -0x4(fp)
                cmpd    0x50, -0x4(fp)
                ble     _settab+0x33
                movd    -0x4(fp), r0
```

Notice that we accessed the addresses from the *a.out* file using the **?** command. Usually when you ask for a printout of multiple items, **adb** advances the current address the number of bytes necessary to satisfy the request. In the last example five instructions were displayed and the current address was advanced over the full instruction.

Once you have set the breakpoints, the **:r** command runs the program.

```
:r
a.out: running
breakpoint      _settab:        enter   [], 0x4
```

Adb stops the program at the breakpoint. After the program stops at the first breakpoint, you can single-step through the program to make sure that the stack is set up. To single-step type:

```
:s
a.out: running
stopped at      _settab+0x6:    movqd   0x0, -0x4(fp)
```

At this point you can use **adb** requests. To display the stack trace, for example, type:

```
$c
_settab (0x1d4c) from 0x1f2
_main (0xw,0x7fff8c,0x7fff94) from 0x53
```

Continue the program by typing **:c**. The example program produces the following output:

```
:c
a.out: running
breakpoint      _fopen:       enter   [r4, r5, r6, r7], 0x4
```

Another useful **adb** request at this breakpoint is to print locations from the array called *tabs*. Type the request:

```
tabs,3/8x
_tabs: 0x1      0x0     0x0     0x0     0x0     0x0     0x0     0x0
       0x0      0x0     0x0     0x0     0x0     0x0     0x0     0x0
       0x1      0x0     0x0     0x0     0x0     0x0     0x0     0x0
```

to display three lines of eight locations each from the array. By this point in the execution, *settab* has set a 1 at every eighth location of the array *tabs*.

Advanced Breakpoint Usage

The same example program illustrates more advanced uses of breakpoints.

We can continue the execution of the program from the breakpoint at *_fopen* by typing:

```
:c
a.out: running
breakpoint      _tabpos:        enter   [], 0x0
```

We encounter the first breakpoint at *tabpos*. Several breakpoints of *tabpos* occur until the program has changed the tab into equivalent blanks. Since we feel that the program is working, we can remove the breakpoint at that location:

tabpos:d

Continue the program again by typing:

```
:c
a.out: running
        asdlfkjdasfasfjk
sdflkjsd        flkj
12390   84
process terminated
```

Now we can reset the breakpoint at *settab* and display the instructions located there. Enter the commands:

settab:b settab,5?ia

```
_settab:        enter   [], 0x4
_settab+0x6:    movqd   0x0, -0x4(fp)
_settab+0x9:    cmpd    0x50, -0x4(fp)
_settab+0x10:   ble     _settab+0x33
_settab+0x12:   movd    -0x4(fp), r0
_settab+0x15:
```

Note that setting a breakpoint causes the value of dot to change; executing the program under **adb** does not change dot. Therefore:

settab:b .,5?ia
fopen:b

prints the last thing dot was set to (*fopen*) instead of the location (*settab*) at which the program is executing.

Now display the breakpoints:

```
$b
breakpoints
count   bkpt            command
1       _settab         settab,5?ia
1       _fopen

memory breakpoints
```

This display shows the earlier request for the *settab* breakpoint. When the breakpoint at *settab* is encountered, the **adb** request to display the instructions is executed, resulting in the output above.

Other Breakpoint Requests

- Arguments and change of standard input and output are passed to a program as:

  :r *arg1 arg2 . . . <infile >outfile*

  This request kills any existing program under test and starts the *a.out* again.

- **Adb** allows a program to be entered at a specific address by typing:

  address:r

- The count field can be used to skip the first *n* breakpoints by typing:

  ,n:r

  The request:

  ,n:c

  can also be used to continue through the next *n* breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by:

  address:c

- The program being debugged runs as a separate process and can be killed by:

  :k

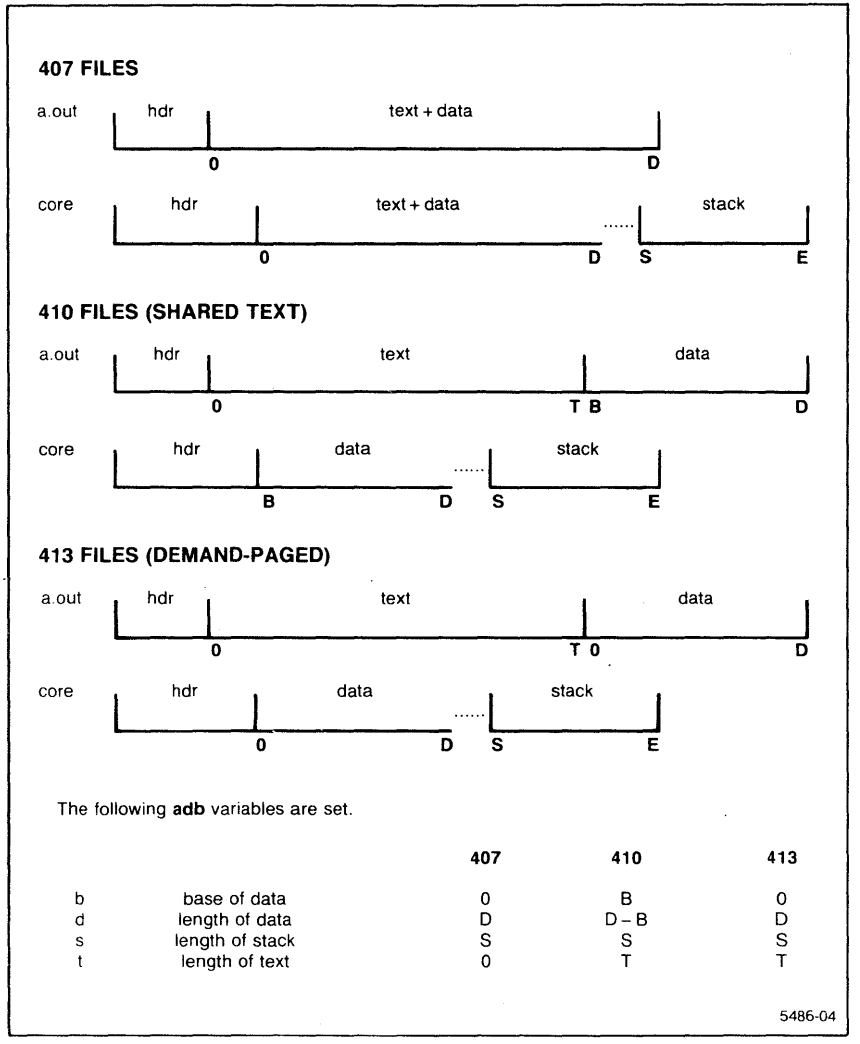
Maps

UTek supports several executable file formats. The name of the file type (also called the magic number) tell the loader how to load the program file into memory. File type 413 is the most common and is generated by the C compiler command **cc**. A 410 file is produced by a C compiler command of the form **cc -n**, whereas a 407 file is produced by the command **cc -N**. **adb** interprets these different file formats and provides access to the different segments of these files through a set of maps. To print the maps type **$m**.

In 407 files, text and data are intermixed. Therefore **adb** cannot differentiate data from instructions.

In 410 and 413 files, the instructions and data are separated. In both these types of files, the corresponding core file does not contain the program text.

Figure 5L-1 shows the display of two maps for the same program linked as a 407, 410, or 413 file.



The following **adb** variables are set.

		407	410	413
b	base of data	0	B	0
d	length of data	D	D – B	D
s	length of stack	S	S	S
t	length of text	0	T	T

5486-04

Figure 5L–1. adb Address Maps.

The *f1* field is the length of the header at the beginning of the file. The *f2* field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data *f2* is the same as the length of the header. For 410 and 413 files *f2* is the length of the header plus the size of the text portion.

The *b* and *e* fields are the starting and ending locations for a segment. Given an address **A**, the location in the file is calculated as:

b1)≤A)≤e1)–file address = (A–b1) + f1
b2)≤A)≤e2)–file address = (A–b2) + f2

You can access locations by using the **adb**–defined variables. The **$v** request prints the variables initialized by **adb**:

b	**base address of data segment**
d	**length of the data segment**
s	**length of the stack**
t	**length of the text**
m	**execution type (407,410,413)**

In Figure 1 those variables not present are 0. You can use these variables by expressions such as:

<b

in the address field. Similarly the value of the variable can be changed by an assignment request.For example:

O2000>b

sets **b** to hexadecimal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

Adb reads the header of the core image file to find the values for these variables. If the second file specified does not seem to be a core file, or if it is missing, then the header of the executable file is used instead.

Advanced Uses of adb

Formatted Dump

The request:

<b,-1/4o4^8Cn

prints four octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b the base address of the data segment print the base address to the end of the file. A negative count is used to loop indefinitely or until an error condition is detected

The format **4o48Cn** is broken down as follows:

4o print four octal locations

4^ back the current address up four locations

8C print eight consecutive characters using an escape convention

n print a <RETURN> character

The request:

<b,<d/4o4^8Cn

could have been used instead to allow the printing to stop at the end of the data segment, since **<d** provides the data segment size in bytes.

Adb lets you rearrange its output into more convenient formats. The formatting requests can be combined with **adb**'s ability to read in a script and produce a core image dump script. To do this, invoke **adb** by typing:

adb a.out core < dump

An example of such a script is:

```
0t120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace''
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request **0t120$w** sets the width of the output to 120 decimal characters (default width is 80 characters).

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095 hexadecimal. The request = can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

```
=3n"C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request **$v** displays all non-zero **adb** variables.

The request **0$s** sets the maximum offset for symbol matches to 0, thus suppressing the printing of symbolic tables in favor of hexadecimal values. Note that this request is done only for the printing of the data segment.

The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of the file with an octal address field and eight octal numbers per line.

Converting Values

You can use **adb** to convert values from one representation to another. For example, the request:

```
072 = odx
```

displays

```
0162    58      0x72
```

which are the octal, decimal and hexadecimal representations of 072 hexadecimal. **Adb** remembers the formats you last used, so typing another number will print it in the given formats.

Character values may be converted in a similar fashion. For example:

```
'a' = co
```

prints

```
a       0141
```

The = command can also be used to evaluate expressions. Note: all binary operators have the same precedence, and their precedence is lower than that of unary operators.

Writing to Files

Adb writes to files using the **w** or **W** request. The *write* request is used frequently in conjunction with the locate request, **I** or **L**. Invoke the write or locate requests by typing:

```
?I value
?L value
?w value
?W value
```

where value is an expression. The upper case requests match or write to four bytes, whereas the lower case requests match or write to two bytes.

In order to modify a file, **adb** must be called by typing:

```
adb -w file1 file2
```

When called with this option, *file1* and *file2* are created if necessary and opened for reading and writing.

As an example of the **-w** option, consider a C program that has an internal logic flag. You could set the flag using **adb** and run the program. For example:

```
adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The **:s** request is normally used to single step through a process. In this case, it starts *a.out* as a subprocess with arguments *arg1* and *arg2*. If there is a subprocess running, **adb** writes to it rather than to the file, so the **w** request causes *flag* to be changed in the memory of the subprocess.

Anomalies

Following is a list of some strange things that users should be aware of.

C backtraces are incorrect if you place a breakpoint at the first instruction of a subroutine, rather than after the first instruction. **Adb** cannot reference local or register variables.

Table 5L–1
ADB COMMAND SUMMARY

Request	Definition
? format	Print from *a.out* file according to format
/ format	Print from *core* file according to format
= format	Print the value of *dot*
?w expr	Write expression into *a.out* file
/w expr	Write expression into *core* file
?l expr	Locate expression in *a.out* file
:b	Set breakpoint at *dot*
:c	Continue running program
:d	Delete breakpoint
:k	Kill the program being debugged
:r	Run *a.out* file under **adb** control
:s	Single–step
$b	Print current breakpoints
$c	C stack trace
$e	External variables
$f	Floating registers
$m	Print **adb** segment maps
$q	Exit from **adb**
$r	General registers
$s	Set offset for symbol match
$v	Print **adb** variables
$w	Set output line width
!	Call shell to read rest of line
>name	Assign *dot* to variable or register name

Table 5L–2
ADB FORMAT SUMMARY

Request	Definition
a	The value of *dot*
b	1 byte in octal
c	1 byte as a character
D	4 bytes in decimal
F	8 bytes in floating point
O	4 bytes in octal
n	Print a new line
r	Print a blank space
s	A null–terminated character string
nt	Move to next *n* space tab
u	2 bytes as unsigned decimal
x	Hexadecimal
Y	Date
^	Backup dot
"..."	Print string

Table 5L–3
ADB EXPRESSION SUMMARY

Expression	Definition
(expression)	Expression grouping
+	Add
–	Subtract
*****	Multiply
%	Integer division
&	Bitwise and
|	Bitwise or
#	Round up to the next multiple
~	Not
*****	Contents of location
–	Integer negate

Using sdb, a Symbolic Debugger

Overview

Sdb is a debugging tool that allows you to debug programs written in C, Berkeley Pascal, and Fortran 77. **Sdb** works by examining a program at the source code level, which makes it an extremely useful tool for debugging programs written in high-level programming languages.

This section is intended to take you through the process of debugging a program with **sdb**, as well as serving as a reference for more advanced use of the debugger. An example program and some examples of commonly used **sdb** commands are provided at the end of the section.

Invoking sdb

When you invoke **sdb**, the program expects to find three things in your current directory: the executable object file, the core image produced when the program crashed, and the language source file. Note that the presence of the core image is not necessary to debug your program.

The general form for invoking **sdb** is:

sdb [a.out] [core]

where *a.out* and *core* are the default names for the object file and its core image file. If you simply type **sdb**, **sdb** finds *a.out* and *core* in the current directory.

The following example shows a typical sequence of commands to invoke **sdb**:

```
cc -go foo.c
a.out
Bus error - core dumped
sdb a.out
main:25:  x[i] = 0;
*
```

When the object file is executed, a bus error occurs, causing the file to core dump. **Sdb** reports that the error occurred in the procedure *main*, at line 25. (Line numbers are relative to the beginning of the file.) It also prints the source text of line 25. The asterisk (*) in the above example is the **sdb** prompt, indicating that you have entered the debugger and it awaits a command.

For **sdb** to work correctly, you must compile your C and Fortran 77 source files with the **-go** option, also called the *debug flag*. Pascal source files must be compiled with the **-g** option. You cannot use **sdb** to debug procedures compiled without the debug flag. If you compile some portions of a program with the debug flag, and an error occurs in a procedure that is compiled without the debug flag, **sdb** prints the procedure name and the address at which the error occurred. However, you cannot use **sdb** to examine that procedure. Debugging can continue for routines compiled with the **-go** or **-g** option.

Debugging Programs

This section takes you through a sequence of steps that you might want to follow to debug a program. These steps include tracing procedures in the stack, setting breakpoints, executing the program with breakpoints, continuing execution after breakpoint, and deleting breakpoints.

At the end of this **sdb** reference guide you will find an example program executed under the control of **sdb**.

Tracing Procedure Calls

Type **t** for the trace command. This command prints the line number and the procedure where your program halted, as well as a back trace of the procedure calls and their parameters. You can examine variables in the core file for procedures shown in the back trace. The following pages contain instructions for examining variables.

Setting Breakpoints

After finding out where the program halts, you can try to run it again, stopping just before it halts. Set a breakpoint in the procedure where the error occurred by typing **[procedure name]:b**. The general form of breakpoint commands is:

[procedure name] [file name]:[linenumber]b command

Sdb commands are summarized later in this section. The **b** command sets a breakpoint at the first executable line of a file or procedure, unless you specify a line number. The **B** command lists all of the breakpoints that are currently set.

Executing a Program with sdb

After you have set breakpoints in the source file, run it using the **r** command. This command restarts the program as though it were invoked from the shell: therefore, you must call it with the same arguments you used at the initial execution. Type **r[arguments]**. **Sdb** executes the program. When **sdb** reaches the breakpoint it prints the message "Breakpoint at [procedure name] [line number]. Now you can examine variables at the breakpoint.

Continuing from a Breakpoint

To continue to the next breakpoint, type **c**. This command continues the execution of the program until it reaches another breakpoint or the program ends. To skip a certain number of breakpoints before stopping, type **c[number]** where number is the number of breakpoints to skip. This command is especially useful if you are in a loop and only want to check variables at a certain breakpoint.

You can also type **s** to single step the next executable line of the source file. This procedure is very slow if you single step across a line that calls a subroutine compiled without the **—go** option. If the current line calls the subroutine and you do not want to single step through a subroutine, typing **S** executes up to the next line in the current procedure.

Deleting Breakpoints

If you type **d** by itself, breakpoints are deleted interactively. The location of a breakpoint displays, and you can delete it by typing **d** or **y**. If you do not want to delete a breakpoints, hit <RETURN>.

You can also delete breakpoints using the form:

[procedure name:][linenumber]d

to delete a specified breakpoint.

Leaving the Debugger

If you have finished debugging the program, you can stop its execution by typing **k**. To exit and return to the shell, type **q**.

Displaying and Manipulating the Source File

Sdb has been designed to make it easy to debug a program without constant reference to a current source listing. The debugger performs context searches within the source files of the program being debugged and displays selected portions of the source files. The commands for manipulating source text are similar to those of the UTek editors **ed** and **ex**. Like **ed** and **ex**, **sdb** keeps track of the current file and line within the file. **Sdb** also knows how the lines of a file are partitioned into procedures, so that it has a notion of current procedure.

Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for examining the source program and determining the context of the current line. The commands are:

w	print a window of 10 lines around the current line.
z	print 10 lines starting at the current line. Current line advances by 10.
<CTRL-D>	print the next 10 lines. Current line advances by 10.
p	print the current line.
e	print the current procedure and filename.

Changing the Current Line

Like the UTek editors **ex** and **ed**, **sdb** allows you to search for regular expressions in source files. The two commands are

/regular expression/
?*regular expression****?***

The first command searches forward through the file for a line containing a string that matches the regular expression. The second command does the same thing, except that it searches backward through the file. You can omit the trailing / and ? from these commands without changing their operation.

To move the current line forward or backward, you can also use the + and – commands. These commands move the current line forward or backward a specified number of lines. These two commands can also be combined with the display commands so that

+15z

advances the current line by 15, then prints ten lines.

Changing the Current Line in the Source File

The e command is used to change the current source file. Either of the forms

e *procedure*
e *filename*

can be used. The first command makes the file containing the named procedure the current file, and the current line becomes the first line of the procedure. The second command makes the named file current, and the current line is the first line of the file. An **e** command with no argument prints the current procedure and file name.

Examining Variables

Sdb allows you to display the contents of a variable in different formats, display the address of the variable, or change the contents or the value of a variable.

Displaying Variables

The general form for displaying variables is:

variable/

To display a variable in a procedure other than the current procedure, type **procedure name:variable/**.

Normally **sdb** displays the variable in a format determined by its declared type in the source program. To request a different format, a length or format specifier is added after the slash. For example, the command **errflg/bd** displays the value of the variable *errflg* in one byte with decimal format.

You can display a variable using the following format specifiers:

c	character
d	decimal
u	decimal unsigned
o	octal
x	hexadecimal
f	32 bit single precision floating point
g	64 bit double precision floating point
s	assume variable is a string pointer and print characters until a null is reached
a	print characters starting at the variable's address until a null is reached

If you display a variable in formats d, o, x, or u, you can also specify its length. The length specifier precedes the format specifier. If you do not request a specific length for a variable, it displays in the word length of the host machine, so 32 bits on a workstation. The following length specifiers are available:

b	one byte
h	two bytes (half word)
l	four bytes (long word)

The general form of a command to access a variable and display it with a different format and length is:

variable/[length] [format]

so,

i/bx

displays variable *i* in one byte hexadecimal.

Sdb also knows about structures, one dimensional arrays, and pointers, so that all of the following commands work.

array[2]/
sym.id/
psym->usage/
xsym[20].p->usage/

The only restriction is that array subscripts must be numbers.

You can also display core locations by specifying their absolute addresses. The command

1024/

displays location 1024 in decimal.

Two other very useful commands give information about variables. The = command displays the address of a variable, so

```
i=
```

displays the address of i. Another useful feature is the command

```
./
```

This redisplays the last variable you typed.

Displaying Addresses

To find out the address of a variable, type **variable=**. The general form of this command is

[procedure name]:variable=[format]

This allows you to specify a procedure name and the format of the address. Another useful variant is the command **linenumber=**. This displays the address in the object code that corresponds to the beginning of *linenumber* in the current source file.

Changing Variables

To change the value of a variable, type *variable!value*. The new value that you request can be a number, character, constant or another variable. Changing the value of variables can be very useful in determining which variable causes run–time errors.

Example sdb Routine

cat testdiv2.c

```
main() {
        int i;
        i = div2(-1);
        printf("-1/2 = %d0', i);
}
div2(i) {
        int j;
        j = i>>1;
        return(j);
}
```

cc -g testdiv2.c

a.out

```
-1/2 = -1
```

sdb

```
No core image              #Warning message from sdb
*/^div2                    # Search for procedure "div2"
6: div2(i) {               # It starts at line 6
*z                         # Print the next few lines
6: div2(i) {
7:      int j;
8:      j = i>>1;
9: return(j);
10: }
*div2:b                    # Place a breakpoint at beginning of div2
div2:8 b                   # Sdb echoes proc name and line number
*r                         # Run the procedure
Breakpoint at              # Execution stops just before line 8
div2:8:     j = i>>1;
*t                         # Print trace of subroutine calls
div2(-1)   [testdiv2.c:8]
main(1,2147483380,2147483388)   [testdiv2.c:3]
*i/                        # Print i
-1
```

```
*s                      # Single step
div2:9:         return(j);  # Execution stops just before line 9
*j/                     # Print j
-1
*8d                      # Delete the breakpoint
*div2(1)/
                        # Try running div2 with different args
0
*div2(-2)/
-1
*div2(-3)/
-2
*q                       # Exit sdb
```

Table 5M–1
EXAMPLE SDB COMMANDS

Command	Definition
t	Trace back procedure calls
35b	Set breakpoint at line 35 in current source file
test.c:55b	Set breakpoint at line 55 of file *test.c*
sub2:78b	Set breakpoint at line number 78 in *sub2*
B	Display all breakpoints
r	Run program from start without arguments
c	Continue until another breakpoint is reached
s	Single step to next executable source line
S	Single step across subroutine calls
w	Print a 10 line window around current source line
k	Kill program
!ls	Invoke shell to execute **ls** command
q	Quit, exit **sdb**
glob1/	Display variable *glob1* in default format
sub1:p->e1/	Display C structure element in procedure *sub1* referenced through pointer *p*
arr[4]/	Display array element
p.e1/	Display C structure or Pascal record element
glob2/lx	Display variable *glob2* as a 32 bit hex
glob4/s	Display string pointed to by *glob4*
sub1:glob2=lx	Display address of *glob2* in *sub1* as 32 bit hex
35=	Display address in object file corresponding to line 35
sub1:glob4!34	Change value of variable *glob4* in *sub1* to decimal 34

RCS — A Revision Control System

The Revision Control System (RCS) maintains multiple versions of a text file. The RCS programs provide the following capabilities:

- RCS stores and retrieves different versions of a file, and can reconstruct past versions.
- RCS saves disk space by storing only the changes made for each version.
- Each revision is identified and available at any time.
- RCS can merge different revisions together.
- RCS controls access to revisions so that only one user at a time can modify a file.

Identifying RCS Files

When you work with the RCS program, you deal with two types of files: an RCS file and a working file. An RCS file is identified by a *,v* extension. example:

test,v	is an RCS file
test	is the working file

The working file is the file that you can edit, compile, and read. Each time you submit a working file to RCS, it becomes an RCS file. An RCS file keeps track of each submitted working file and gives it a revision number. Either of the files can be specified with or without a full pathname.

You can think of RCS as being like a file system: When a file is updated, a copy is made and left so that previous versions of the file can be accessed. Refer to Figure 6A–1. The RCS file cabinet is comparable to an RCS directory that has two RCS files in it: *test,v* and *temp,v*. To edit the latest revision in *test,v*, you have to take the file out of the file cabinet. When you take it out, RCS makes a copy of revision 1.3, leaves it in the cabinet, and gives you a copy (the "working file") to work on. If you make changes to the working file and put it back into the file cabinet, it will be filed as revision 1.4. You can, at any time, access any revision in an RCS file.

RCS also makes it possible to lock a file drawer after you have obtained a working file so that no one else can make changes to the revision you are working on.

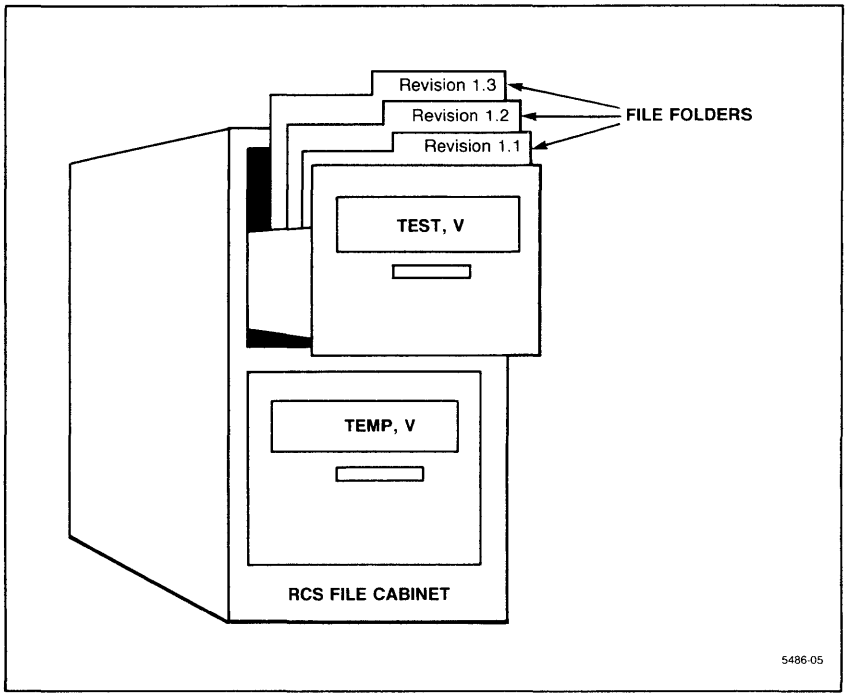


Figure 6A–1. The RCS File Cabinet.

Checking In and Checking Out

If you want to put a file under the control of RCS, you start by *checking in* the file. For example, the following command checks in a file called *temp.c*:

```
$ ci temp.c
```

The system responds:

```
temp.c,v <— temp.c
initial revision: 1.1
enter description, terminated with ^D or '.':
NOTE: This is NOT the log message!

>>
```

At this point, you enter a description of your file. This description becomes part of the RCS file.

When you check in a working file it is moved into the RCS file, and the working file no longer exists in your directory. For example, if you enter **ls** at this point, the *temp.c* will not be listed. To edit or read the file, you must use the **co** command to get a copy of your working file from the RCS file:

```
$ co temp.c
```

After you have edited the file, you can check it in again. At check–in, the working file is given the next consecutive version number, put into the RCS file, and removed from your directory. This time, instead of being prompted for a description, you will be asked to enter a log message—a description of the changes you made to the version you are checking in.

Locking the File

Your system administrator can set the locking attribute of all RCS files to *strict*. This means that you must use the **–l** option to check out a file:

```
$ co –l temp.c
```

The **–l** option ensures that you are the only one who can check in the next update.

If the locking attribute is set to strict and you do not use the **—l** option when checking out the file, when you try to check in the file the system will display the following error message:

```
ci error: no lock set by <login–name>
```

To avoid losing the changes you just made, you must first use the **rcs –l** command which locks the latest revision. You can then check in your file with the **ci** command.

If you are the only one who will be putting revisions into your RCS file, then you don't necessarily need the locking attribute set to *strict*. To change from *strict* to *non-strict* use the following command:

```
$ rcs –U temp.c
```

To change back to *strict* use:

```
$ rcs –L temp.c
```

You may want to check in a file but still maintain a file you can work on (for example, to do some more editing). If your locking attribute is set to strict, use:

ci –l temp.c

The above command checks the file in, checks it out again, and locks it. This means that you can log your first changes into one revision of the file and begin working on the next revision.

If you want to check in the changes you have made to a file, but still keep a current copy for reference use:

ci –u temp.c

This command checks the file in and checks it out again, but doesn't put a lock on the file that is checked out. If you have strict locking, you cannot change the file that is checked out. This is because you cannot check in a file that isn't locked. By default, your system administrator will have the locking on your system set to strict. You can set the environment variable RCSLOCK to *nostrict* to override this default.

Keyword Substitution

RCS has a variety of keywords that you can include in your text or within a comment statement. When the file is checked out, the keywords give helpful information about the revision.

You enter a keyword in your text as follows:

$keyword$

For example:

$Author$

When the file is checked out, each keyword is replaced with a string of information. For example, if your file contains the $Author$ keyword, then when you check out the revision the $Author$ will be replaced with the login name of the user who checked in the revision:

$Author: chrish $

The following is a list of RCS keywords available and their values:

$Author$	The login name of the user who checked in the revision.
$Date$	The date and time the revision was checked in.

$Header$	A standard header containing the RCS file name, the revision number, the date, the author, and the state. (The state is stage the revision is in. For example, Exp could mean experimental.)
$Locker$	The login name of the user who locked the revision (This information is included only if the file is not locked).
Log	The log message supplied during check in, preceded by the header. Existing log messages are NOT replaced. Instead, the new log message is inserted after $Log:
$Revision$	The revision number that RCS assigns to the revision.
$Source$	The full pathname of the RCS file.
$State$	The state assigned to the revision with **rcs -s** or **ci -s**.

Looking at the Revision History

The **rlog** command prints information about the RCS file and the revisions that it contains. **Rlog** displays information in the following format:

RCS file: rcs filename; **Working file:** working filename
head: last version number
locks: author; version number; locking attribute
access list: login names of people who have access to RCS file
symbolic names: names given to particular revisions
comment leader: character or string that precedes keywords
total revisions: number of revisions made **selected revisions:** # of revisions you asked to see
description:
description of file contents

---

revision: last revision number
date: date revision checked in **author:** login name of author
state: state of revision
lines added/deleted:# of lines added to/# of lines deleted from revision
log message

The information under the dotted line is repeated for every revision, starting with the last revision and decreasing to the first.

A useful option of **rlog** is the **-c** option, which prints only the current revision number of an RCS file.

Accessing RCS and Working Files

You can access an RCS file from anywhere in your file structure. It is good housekeeping, but not necessary, to create an *RCS* directory and keep all your RCS files there. When the RCS file is omitted in a command line or specified without a path, RCS looks first in the ./RCS directory and then in the current directory.

There are three ways to specify an RCS file argument:

- You can specify only an RCS file. The working file is placed in the current directory, and the *,v* extension is dropped. For example, if the RCS file is */cc/johnd/English/verb,v*, then the working file is named *verb*.
- You can specify only the working file. The RCS file is placed in the current directory and a *,v* extension is added. For example, if the working file is given as */cc/johnd/English/verb* then the RCS file will be *verb,v*.
- You can specify both an RCS file and a working file

For Further Information

This section has introduced you to the Revision Control System. There are many RCS commands available and each command has several options. Table 6A–1 lists RCS commands and a short description of each. For more information about each command, refer to the *UTek Command Reference*.

Table 6A-1
RCS COMMANDS

Command	Description
ci(1)	Checks in RCS revision
co(1)	Checks out RCS revision
ident(1)	Gives keyword information for specified RCS file
rcs(1)	Changes RCS file attributes
rcsdiff(1)	Compares RCS revisions
rcsmerge(1)	Merges RCS revisions
rlog(1)	Prints log messages and other information about RCS files
rcsfile(1)	Describes format of RCS file

Using Make

Overview

Structured programming helps organize programming projects, and using **make** is an important tool in structured programming. **Make** is a UTek utility that keeps track of interrelated program modules and modifies them when one or more modules change.

Programmers frequently divide programs into smaller, more manageable pieces. But it is easy to forget what files need to be reprocessed or recompiled after a change to part of the source code. You can tell **make** the sequence of files and commands necessary to complete the target program, as well as files that need to be up–to–date to complete the program. Then when you execute **make**, the proper files are created with minimal effort. Because **make** knows the relationship between the various files that comprise the program, it executes only the necessary commands, and only on files affected by a change.

This section is directed at the beginning user of **make**, as well as more advanced users who can use it as a reference guide. Go to the end of this section for information on more advanced uses of **make**.

Basic Features

Make works by updating a *target file*, which you sometimes specify on the command line. **Make** checks to see that all the files on which the target depends exist and are current. If the dependent files have not been modified since the target was modified, **make** updates the target. Thus **make** depends on its ability to find the date and time a file was last modified.

You enter the target file on the command line, but you must define what other files are necessary to build the target program. The *description file* (also known as a *makefile*) defines these dependencies. In the description file you set up a hierarchy of file dependencies, and you list commands to be executed to build the target program. When you invoke **make**, it checks the modification dates of dependent files and of the target program. If the dependent files have changed more recently than the target program, **make** looks at the description file and determines what commands to execute to bring all the program modules and the program itself up–to–date.

Later in this section the description file is explained in greater detail. For now, follow this simple example of how **make** works.

First, set up two files, *test.c and sub.c.* To create the file *test.c* enter:

```
main()
{  int t;
   t=0;
   printf("Hello world");
   t= t+sub();
   printf("%d\n",t);
}
```

To create the file *sub.c* enter:

```
sub()
{   int j;
    j=5;
    return(j);
}
```

Together *test.c* and *sub.c* compile to produce an executable file called *test* that produces the output "Hello world5." Create a description file called *makefile* that defines the dependencies of the executable file *test.* Enter:

test: test.c sub.c <RETURN>
<TAB>cc test.c sub.c -o test

The first line of *makefile* says that the executable file *test* depends on *test.c* and *sub.c.* The second line of *makefile* lists commands to be executed. It tells **make** to compile *test.c* and *sub.c* and send the output to a file called *test.* To run **make**, enter **make test.** In response to your entry the system displays the commands as it executes them.

make test

```
cc test.c sub.c -o test
```

The default name for a description file is *makefile* or *Makefile,* so when you invoke **make** it uses the *makefile* to search for other files that the target *test* depends on, brings the dependent files up–to–date, and executes the associated commands.

If no description file is present, **make** can still update a target file. See the "Make and Default Rules" section for further information.

Description Files

The most fundamental part of using **make** is writing the description file or *makefile*. As we stated earlier, it defines the dependencies for all the parts of a program. The example description file above defined the dependencies for one target, the file *test*, but a description file can define the dependencies for many different targets. So the description file can define several hierarchies of dependency at the same time, as shown in the following program.

```
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
```

The dependencies that you define in a description file are the *rules* that **make** uses for its operation. But in writing a description file, you do not have to specify every dependency explicitly. **Make** contains its own *default rules*. As an example, let's consider the description file we wrote earlier, but with a minor change. The target program test now depends on *test.o* and *sub.o* instead of *test.c* and *sub.c*.

test: test.o sub.o <RETURN>
<TAB> cc test.o sub.o -o test

Normally, the files *test.c* and *sub.c* must be compiled to create the object files. But the default rules of **make** know that the source file suffix *.c* in C can be transformed to the object file suffix *.o*. **Make** contains a table of commonly-used program and processor suffixes and rules for transforming one suffix to another. The following are commonly-used suffixes that **make** recognizes. See the "Make and Default Rules" section for instructions on printing all the suffix transformation rules.

,v	RCS revision file
.c	C source code
.s	Assembly code
.y	Yacc input
.sh	Shell script
.l	Lex input
.p	Pascal source code
.f	Fortran 77 source code
.r	Ratfor source code
.e	EFL source code
.h	C include file
.o	Object code

Because **make** has default suffix transformation rules, always write the description file so that the object files depend on the target program. The target actually relies on executable files, instead of on source files. The target relies on the executable files, and the executable files depend on source code. For example, if the file *x.c* has a "#include defs" line, the object file *x.o* depends on the file *defs*; the file *x.c* does not.

In summary, **make** has default rules that underlie the function of the description file.

Parts of the Description File

This section discusses the rules used to write a description file. The parts of the description file discussed in this section include: target rules, suffix rules, UTek commands, macros, and comment lines. This section also discusses description file syntax.

Target Rules

The *target rule* communicates dependency information to **make**. The target rule is also called a *dependency rule*. You can recognize a target rule in the description file because it contains an embedded colon. Target rules have the form:

target: dependents

where *target* is the file or operation you update when you enter **make** *target*. *Dependents* are the files and targets *target* relies on for information. When the dependent files are updated, the target must also be updated. The target line from the example description file you entered earlier was:

test: test.o sub.o

This line says that the target program test depends on the files *test.o* and *sub.o*. When you enter **make test**, **make** checks to see that *test.c* and *sub.c* have been updated as recently as the target program *test*.

You can change the last–modified time of a dependent without changing the contents of the dependent file using the **touch** command. For example, enter:

touch *filename*

The last–modified time for that file becomes the current time. Thus **make** can update a target, whether its dependents changed or not.

The target line can also take the form:

target:: dependents

The double colon means that more than one rule can exist for a particular target. In other words, the target has two different sets of dependents. So to place a target on two or more lines in a description file you enter:

target1 target2:: dep1 dep2
target2:: dep3

Entering these two target rules in the same description file indicates that *target2* depends on more than one set of names. If *dep3* was modified more recently than *target2*, **make** brings *target2* up–to–date, but does not modify *target1*.

If *dep1, dep 2,* and *dep3* have all changed more recently than *target1* and *target2*, both target rules are interpreted. **Make** reads each rule in the order it appears in the description file. **Make** modifies the first instance of the target, and when that modification is complete, **make** moves on to the next target.

Commands

As we discussed earlier, **make** has a mechanism that modifies a target. You provide this mechanism by entering a UTek command line that is associated with each target line. These UTek commands must be executed to bring the target up–to–date.

The UTek command line follows the target rule (dependency) line. A semicolon (;) or <RETURN> followed by <TAB> separate the dependency line from the command line. For example:

```
test: test.o sub.o <RETURN>
<TAB> cc test.o sub.o -o test
```

In this example, <RETURN> followed by <TAB> separates the target line from the command line. The command line in the example executes the **cc** command with the **-o** option on the files *test.c* and *sub.c*. (Recall that **make**'s default rules for suffix transformation change *.o* files to *.c* files.)
As **make** executes each command, the command displays on your terminal. To turn off the display of individual commands, precede the description file entry for that command with the character @.

Make checks the exit status of each command. A non-zero exit status for any command causes **make** to terminate, unless you precede the description file entry for that command with the character –.

Normally, **make** executes the UTek commands directly, instead of passing them to the shell. This can cause problems with UTek commands that work together. For example, the target rule:

```
target:
        cd ..
        make all
```

When **make** tries to read this target rule, the error message *make : cd : No such file or directory* displays. The **cd** command is executed separately from the **make** command. And because it is executed as a separate invocation, the **make** command has no knowledge of the **cd** command.

To correct this problem, you can enter a semicolon (;) to separate each UTek command, instead of separating the command with a <RETURN>. For example,

```
target:
        cd ..; make all
```

Make invokes the shell to execute a command when the command includes the special characters =, <, >, |, ^, (,), &, *, ?, [,], :, $, ', \, or <RETURN>. The characters ' and " in a command do not cause **make** to invoke the shell.

To summarize, the target rule and the UTek commands work together. The target rule states what files depend on the target, and the commands tell **make** what UTek commands to execute when the dependent files have been modified. The general form for this part of description file is:

target :[:] dependents <separator> commands

Suffix Rules

In addition to target rules, you can define suffix transformation rules in addition to the default rules built into **make**. Suffix rules take the following form:

suffix1 suffix2: <separator> commands

where *suffix1* is a non–null file name suffix, like *.c* or *,v,* and *suffix2* is a either a null or non–null file name suffix. If *suffix2* is not null, it must begin with a period. A file ending with *suffix1* that has the same root as *suffix2* causes **make** to execute the specified commands before *suffix2* is considered current.

Normally you do not need to define suffix rules, because make transforms one suffix to another automatically for commonly–used programming languages. The rules for transforming suffixes exist as default rules. This example shows a default rule used by **make** for changing a *.c* file to a *.o* file:

.c.o:
commands:
cc [*options*] *sourcefile* –o *objectfile*

This suffix rule takes *file.c*, compiles the dependency files, and directs the output to the target name to create *file.o*.

NOTE

Suffix rules are matched in the order defined in the description file.

Defining Macros

You can use **make** to define macros at the beginning of the description file. After you define a macro at the beginning of the makefile you can substitute that macro in a target rule or a dependency line.

The line in the makefile that defines the macro contains an embedded equals sign (=). For example:

OBJECTS = x.o y.o z.o
LIBES = –lS

To invoke the macro within a target or command line, you must precede its name with a dollar sign($). Macro names that are more than one character long must also be enclosed in parentheses. For example:

```
$Z
```

or

```
$(xy)
```

The name of the macro is either the single letter following the dollar sign or the name inside parentheses. When you invoke the macro on a dependency line or a command line use $(*macroname)*; the dollar sign tells **make** to substitute the value of the macro you defined at the beginning of the *makefile*. This example shows macro definition and substitution:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS) $(LIBES) -o prog
```

For each occurrence of $(OBJECTS) **make** substitutes x.o, y.o, and z.o. And for each occurrence of $(LIBES) **make** substitutes -lS. Defining macros at the beginning of the *makefile* gives it much more flexibility. You can change many of the target rules and command lines simply by changing the macro definitions at the beginning of the description file.

You can define a macro with the null value by using the equals sign by itself. For example, enter:

```
COFLAGS=
```

Special Macros

In addition to macros that you define in the description file, **make** already has a set of special, predefined macros. The values assigned to these special macros are frequently used in *makefiles*. This saves you the trouble of entering their definitions at the beginning of the *makefile*.

Special macros include:

$$	The dollar sign character ($). A single $ indicates a macro value, so **make** needs two dollar signs to pass the character $ to the shell.
$(MAKE)	The name of the **make** utility (unless your system administrator changes it, this name is normally **make**). This is useful in multi–level description files, which are discussed later.
$(MFLAGS)	The flags you entered on the command line for this invocation of **make**. This is useful in multi–level description files, which are discussed later.
$@	The name of the current target.
$<	The list of dependencies that caused the suffix transformation rule to be used.
$?	The list of dependencies that were out of date.
$*	The root of the target name.
$%	The name of the member of the current archive file.

Suppose, for example, that you want to direct the output of a command into the current target file. Use the $@ macro to substitute for the current target name. Enter in the description file:

```
prog: $(OBJECTS)
        cc $(OBJECTS) $(LIBES) -o $@
```

This example description file entry for the target *prog* compiles the program using particular object files and libraries, then redirects the output into the target file by using –o $@.

The special macros $@, $<, $?, $*, and $% can be expanded even further.

The entry $(*specialmacro*D) expands to the directory names of its values. The entry $(*specialmacro*F) expands to the basenames of its values.

The entry $(*specialmacro:x=y*) expands every occurrence of *x* in the macro value to *y*. For example, if the macro $? has the value *file1.o file2.o*, then $(?:.o=.c) expands the value of the macro to *file1.c* and *file2.c*.

Just as **make** operates using default rules for suffix transformation, it also has default macros other than those we discussed above. To print out a list of those default macros enter:

```
make -f /dev/null -p
```

The default macros display at the beginning of the printout. The default macros include:

```
MFLAGS =
LOADLIBES =
EFLAGS =
FC = f77
RFLAGS =
RC = f77
CFLAGS =
PFLAGS =
PC = pc
AS = as
COFLAGS = -q
CO = co
LD = ld
CC = cc
LFLAGS =
LEX = lex
YFLAGS =
YACC = yacc
MAKE = make
```

For now you can ignore everything on this printout except the macros that are listed at the beginning of the file. The macros are followed by default suffix rules that are discussed later in this section.

As you can see, all of these default macros except those for command flags are defined. Those that end in FLAGS are set to the null string. They are present because many of the default rules for suffix transformation contain the FLAGS macro. Then if you define a FLAGS macro on the **make** command line, that macro is interpreted as **make** reads the default rules for suffix transformation. The exception to this is COFLAGS, which is always set to **-q**.

Enter the following *makefile* rule that uses default macros:

```
get:  get.o subs.o
            $(CC) $(CFLAGS) -o $@ get.o subs.o
```

Although $(CFLAGS) is not defined at the beginning of the *makefile*, if you invoke **make CFLAGS=-O**, $(CC) has the value **cc** and $(CFLAGS) takes the value **-O** from the definition you entered on the command line.

Special Entries

Another component of the description file is a *special entry*. Special entries refine the environment in which the **make** utility runs. Each special entry has a name that begins with a period (.). Frequently, special entries are also followed by information that modifies the **make** environment.

Some special entries correspond to **make** command options. For a list of those special entries see the *UTek Command Reference, make(1)*.

Following is a list of special entries that specify the information **make** needs to run. Each entry takes the form:

special entry: arguments

Entry	*Result*
.DEFAULT	When no rule exists for a target, the arguments following this entry are executed. A frequent use of this entry is to access SCCS files, since this version of **make** has no default rules for SCCS files.
.DIRECTORIES	Without arguments, the directory list is set to null. With a list of directories as arguments, the new directories are added to the directory list.
.IGNORE	**Make** ignores non-zero exit statuses. This is the same as the **-i** option.
&.SILENT	Suppresses the printing of commands as they are executed. This is the same as the **-s** option.
.SUFFIXES	Without arguments, the suffix list is set to null. With suffixes as arguments, the new suffixes are added to the current suffix list.

The special entry .DIRECTORIES is extremely useful. It tells **make** what directories to search for dependent files, after **make** has searched the current directory. Suppose, for example that you have two program modules, *test.c* and *sub.c*. *Test.c* exists in the current directory, but *sub.c* is in the special library file */usr/me/lib*. Use the following description file to run **make test**:

```
.DIRECTORIES: /usr/me/lib

test: test.o sub.o
        $(CC) $(COFLAGS) -o $@
```

When you enter **make test**, the system responds with *cc -o test*.

A complete list of special entries is available in the *UTek Command Reference, make(1)*.

Description File Syntax

This section summarizes some syntax rules discussed in previous sections and introduces some new ones.

- The target or dependency line uses a single colon (:) to separate the target from its dependent files. If the same target is present on two different dependency lines, both lines must use a double colon (::) to separate the target from its dependent files.
- The target rule is separated from its associated command by a semicolon (;), or by <RETURN> followed by <TAB>. UTek commands are passed to a separate invocation of the shell, unless you separate them with a semicolon.
- The dependent filenames in both target and suffix rules cannot contain the special characters :, >, &, ¦, space, and <TAB>, unless the special character is preceded by a backslash (\). All default suffix rules enclose filenames in double quotes ("). It is good practice to enclose filenames in double quotes when you write suffix rules into your makefile.
- A line containing an embedded equals sign (=) not preceded by a colon (:) or a tab defines a macro. A macro definition that has no characters to the right of the equal sign has the null string as its value.
- The number sign (#) begins a comment line. **Make** ignores blank lines and lines beginning with #.
- If a non–comment line of the description file is too long, you can continue it by placing a backslash (\) at the end of the line. The backslash, <RETURN>, and following blanks are replaced by a single space. Do not use a backslash at the end of a comment line.
- The @ symbol in front of the description file command line causes **make** to execute the command, but not display it as it is executed.

Invoking the Make Command

The general form for invoking **make** is:

make *options macro definitions targets*

where all arguments to the **make** command are optional. Summarizing these arguments clarifies how **make** works.

Make first examines the macro definitions, and substitutes their values everywhere they are invoked in the description file. The macro definitions on the command line take precedence over the definition of the same macro in the description file.

After **make** substitutes for the value of the macros, it examines the options. The following list details the most commonly–used options. For a complete list of options, see the *UTek Command Reference, make(1).*

–d	Print debugging output. See the examples at the end of this section for further information on debugging.
–f*file*	Use *file* instead of *makefile* or *Makefile* as the name of the description file.
–m	Do not bring the description file up–to–date.
–n	Print the commands used to bring the target up–to–date, but do not execute them. However, command lines in the description file beginning with **make** are executed.
–N	Print the commands used to bring the target up–to–date, but do not execute them unless you enter $(MAKE) or ${MAKE} on the description file command line. This lets you trace multi–level description files, which are discussed later in this section.

After reading the macro definitions and the options of the command line, **make** assumes that the remaining arguments are target names. If you do not use the **–f** option to specify a file where **make** should look for the target name, by default it looks in *makefile* or *Makefile*. You can specify multiple target names on the command line; as **make** reads multiple names it updates the targets in the order they appear on the command line, from left to right.

If you do not specify a target name, **make** updates the first target in *makefile* or *Makefile*.

Advanced Uses of Make

Once you are familiar with the basic features of **make** and how to write a description file, you can use **make** for more advanced applications. The topics covered in this section include: using **make** for archiving and library functions, writing new implicit rules for yourself or for your system, and using **make** to debug modules.

How Make Reads Default Rules

To understand how **make** reads default rules it is necessary to understand how the command reads the description file and how **make** fills in the undefined gaps of the makefile with default rules.

Make and the Description File

When you invoke **make**, you specify one or more targets. In this case **make** searches inside the description file in the current directory for the rules that apply to those targets.

If you simply type **make**, without specifying a target, **make** looks in *Makefile* or *makefile* for rules to execute. If you invoke **make** without the target but with the **-f***file* option, it searches *file* for rules to execute. When **make** finds the description file, it looks at the first target rule that does not begin with a period. Many makefiles have a target rule for "all" (that is, all the modules of a program) as the first target rule, so that simply typing **make** updates all the program modules.

If you type **make** *target*, but a description file is *not* present in the current directory, **make** uses the default rules. Using the previous example, enter:

```
make test
```

The default rule that compiles a *.c* file into a *.o* file is read. When you want to compile only one module of a program with standard features, the default rules provide a fast and efficient way to accomplish the task without writing a description file.

When a description file is present, **make** first expands the macros. **Make** substitutes for all instances of the first macro you defined, and when that operation is complete, **make** substitutes for all instances of the next macro. So, if you define the same macro to be two different things at different points in the description file, only the last definition of the macro is interpreted. **Make** substitutes for *all* the macros in the description file before it does anything else.

When **make** completes the expansion of macros, it brings the target up-to-date with respect to its dependents. **Make** looks at the file system to see when the target and the files it depends on were last modified. If the last-modified date of the target is older than the last-modified date of its dependents, **make** executes the commands associated with the target rule.

Make executes some commands directly; other commands **make** passes to an invocation of the shell. **Make** passes a command to the shell if the command is a built-in Bourne shell command, or if the command contains a Bourne shell or C shell metacharacter. **Make** displays each command as it is executed, unless you invoke it using the **-s** option, or the special entry .SILENT is present in the description file.

Make and Default Rules

After **make** reads the description file, it reads default rules to fill in many gaps. For example, target rule dependents can be updated files; **make** knows what commands to execute to bring a file up-to-date. **Make** searches for default rules in a particular order, an order that affects its interpretation of the description file.

A set of default rules is stored inside **make** itself. You can turn off these rules by invoking **make** with the **-B** option.

The second location where **make** looks for default rules is in the system-wide file */usr/lib/makerules*. The system administrator can set up default rules in this file for all the users on the system. This feature is provided for UTek facilities that do not have access to the source code of **make**. You can turn off these rules by invoking **make** with the **-Y** option.

After **make** looks at system-wide default rules, it looks for default rules set up by the individual user. You can write personal default rules and put them in the file *$HOME/.makerc*. To change the file name where **make** searches for personal default rules to another file name, set the environment variable MAKERULES equal to another file name. You can turn off these rules by invoking **make** with the **-R** option.

Whenever **make** is invoked, it attempts to bring *makefile* and *Makefile* up-to-date. **Make** does this by using a set of *automatic rules*. You can turn these rules off by invoking **make** with the **-m** option. You can also turn the automatic rules off by invoking make with the **-f***file* option. The **-n** and **-N** options are ignored when automatic rules are read.

The order in which **make** reads automatic rules is similar to the order in which it reads normal default rules. First, **make** looks within itself. Then it examines the file */usr/lib/mfrules* for system-wide automatic rules. These rules are written by the system administrator. They provide the ability to change automatic rules for UTek facilities that do not have access to source code. After **make** reads the system-wide automatic rules file it reads personal automatic rules in the file *$HOME/.mfilerc*. To store personal automatic rules in another file name, set that file name in the environment variable MFRULES.

You can examine the complete set of default rules and macro definitions by typing:

make -pfn /dev/null 2>/dev/null

This list includes system–wide and personal default rules, as well as default rules from **make** itself.

Writing Default Rules

As this section discussed earlier, default rules take the same form as the rules you write into a description file; **make** just reads default rules from a different location. Suffix rules have the general form:

suffix1.suffix2:
commands

As an example suppose that the files *foo.c, bar.c* and *baz.c* are in your current directory. Enter the following description file:

```
CLFAGS = -O
foo: foo.o bar.o baz.o
        $(CC) $(CFLAGS) -o foo foo.o bar.o baz.o
```

Enter **make**. This executes the default rule that turns *.c* files into *.o* files and prints the following commands:

```
cc -O -c foo.c
cc -O -c bar.c
cc -O -c baz.c
cc -O -o foo foo.o bar.o baz.o
```

The default rule for transforming *.c* to *.o* suffixes exists for the most normal case. But suppose that you want to compile C source into assembly language, then run it through a **sed** script and assemble it. In the description file enter:

```
$(CC) $(CFLAGS) -S $<
sed -f sed.script $*.s ¦$(AS) -o $*.o
rm -f $*.s
```

When you have entered this rule in your personal default rules file, type **make**. **Make** executes the following commands:

```
cc -O -S foo.c
sed -f sed.script foo.s ¦ as -o foo.o
rm -f foo.s
cc -O -o foo foo.o bar.o baz.o
```

Writing your own suffix rules and placing them in your personal default rules file lets you adapt **make** to your special needs.

Multi-Level Description Files

Multi-level description files provide a way for you to use **make** to execute other description files. For example, if you have related description files in different directories, but all those description files are related, you can make a master description file that executes **make** for all the other files.

To help you understand how this works, this discussion presents an example of a multi-level description file. This example has description files for two levels of the directory hierarchy; one in the directory */usr.man*, and the other in the directory */usr.man/man1*. The description file for the *man1* directory is duplicated in the directories */usr.man/man[2-8]*. So the description file in the directory */usr.man* essentially executes a **make** for all the description files in its subdirectories. This is a convenient way to maintain all the manual pages using one **make** command.

Following are the two example description files. The first is in the directory */usr.man*, while the second is in the directory */usr.man/man1*.

```
#Description file for usr.man
SUBDIR= man0 man1 man2 man3 man4 man5 man7 man8

all:    ${SUBDIR}

${SUBDIR}:
        cd $@; $(MAKE) $(MFLAGS)

clean:
        for i in ${SUBDIR}; do cd $$i;
        $(MAKE) $(MFLAGS) clean; done

install:
        for i in ${SUBDIR}; do cd $$i;
        $(MAKE) $(MFLAGS) install; done
```

Followir
same ta
know e>
names.
commar
execute

To maintain a library with two members, *ctime.o* and *fopen.o*, cre
makefile:

```
lib: lib(ctime.o) lib(fopen.o)
        echo lib up-to-date
```

The members of the library are enclosed in parentheses. The pa
make to interpret the members with a *.a* suffix.

When you type **make lib**, **make** executes the commands defined
rule above. It breaks *lib* and *ctime.o* apart and defines two macr
$* = ctime. Then **make** finds the file *ctime.c*, and sets the $< m
Now that **make** has the *.c* file *ctime.c*, it is compiled, archived, an
.o file is deleted. **Make** performs the same process on the library
The resulting members of the *lib* archive library are *ctime.a* and *f*

As mentioned earlier in this section, the special macro $% define
member of an archive. This macro is evaluated each time the ta
($@) is evaluated. So when you write a description file rule for a
you can substitute $% for the archive member in parentheses. F
in the description file:

```
lib: lib(ctime.o)
        $(CC) -c $(CFLAGS) $<
        ar rv $% $*.o
        rm -f $*.o
```

To let *ctime.o* have dependencies, you must include the full path
dependent on the dependency line. For example, enter:

```
lib(ctime.o): $(INCDIR)/stdio.h
```

Debugging Make: the -d Option

The **-d** option lets you debug the **make** program. Use this optio
or is not executing the commands that you expect.

To invoke the debugging option enter:

make -md *target*

The -m option is necessary to suppress the automatic rules that
description file is present.

The output from this command is very long and detailed. This section explains the broad outlines of the **-d** option; its specific output depends on the program **make** is updating. The first portion of the debugging output details the commands that **make** executed. The second portion of the output lists the dependencies that **make** used to accomplish its actions.

The following sections take portions of the output of the command **make -md make**; in other words, debugging how **make** creates the target program "make."

Debugging Output: Commands

Consider this first portion of the output:

```
doname(make,0)
doname(defs,1)
doname(./RCS/defs,v,2)
TIME(./RCS/defs,v)=448215757
co -q ./RCS/defs,v
TIME(defs)=448306162
```

Let's examine this output line by line.

doname(make,0)
The target program of **make** is called "make."

doname(defs,1)
The program depends on the target *defs*.

doname(./RCS/defs,v,2)
The target *defs* implicitly depends on *./RCS/defs,v*.

TIME(./RCS/defs,v)=448215757
The last modification date of *./RCS/defs,v* is 448215757. All dates are relative to each other in **make**.

co -q ./RCS/defs,v
This command is executed to make *defs* out of *./RCS/defs,v*.

TIME(defs)=448306162
Make gives *defs* the current time as the time of modification.

After listing the commands and their dependents, and the new modification times, the **-d** output displays all the directories that were opened during execution of the **make** command. For example,

```
6: /usr/include/sys
5: /usr/include
4: .
3: ./RCS
```

After printing the open directories, the **-d** option prints the final values of all the macros. If a macro has been defined in many places this can be useful, because **make** only interprets the final values of all macros. Here is an example of the first few macro values:

```
? = defs rcsid.o ident.o main.o doname.o misc.o
files.o dosys.o gram.o
@ =
< = gram.c
* = gram
XTESTDIR = /merlin/disks/curtests
XDESTDIR = /merlin/disks/current
XCC = /usr/16k/bin/c16
```

Debugging Output: Dependencies

The dependencies from the **-d** option fall into several categories.

- The "leaf" files, those that are at the bottom of the hierarchy of dependency.
- The list of targets, what they depend on, and the commands they cause to be executed.
- The name of the target program (MAIN NAME).
- Default rules.
- Suffixes **make** used to learn about implicit dependencies.

Each of these dependencies could have been used, according to the description file and the other places **make** looks to interpret its rules. But only some of the dependencies are used. You can tell which were used by the notation **done=(***number)* next to each dependent. Most commonly you will see **done=0** and **done=2**. Zero (0) means that the dependency was not updated, and 2 means that it was changed. The value 1 signifies an intermediary step between beginning and completion, and the value 3 means an error occurred when trying to change a dependent.

Let's consider some examples of the various kind of dependency information displayed using the **-d** option.

The dependencies at the bottom of the hierarchy take the form of a path name. Consider these examples:

```
./RCS/gram.y,v done = 2
./RCS/dosys.dds,v done = 0
./RCS/files.c,v done = 2
```

Following the information about dependencies that **make** was ready to use, **make –d** displays possible targets, what they depend on, and the commands necessary to update the target:

```
rcsid.c:  done = 2
depends on: ident.c main.c doname.c misc.c files.c dosys.c gram.y
commands:
        mklog –f "$(EXTLIBS) $ (HDRS) '$(WHICH) $(CC)'"$(STATE)
        rcsid.c $(SRCS)


printnew:  done = 0
depends on: documents ident.c main.c doname.c misc.c files.c dosys.c gram.y
commands:
        $(PRINT) $? Makefile ¦ $(LPR)
        touch printnew
```

Following the information about targets, **make –d** displays the name of the target program. MAIN NAME always indicates **make**'s primary target.

```
make:  done=2 (MAIN NAME)
depends on:  defs rcsid.o ident.o main.o
             doname.o misc.o files.o dosys.o
             gram.y
commands:
        $(CC) $(CFLAGS) –o $(PGM) $(OBJS)
        rcsid.o $(LDFLAGS)
```

After the **–d** option prints information about the main target program, it displays all the default rules, again using done = 2 to indicate whether or not **make** used them. For example:

```
.l.out  done=0
commands:
        $(LEX) $(LFLAGS) "$<"
        $(CC) $(CFLAGS) lex.yy.c $(LOADLIBES)
        –ll –o "$@"
        @–rm –f lex.yy.c
```

After displaying the default rules, **make –d** displays the suffixes **make** used to learn what implicit dependencies were present. Example output might look like this:

```
.SUFFIXES:  done=0
depends on: .sh,v .sh .o .c .f .e .r .y .l .s
            .p .h .o,v .c,v
```

Make and the -p Option

If you want to know what the default rules and macros will do before you execute the **make** command, you can use the **-p** option. This option prints general information about macros, rules, and dependencies. It tells you what the makefile and default rules will do. It takes this information from all the sources that **make** reads: source code rules, system-wide rules, personal rules, and automatic rules. This is a very useful option if you want to know *in detail* what **make** will do when executed.

To use the **-p** option enter:

make -mp *target*

The **-m** option is necessary to suppress the automatic rules that are read when no description file is present.

The output of **make -mp** is almost identical to that of **make -md**. With the **-p** option, macros display first, followed by rules. With the **-p** option, the commands **make** would execute display at the end of the output.

You can interpret the output in the same way that you would interpret the **-d** output. See the previous section for details on interpreting the **-d** output.

When you have entered this rule in your personal default rules file, type **make**. **Make** executes the following commands:

```
cc -O -S foo.c
sed -f sed.script foo.s ¦ as -o foo.o
rm -f foo.s
cc -O -o foo foo.o bar.o baz.o
```

Writing your own suffix rules and placing them in your personal default rules file lets you adapt **make** to your special needs.

Multi-Level Description Files

Multi-level description files provide a way for you to use **make** to execute other description files. For example, if you have related description files in different directories, but all those description files are related, you can make a master description file that executes **make** for all the other files.

To help you understand how this works, this discussion presents an example of a multi-level description file. This example has description files for two levels of the directory hierarchy; one in the directory */usr.man*, and the other in the directory */usr.man/man1*. The description file for the *man1* directory is duplicated in the directories */usr.man/man[2-8]*. So the description file in the directory */usr.man* essentially executes a **make** for all the description files in its subdirectories. This is a convenient way to maintain all the manual pages using one **make** command.

Following are the two example description files. The first is in the directory */usr.man*, while the second is in the directory */usr.man/man1*.

```
#Description file for usr.man
SUBDIR= man0 man1 man2 man3 man4 man5 man7 man8

all:    ${SUBDIR}

${SUBDIR}:
        cd $@; $(MAKE) $(MFLAGS)

clean:
        for i in ${SUBDIR}; do cd $$i;
        $(MAKE) $(MFLAGS) clean; done

install:
        for i in ${SUBDIR}; do cd $$i;
        $(MAKE) $(MFLAGS) install; done
```

Following is the description file for */usr.man/man1*. As you can see, it has all the same targets as the description file of its parent directory. It is not important to know exactly what each target does, but notice that each file has the same target names. In the *usr.man* description file the macros $(MAKE) (which is the **make** command) and $(MFLAGS) (the options to the **make** command) work together to execute a **make** for its corresponding target name in the subdirectory.

```
#description file for the directory usr.man/man1
CAT=            /usr/man/cat1
DESTDIR=        /usr/man/man1
XDESTDIR=       /ecs/disks/current
COFLAGS= -q -P

SRCS=           admin.1\
                ar.1\
                arcv.1\

VXSRCS=         adb.1.vax

WKSRCS=         adb.1.wk

ALLSRCS= $(SRCS) $(VXSRCS) $(WKSRCS)

all:            $(ALLSRCS)

clean:          rcsin $(ALLSRCS)
                rm -f $(ALLSRCS)

install:$(ALLSRCS)
        @-mkdir $(CAT)
        @-mkdir $(DESTDIR)
        @-for i in $(SRCS); do \
                if [ -M $(DESTDIR)/$$i -lt -M $$i ] ; \
                then \
                install -c -o sys -m 444 $$i $(DESTDIR) ; \
                echo "Updated $$i" ; \
                fi ; \
        done
        @-for i in $(VXSRCS); do \
                Name=`basename $$i .vax` ; \
                if [ -M $(DESTDIR)/$$Name -lt -M $$i ] ; \
                then \
                install -c -o sys -m 444 $$i $(DESTDIR)/$$Name ; \
                echo "Updated $$i -> $$Name" ; \
                fi ; \
        done
```

```
                @-for i in $(WKSRCS); do \
                        Name=`echo $$i | sed 's/\.wk$$/wk/\
;s/\.stratos$$/st/;s/.merlin$$/me/'` ; \
if [ -M $(DESTDIR)/$$Name -lt -M $$i ] ; \
then \
install -c -o sys -m 444 $$i $(DESTDIR)/$$Name ; \
echo "Updated $$i -> $$Name" ; \
fi ; \
done
```

Basically, this second description file defines macros that describe three different kinds of manual pages, to be installed in three different places. The important thing to note is that instead of executing a **make install** from the directory */usr.man/man1*, the first description file executes a **make install** for all the subdirectories, including */usr.man/man1*.

To ensure that the top–level description file always executes the same commands, it is important to use the pre–defined macros $(MAKE) and $(MFLAGS). They call the same **make** program throughout all the levels, and invoke it using the same options at each level.

The **–n** option to **make** normally lets you print the commands that **make** would execute, without tracing them. But this option will not work in tracing multi–level description files because if a top–level file cannot actually execute the **make** command, it can't find out any information about what action **make** would have at lower levels. You can use the **–N** option to trace a multi–level description file. When you enter **make –N**, this displays commands without executing them. But it *does* execute the command lines of a description file that contain $(MAKE) or ${MAKE}. So if you write your top–level description files using the pre–defined $(MAKE) macro, but your lower level description files without the macro, you can always trace the lower levels.

Maintaining Archive Libraries

Make contains a rule for building libraries that is based on the *.a* file name suffix. So a *.c.a* suffix transformation rule internal to **make** compiles a C language source file, adds it to the library, and removes the obsolete *.o* file.

The actual default suffix rule to change a *.c* file to a *.a* file is:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

To maintain a library with two members, *ctime.o* and *fopen.o*, create the following *makefile*:

```
lib: lib(ctime.o) lib(fopen.o)
        echo lib up-to-date
```

The members of the library are enclosed in parentheses. The parentheses force **make** to interpret the members with a *.a* suffix.

When you type **make lib**, **make** executes the commands defined in the *.c.a* default rule above. It breaks *lib* and *ctime.o* apart and defines two macros, $@ = lib and $* = ctime. Then **make** finds the file *ctime.c*, and sets the $< macro to *ctime.c*. Now that **make** has the *.c* file *ctime.c*, it is compiled, archived, and the intermediary *.o* file is deleted. **Make** performs the same process on the library member *fopen.o*. The resulting members of the *lib* archive library are *ctime.a* and *fopen.a*.

As mentioned earlier in this section, the special macro $% defines the current member of an archive. This macro is evaluated each time the target name macro ($@) is evaluated. So when you write a description file rule for archive members you can substitute $% for the archive member in parentheses. For example, enter in the description file:

```
lib: lib(ctime.o)
        $(CC) -c $(CFLAGS) $<
        ar rv $% $*.o
        rm -f $*.o
```

To let *ctime.o* have dependencies, you must include the full path name of its dependent on the dependency line. For example, enter:

```
lib(ctime.o): $(INCDIR)/stdio.h
```

Debugging Make: the -d Option

The **-d** option lets you debug the **make** program. Use this option when **make** halts or is not executing the commands that you expect.

To invoke the debugging option enter:

make -md *target*

The -m option is necessary to suppress the automatic rules that are read when no description file is present.

Using Make and RCS Together

Introduction

This section assumes a working knowledge of both RCS and **make**, utilities described in sections 6A and 6B. The source code control of RCS, combined with the ability of **make** to update program modules, gives you a complete system for managing programming projects.

Directory Searching

The UTek version of **make** lets you write description file rules for RCS files as though the target and dependent files were all in the current directory. For example, if you have a target program called *test* where the file *test.o* depends on *test*, **make** normally searches for a file with the root name *test* as a predecessor to *test.o*.

Make has default suffix rules for RCS files. The end of this section lists all of these suffix rules. Basically, these rules tell **make** how to transform a *,v* file to an executable file. Consider the following target rule:

test: test.o sub.o

If you enter:

make test

make looks for source files that have the same root name as *test*. **Make** looks for the source files *test.c* and *sub.c* in the current directory and in a subdirectory named RCS.

Normally **make** looks in the current directory for files with the same root name. But the special entry .DIRECTORIES lists other directories where **make** looks for files. ./RCS is the default directory for the special entry .DIRECTORIES. Using the example above, **make** would look for *test.c,v* and *sub.c,v* in the RCS subdirectory.

The default list of directories contains only *./RCS*. A default suffix rule exists that transforms a *.c,v* file to a *.c* file. This means that *test.c* by default depends on *./RCS/test.c,v*. You do not need to specify this RCS dependency in your description file since it is written into the default suffix transformation rules and the default directory search.

Writing Special RCS Suffix Rules

The list at the end of this section details the default rules for RCS suffix transformation. As with other default rules, you can modify them to meet your needs.

Rules you write yourself enlarge on the default suffix rules. All the *,v* suffix transformation rules are alike in the commands they use to create a source file from a *,v* file:

```
$(CO) $(COFLAGS) "$<"
```

Substituting the default values of these macros, **make** generates the *.c* source file by executing **co –q** on the target *,v* file. After **make** generates the *.c* source file, it uses the *.c.o* suffix transformation rule to create an executable *.o* file.

But when you write your own RCS suffix rules, you can eliminate the intermediate *.c* files. Intermediate source files are unnecessary since they also exist in the RCS directory.

For example, suppose that you want to check a file out of RCS control, compile it into assembly language, run it through a **sed** script, and assemble it to create an executable file. Enter in the *makefile*:

```
$(CO) $(COFLAGS) $<
$(CC) $(CFLAGS) –S $*.c
sed –f sed.script $*.s ¦ $(AS) –o $*.o
rm –f $*.s $*.c
```

This script removes the intermediary *.s* and *.c* files.

NOTE

If the RCS file is located in an RCS directory, **co** *places the working file in the current directory, not necessarily in the parent directory of the RCS directory.*

Suffix Conversion Rules for RCS Files

Make contains default rules for performing the following RCS suffix conversions:

NULL SUFFIX RULES:

```
,v
.c,v
.s,v
.y,v
.sh,v
.l,v
.p,v
.f,v
.r,v
.e,v
```

OTHER RULES:

```
.c,v.o
.c,v.a
.s,v.a
.p,v.o
.e,v.o
.r,v.o
.f,v.o
.s,v.o
.y,v.c
.y,v.o
.l,v.o
.l,v.c
.c,v.c
.sh,v.sh
.f,v.f
.e,v.e
.r,v.r
.y,v.y
.l,v.l
.s,v.s
.p,v.p
.h,v.h
```

The Awk Programming Language

General

The **awk** is a file–processing programming language designed to make many common information and retrieval text manipulation tasks easy to state and perform. The **awk**:

- Generates reports
- Matches patterns
- Validates data
- Filters data for transmission

Program Structure

The **awk** program is a sequence of statements of the form

```
pattern {action}
pattern {action}
...
```

The **awk** program is run on a set of input files. The basic operation of **awk** is to scan a set of input lines, in order, one at a time. In each line, **awk** searches for the pattern described in the **awk** program, then if that pattern is found in the input line, a corresponding action is performed. In this way, each statement of the **awk** program is executed for a given input line. When all the patterns are tested, the next input line is fetched; and the **awk** program is once again executed from the beginning.

In the **awk** command, either the pattern or the action is omitted, but not both. If there is no action for a pattern, the matching line is simply printed. If there is no pattern for an action, then the action is performed for every input line. The null **awk** program does nothing. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

For example, this **awk** program prints every input line that has an *x* in it:

```
/x/  {print}
```

An **awk** program has the following structure:

- a <BEGIN> section
- a <record> or main section
- an <END> section

The <BEGIN> section is run before any input lines are read, and the <END> section is run after all the data files are processed. The <record> section is the section that is run over and over for each separate line of input.

Values are assigned to variables from the **awk** command line. The <BEGIN> section is run before these variable assignments are made.

The words BEGIN and END are actually patterns recognized by **awk**. These are discussed further in the pattern section of this guide.

Lexical Conventions

All **awk** programs are made up of lexical units called tokens. In **awk** there are eight token types:

- numeric constants
- string constants
- keywords
- identifiers
- operators
- record and file tokens
- comments
- separators

Numeric Constants

A *numeric constant* is either a decimal constant or a floating constant. A decimal constant is a nonnull sequence of digits containing at most one decimal point as in **12, 12., 1.2,** and **.12**. A floating constant is a decimal constant followed by e or E, followed by an optional + or – sign, followed by a nonnull sequence of digits as in **12e3, 1.2e3, 1.2e—3,** and **1.2E+3**. The maximum size and precision of a numeric constant are machine dependent.

String Constants

A *string constant* is a sequence of zero or more characters surrounded by double quotes as in **"," "a", "ab",**
and " 12". A double quote is put in a string by preceding it with a backslash. For example, "He said, \Sit!\"". A newline is put in a string by using \n in its place. No other characters need to be escaped. Strings can be (almost) any length.

Keywords

Strings used as keywords are shown in Figure 6D–1.

Keywords		
begin	break	length
end	close	log
FILENAME	continue	next
FS	close	number
NF	exit	print
NR	exp	printf
OFS	for	split
ORS	getline	sprintf
OFMT	if	sqrt
RS	in	string
	index	substr
	int	while

Figure 6D–1. Strings Used as Keywords.

Identifiers

Identifiers in **awk** denote variables and arrays. An identifier is a sequence of letters, digits, and underscores, beginning with a letter or an underscore. Uppercase and lowercase letters are different.

Operators

The **awk** has assignment, arithmetic, relational, and logical operators similar to those in the C programming language and regular expression pattern matching operators similar to those in the UTek operating system programs **egrep** and **lex**.

Assignment operators are shown in Figure 6D–2.

Assignment Operators		
Symbol	**Usage**	**Description**
=	assignment	
+ =	plus–equals	**X** + = **Y** is similar to **X** = **X** + **Y**
— =	minus–equals	**X**— = **Y** is similar to **X** = **X**—**Y**
* =	times–equals	**X** * = **Y** is similar to **X** = **X***Y**
/ =	divide–equals	**X** = **Y** is similar to **X** = **X**/**Y**
% =	mod–equals	**X** %= **Y** is similar to **X** = **X**%**Y**
+ +	prefix and postfix increments	+ +**X** and **X**+ + are similar to **X**=**X**+**1**
—	prefix and postfix decrements	— –**X** and **X**— – similar to **X** = **X** — **1**

Figure 6D–2. Symbols and Descriptions for Assignment Operators.

Arithmetic operators are shown in Figure 6D–3.

Arithmetic Operators	
Symbol	**Description**
+	unary binary plus
—	unary and binary minus
*	multiplication
/	division
%	modulus
(...)	grouping

Figure 6D–3. Symbols and Descriptions for Arithmetic Operators.

Relational operators are shown in Figure 6D–4.

Relational Operators	
Symbol	**Description**
<	less than
<=	less than or equal to
= =	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

Figure 6D–4. Symbols and Descriptions for Relational Operators.

Logical operators are shown in Figure 6D–5.

Logical Operators	
Symbol	**Description**
&&	and
||	or
!	not

Figure 6D–5. Symbols and Descriptions for Logical Operators.

Regular expression matching operators are shown in Figure 6D–6.

Regular Expression Pattern Matching Operators	
Symbol	**Description**
~	matches
!~	does not match

Figure 6D–6. Symbols and Descriptions for Regular Expression Pattern.

Record and Field Tokens

The variable **$0** is a special variable whose value is that of the current input record. The variables **$1**, and **$2...** are special variables whose values are those of the first field, the second field, and so on, of the current input record. The keyword **NF** (Number of Fields) is a special variable whose value is the number of fields in the current input records. Thus **$FN** has, as its value, the value of the last field of the current input records. Notice that the field of each record is numbered 1 and that the number of fields can vary from record to record. None of these variables is defined in the action associated with a BEGIN or END pattern, where there is no current input record.

The keyword **NR** (Number of Records) is a variable whose value is the number of input records read so far. The first input record read is **1**.

Record Separators

The keyword **RS** (Record Separators) is a variable whose value is the current record separator. The value of **RS** is initially set to newline, indicating that adjacent input records are separated by a newline. Keyword **RS** is changed to any character **c** by including the assignment statement **RS** = **"c"** in an action.

Field Separator

The keyword **FS** (Field Separator) is a variable indicating the current field separator. Initially, the value of **FS** is a blank, indicating that fields are separated by white space, that is, any nonnull sequence of blanks and tabs. Keyword **FS** is changed to any single character **c** by including the assignment statement **F** = **"c"** in an action or by using the optional command line argument **—Fc**. Two values of c have special meaning, **space** and **t**. The assignment statement **FS** = " " makes white space in the field separator; and on the command line, **—Ft** makes tab the field separator.

If the field operator is not a blank, then there is a field in the record on each side of the separator. For instance, if the field separator is 1, the record **1XXX1** has three fields. The first and last are null. If the field separator is blank, then fields are separated by white space, and none of the **NF** fields is null.

Multiline Records

The assignment **RS** = " " makes an empty line the record separator and makes a nonnull sequence (consisting of blanks, tabs, and possibly a newline) the field separator. With this setting, none of the first **NF** fields of any record are null.

Output Record and Field Separators

The value of **OFS** (Output Field Separator) is the output field separator. It is put between fields by print. The value of **ORS** (Output Record Separators) is put after each record by print. Initially **ORS** is set to a newline and **OFS** to a space. You can change these values to any string by assignments such as **ORS** = **"abc"** and **OFS** = **"xyz"**.

Comments

A comment is introduced by a # and terminated by a newline. For example:

```
# this line is a comment
```

A comment can be appended to the end of any line of an **awk** program.

Separators and Brackets

Tokens in **awk** are usually separated by nonnull sequences of blank, tabs, and newlines, or by other punctuation symbols such as commas and semicolons. Braces { . . . } surround actions, slashes / . . . / surround regular expression patterns, and double quotes "..." surround strings.

Primary Expressions

In **awk**, patterns and actions are made up of expressions. The basic building blocks of expressions are the *primary expressions*:

- numeric constants
- string constants
- var
- function

Each expression has both a numeric and a string value, one of which is usually preferred. The rules for determining the preferred value of an expression are explained below.

Numeric Constants

The format of a numeric constant was defined previously in the topic *Lexical Conventions*. Numeric values are stored as floating point numbers. Both the numeric and string value of a numeric constant are the decimal number represented by the constant. The preferred value is the numeric value.

Numeric values for string constants are in Figure 6D–7.

Numeric Constants		
Numeric Constant	**Numeric Value**	**String Value**
0	0	0
1	1	1
.5	0.5	.5
.5e2	50	50

Figure 6D–7. Numeric Values for String Constants.

String Constants

The format of a string constant was defined previously in *Lexical Conventions*. The numeric value of a string constant is **0** unless the string is a numeric constant enclosed in double quotes. In this case, the numeric value is the number represented. The preferred value of a string constant is its string value. The string value of a string constant is always the string itself.

String values for string constants are in Figure 6D–8.

String Constants		
String Constant	**Numeric Value**	**String Value**
""	0	empty space
"a"	0	a
"XYZ"	0	xyz
"0"	0	0
"1"	1	1
".5"	0.5	.5
".5e2"	50	.5e2

Figure 6D–8. String Values for String Constants.

Vars

A *var* is one of the following:

- identifier
- identifier{expression}
- $term

The numeric value of any uninitialized *var* is **0**, and the string value is the empty string.

An *identifier* by itself is a simple variable. A *var* of the form *identifier* {expression} represents an element of an associative array named by *identifier*. The string value of *expression* is used as the index into the array. The preferred value of *identifier* or *identifier* {expression} is determined by context.

The *var* **$0** refers to the current input record. Its string and numeric values are those of the current input record. If the current input record represents a number, then the numeric value of **$0** is the number and the string value is the literal string. The preferred value of **$0** is string unless the current input record is a number. The **$0** cannot be changed by assignment.

The *var* **$1, $2, . . .** refer to fields 1, 2, . . . of the current input record. The string and numeric value of **$i** for **1<= i<= NF** are those of the *i*th field of the current input record. As with **$0**, if the *i*th field represents a number, then the numeric value of **$i** is the number and the string value is the literal string. The preferred value of **$i** is string unless the *i*th field is a number. The **$i** is changed by assignment. The **$0** is then changed accordingly.

In general, $*term* refers to the input record if *term* has the numeric value 0 and to field *i* if the greatest integer in the numeric value of *term* is *i*. If **i** < 0 or if **i** > = 100, then accessing **$i** causes **awk** to produce an error diagnostic. If **NF** < i < = 100, then **$i** behaves like an uninitialized *var*. Accessing **$i** for **i** > **NF** does not change the value of **NF**.

Functions

The **awk** has a number of built-in functions that perform common arithmetic and string operations.

The arithmetic functions are in Figure 6D-9.

Functions	
exp	*(expression)*
int	*(expression)*
log	*(expression)*
sqrt	*(expression)*

Figure 6D-9. Built-in Functions for Arithmetic and String Operations.

These functions (exp, int, log, and sqrt) compute the exponential, integer part, natural logarithm, and square root, respectively, of the numeric value of *expression*. The *(expression)* can be omitted; then the function is applied to **$0**. The preferred value of an arithmetic function is numeric.

String functions are shown in Figure 6D-10.

String Function	
getline	
index	(expression1, expression2)
length	(expression)
split	(expression, identifier)
split	(expression1, identifier, expression2)
sprintf	(format, expression1, expression2...)
substr	(expression1, expression2)
substr	(expression1, expression2, expression3)
system	(expression)

Figure 6D-10. Expressions for String Functions.

The function getline causes the next input record to replace the current record. It returns 1 if there is a next input record or a 0 if there is no next input record. The value of **NR** is updated.

The function index (e1,e2) takes the string value of expressions e1 and e2, and returns the first position where e2 occurs as a substring in e1. If e2 does not occur in e1, index returns 0. For example, **index ("abc", "bc") = 2** and **index ("abc", "ac") = 0**.

The function length without an argument returns the number of characters in the current input record. With an expression argument, length (e) returns the number of characters in the string value of e. For example, **length ("abc") = 3** and **length (17) = 2**.

The function split *(e array, sep)* splits the string value of expression e into fields that are then stored in *array [1], array [2], . . . , array [n]* using the string value of *sep* as the field separator. Split returns the number of the fields found in e. The function split *(e, array)* uses the current value of **FS** to indicate the field separator. For example, after invoking n = split (**$0**), **a**[1], **a**[2, . . . , **a**[n] is the same sequence of values as **$1**, **$2 . . .**, **$NF**.

The function sprintf (**f, e1, e2 . . .**) produces the value of expressions **e1, e2 . . .** in the format specified by the string value of the expression **f**. The format control conventions are those of the printf statement in the C programming language.

The function substr *(string, pos)* returns the suffix of *string* starting at position *pos*. The function substr *(string, pos, length)* returns the substring of *string* that begins at position *pos* and is *length* characters long. If pos + length is greater than the length of *string* then substr *(string, pos, length)* is equivalent to substr *(string, pos)*. For example, substr (**"abc", 2, 1**) = **"b"**, substr (**"abc", 2, 2**) = **"bc"**, and subtr (**"abc", 2, 3**) = **"bc"**. Positions less than 1 are taken as 1. A negative or zero length produces a null result.

The function system(*string*) produces the output from executing the shell */bin/sh* on *string*. For example, to print the current date use:

```
awk 'BEGIN {date=system("date");
printf "%s" date; exit}'
```

This also means that an awk program with this line at the beginning of the file:

```
#! /bin/awk –f
```

is the same as executing **awk –f** *filename* on the shell command line.

The preferred value of sprintf and substr is string. The preferred value of the remaining string functions is numeric.

Terms

Various arithmetic operators are applied to primary expressions to produce larger syntactic units called *terms*. All arithmetic is done in floating point. A term has one of the following forms:

- primary expression
- term binop term
- unop term
- incremented var
- (term)

Binary Terms

In a term of the form:

- term1
- binop
- term2

The form *binop* can be one of the five binary arithmetic operators +, —, *(multiplication), /(division), %(modulus). The binary operator is applied to the numeric value of the operand *term1* and *term2*, and the result is the usual numeric value. This numeric value is the preferred value, but it can be interpreted as a string value (See *Numeric Constants*). The operators *, /, and % have higher precedence than + and —. All operators are left associative.

Unary Term

In a term of the form:

unop term

The form *unop* can be unary + or —. The unary operator is applied to the numeric value of *term*, and the result is the usual preferred numeric value. However, it can be interpreted as a string value. Unary + and — have higher precedence than *, /, and %.

Incremented Vars

An *incremented var* has one of the forms:

+ + var
— — var
var + +
var — —

The *+ + var* has the value *var + 1* and has the effect of *var = var + 1*. Similarly, *— — var* has the value *var — 1* and has the effect of *var = var — 1*. Therefor, *var + +* has the same value as *var*, and has the effect of *var = var + 1*. Similarly, *var — —* has the same value as *var* and has the effect of *var = var — 1*. The preferred value of an *incremented var* is numeric.

Parenthesized Terms

Parentheses are used to group terms in the usual manner.

Expressions

An **awk** expression is one of the following:

- *term*
- *term term ...*
- *var asgnop expression*

Concatenation of Terms

In an expression of the form *term1 term2...*, the string value of the terms are concatenated. The preferred value of the resulting expression is a string value that can be interpreted as a numeric value. Concatenation of terms has lower precedence than binary + and —. For example, 1 + 2 3+4 has the string (and numeric) value 37.

Assignment Expressions

An *assignment expression* is one of the forms;

var asgnop expression

In this case, *asgnop* is one of the six assignment operators:

```
=
+ =
— =
* =
/ =
% =
```

The preferred value of *var* is the same as that of *expression*.

In an expression of this form, the numeric and string values of *var* become those of *expression*:

var = expression

The expression:

var op = expression

is equivalent to:

var = var op expression

In this case, *op* is one of; +, —, *, /, %. The *asgnops* are right associative and have the lowest precedence of any operator. Thus, a + = b * = c—2 is equivalent to the sequence of assignments.

```
b = b * (0—2)
a = a+2
```

Using Awk

There are two ways in which to present your **awk** program of pattern–action statements to **awk** for processing:

If the program is short (a line or two), it is often easiest to make the program the first argument on the command line:

awk' *program* '*files*

In this command *files* is an optional list of input files and *program* is your **awk** program. Note that there are single quotes around the program in order for the shell to accept the entire string (program) as the first argument to **awk**. For example, write to the shell:

awk' **/X/ {print}** '*files*

This runs the **awk** script /x/ {print} on the input *files*. If no input files are specified, **awk** takes input from the standard input. You can also specify that input comes from the standard input by using the hyphen (-) as one of the files. The pattern–action statement:

awk'*program*'*files* –

looks for input from files and from the standard input and processes first from files, then from the standard input.

Alternately, if your **awk** program is long, it is more convenient to put the program in a separate file, *awkprog*, and tell **awk** to fetch it from there. This is done by using the **—f** option after the **awk** command:

awk —f *awkprog files*

In this command *files* is an optional list of input files that can include the standard input.

The following example program prints *hello world* on the standard output:

```
awk'BEGIN {
        print "hello, world"
        exit

      }
```

prints

hello, world

Recall that the word "BEGIN" is a special pattern indicating that the action following in braces is run before any data is read. The words "print" and "exit" are both discussed in later sections.

You could also create a file containing:

```
BEGIN {
                print "hello, world"
                exit
                }
```

if this file is named *awkprog*, enter the command

awk —f *awkprog*

This has the same effect as the first procedure.

Input: Records and Fields

The **awk** reads its input one record at a time unless you change it. A record is a sequence of characters from the input ending with a newline character or with an end of file. Thus, a record is a line of input. The **awk** program reads in characters until it encounters a newline or end of file. The string of characters, thus read, is assigned to the variable **$0**. You can change the character that indicates the end of a record by assigning a new character to the special variable **RS** (the record separator). Assignment of values to variables and these special variables such as **RS** are discussed later.

Once **awk** reads in a record, it splits the record into *fields*. A field is a string of characters separated by blanks or tabs, unless you specify otherwise. You can change field separators from blanks or tabs to whatever characters you choose, in the same way that record separators are changed. That is, the special variable FS is assigned a different value.

As an example, let us suppose that the file *countries* contains the area in thousands of square miles, the population is millions, and the continent for the ten largest countries in the world.

Sample input file *countries*:

Russia	8650	262	Asia
Canada	3862	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	68	14	Australia
India	1269	637	Asia
Argentina	72	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa

The white spaces are tabs in the original input, and a single blank separates North and South from America. We use this data as the input for many of the **awk** programs in this guide, since it is typical of the type of material that **awk** is best at processing. This material includes a mixture of words and numbers separated into fields, or columns separated by blanks and tabs.

Each of these lines has either tabs separate the fields. This is what **awk** assumes unless told otherwise. In the above example, the first record is:

Russia 8650 262 Asia

When this record is read by **awk**, it is assigned to the variable **$0**. If you want to refer to this entire record, it is done through the variable, **$0**. For example, the following input prints the entire record:

```
{print $0}
```

Fields within a record are assigned to the variables $1, $2, $3, and so forth; that is, the first field of the present record is referred to as $1 by the **awk** program. The second field of the present record is referred to as $2 by the **awk** program. The *i*th field of the present record is referred to as $*i* by the **awk** program. Thus, in the above example of the file countries, in the first record;

$1 is equal to the string "Russia"
$2 is equal to the integer 8650
$3 is equal to the integer 262
$4 is equal to the string "Asia"
$5 is equal to the null string

To print the continent, followed by the name of the country, followed by its population, use the following **awk** script:

```
{print $4, $1, $3}
```

Note that **awk** does not require type declarations.

Input: From the Command Line

You can assign values to variables from within an **awk** program. Because you do not declare types of variables, a variable is created simply by referring to it. An example of assigning a value to a variable is:

```
x = 5
```

This statement in an **awk** program assigns the value 5 to the variable x. It is also possible to assign values to variables from the command line. This provides another way to supply input values to **awk** programs.

For example:

```
awk' {print x}' x = 5 —
```

This prints the value 5 on the standard output for each record from the standard input. The minus sign at the end of this command is necessary to indicate that input is comming from the standard input instead of a file called x = 5. Similarly if the input comes from a file named *file*, the command is:

```
awk'{print x}' file
```

You cannot assign values to variables used in the BEGIN section in this way.

If you must change the record separator and the field separator, you can do so from the command line. For example:

awk —f awk.program RS = ":" *file*

Here, the record separator is changed to a colon (:). This causes your program in the file *awk.program* to run with records separated by the colon instead of the newline character, and with input coming from *file*. You can also change the field separator from the command line.

Another way exists to change the field separator from the command line. There is a separate option **—Fx** that is placed directly after the command **awk**. This changes the field separator from blank or tab to the character x. For example:

awk —F: —f *awk.program file*

This changes the field separator FS to a colon (:). Note that if the field separator is specifically set to a tab, (that is, with the **—F** option or by making a direct assignment to FS) blanks are recognized by **awk** as separating fields. However, even if the field separator is specifically set to a blank, tabs are still recognized by **awk** as separating fields.

Try this example problem to exercise what you have learned in this section. Using the input file *countries* write an **awk** script that prints the name of a country followed by the continent that it is on. Do this in such a way that continents composed of two words (for example, North America) are processed as only one field and not two.

Output: Printing

An action may have no pattern; in this case, the action is executed for all lines as in the simple printing program:

{print}

This is one of the simplest actions performed by **awk**. It prints each line of the input to the output. It also prints one or more fields from each line. For instance, using the file *countries*:

awk'{ **print $1, $3** }' *countries*

This prints the name of the country and the population:

```
Russia 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 14
Sudan 19
Algeria 18
```

Note that the use of a semicolon at the end of statements in **awk** programs is optional. **Awk** accepts both:

```
{print $1}

{print $1;}
```

Both forms mean the same thing. If you want to put two **awk** statements on the same line of an **awk** script, the semicolon is necessary. For example, the following semicolon is necessary if you want the number 5 printed:

```
{x=5; print x}
```

Parentheses are also optional with the print statement.

```
print $3,$2
```

It is the same as:

```
print ($3, $2)
```

Items separated by a comma in a print statement are separated by the current output field separators (normally spaces, even though the input is separated by tabs) when printed. The **OFS** is another special variable that you can change. These special variables are summarized in a later section.

Try another example problem.

Using the input file, *countries*, print the continent followed, by the country followed by the population, for each input record. Then pipe the output to the UTek operating system command **sort** so that all countries from a given continent are printed together.

Print also prints strings directly from your programs with the **awk** script:

```
{print "hello, world"}
```

As another example exercise, print a header to the output of the previous problem. The header says "Population of Largest Countries", and is followed by headers to the other columns, for example, Country or Population.

As we have already seen, **awk** makes available a number of special variables with useful values; for example, **FS** and **RS**. We now introduce another special variable in the next example. **NR** and **NF** are both integers that contain the number of the present record and the number of fields in the present record, respectively. So this example prints each record number and the number of fields in each record, followed by the record itself:

```
{print NR, NF, $0}
```

Using this program on the file, *countries* yields:

```
14 Russia      8650   262   Asia
25 Canada      3852   24    North America
34 China       3692   866   Asia
45 USA         3615   219   North America
55 Brazil      3286   116   South America
64 Australia   2986   14    Australia
74 India       1269   637   Asia
85 Argentina   1072   26    South America
94 Sudan       968    19    Africa
104 Algeria    920    18    Africa
```

This **awk** program

```
{print NR, $1}
```

prints:

```
1 Russia
2 Canada
3 China
4 USA
5 Brazil
6 Australia
7 India
8 Argentina
9 Sudan
10 Algeria
```

This is an easy way to supply sequence numbers to a list. Print, by itself, prints the input record. This prints the empty line:

```
print " "
```

Awk also provides the statement *printf* so that you can format output as desired. Print uses the default format "%.6g" for each variable printed.

printf format, expr, expr, ...

This formats the expressions in the list according to the specification in the string, format, and prints them. The format statement is exactly that of the printf in the C library. For example, this prints $1 as a string of 10 characters (right justified).

```
{printf"%10s %6d0, $1, $2, $3}
```

The second and third fields (6–digit numbers) make a neatly columned table.

However, without the comma in the print statement it produces the output:

helloworld

To get a comma in the output, you can either insert it in the print statement as in this case:

```
{ x="hello";y="world"
print x","y
}
```

You can also insert a comma by changing OFS in a BEGIN section:

```
BEGIN {OFS=","}
{x="hello";y="world"
print x,y
}
```

Both of these last two scripts print:

hello, world

Note that the output field separator is not used when $0 is printed.

Output: To Different Files

The UTek operating system shell allows you to redirect standard output to a file. The **awk** program also lets you direct output to many different files from within your **awk** program. For example, with our input file *countries* we want to print all the data from countries of Asia in a file called ASIA, ll the data from countries in Africa in a file called AFRICA, and so on. This is done with the following **awk** program:

```
{if ($4 = = "Asia") print > "ASIA "
 if ($4 = = "Europe") print > "EUROPE"
 if ($4 = = "North") print > "NORTH_AMERICA"
 if ($4 = = "South") print > "SOUTH_AMERICA"
 if ($4 = = "Australia") print > "AUSTRALIA"
 if ($4 = = "Africa") print > "AFRICA "
}
```

The control flow statements are discussed later.

In general, you can direct output into a file after a *print* or a *printf* statement by using a statement of the form:

```
print > "FILE"
```

where FILE is the name of the file receiving the data, and the *print* statement can have any legal arguments to it.

Notice that the filenames are quoted. Without quotes, the filenames are treated as uninitialized variables and all output then goes to the same file.

If > is replaced by >>, output is appended to the file rather than overwriting it.

Note that there is an upper limit to the number of files that are written in this way. At present it is NOFILE —3.

Output: To Pipes

You can also direct printing into a pipe instead of a file. For example:

```
{
print $1 ¦ "sort"
}
```

This takes the first field of each input record, sorts these fields, and then prints them. The command in quotes is any UTek command.

As an example exercise, write an **awk** script that uses the input file to:

- List countries that were used previously
- Print the name of the countries
- Print the population of each country
- Sort the data so that countries with the largest population appear first
- Mail the resulting list to yourself

Another example of using a pipe for output is the following idiom that guarantees that its output always goes to your terminal:

```
print ... ¦ "cat –u /dev/tty"
```

Only one output statement to a pipe is permitted in an **awk** program. In all output statements involving the redirection of output, the files or pipes are identified by their names, but they are created and opened only once in the entire run.

Comments

You can place comments in **awk** programs; they begin with the character # and end with the end of the line. For example:

```
print x, Y    #this is a comment
```

Patterns

A pattern in front of an action acts as a selector that determines if the action is to be executed. A variety of expressions are used as patterns:

- Regular expressions
- Arithmetic relational expressions
- String valued expressions
- Combinations of these

BEGIN and END

The special pattern, BEGIN, matches the beginning of the input before the first record is read. The pattern END matches the end of the input after the last line is processed. BEGIN and END thus provide a way to gain control before and after processing for initialization and wrapping up.

As you have seen, you can use a BEGIN to put column headings on the output:

```
BEGIN {print "Country", "Area",
 "Population", "Continent"}
  {print}
```

This produces the output:

Russia	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	2968	14	Australia
India	1269	637	Asia
Argentina	1072	26	South America
Sudan	968	18	Africa

Formatting is not very good here; *printf* would do a better job.

Recall also that the BEGIN section is a good place to change special variables such as FS or RS:

```
BEGIN { FS = " "
   print "Countries", "Area", "
Population", "Continent"
   }
   {print}
END    {print "The number of records is" ,NR}
```

In this example program, FS is set to a tab in the BEGIN section and as a result all records have exactly four fields.

Note that if BEGIN is present it is the first pattern, END is the last if it is used.

Relational Expressions

An **awk** pattern is any expression involving comparisons between strings of characters or numbers. For example, if you want to print only countries with more than 100 million population enter:

```
$3 > 100
```

This tiny **awk** program is a pattern without an action so it prints each line whose third field is greater than 100:

Russia	8650	262	Asia
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
India	1269	637	Asia

To print the names of the countries that are in Asia type:

```
$4 = = "Asia" {print $1}
```

This produces the output:

```
Russia
China
India
```

The conditions tested are <, <=, ==, !=, >=, and >. In such relational tests, if both operands are numeric, a numerical comparison is made. Otherwise, the operands are compared as strings. Thus:

```
$1 >= "s"
```

This selects lines that begin with S, T, U, and so forth. In the case of countries:

USA	3615	219	North America
Sudan	968	19	Africa

In the absence of other information, fields are treated as strings, so this program compares the first and fourth fields as strings of characters:

```
$1 == $4
```

It prints the single line:

Australia	2968	14	Australia

If fields appear as numbers, the comparisons are done numerically.

Regular Expressions

Awk provides powerful capabilities for searching for strings of characters. These are regular expressions. The simplest regular expression is a literal string of characters enclosed in slashes.

```
/Asia/
```

This is a complete **awk** program that prints all lines containing any occurrence of the name Asia. If a line contains Asia as part of a larger word like Asiatic, it is also printed.

Awk regular expressions include those in:

- the text editor **ed**
- the pattern finder **grep**

For example, you print all lines that begin with A using:

```
/^A/
```

Or all lines that begin with A, B, or C using:

```
/^[ABC]/
```

Or all lines that end with "ia" using:

```
/ia$/
```

In general, the caret (^) indicates the beginning of a line. The dollar sign ($) indicates the end of the line, and characters enclosed in braces match any one of the characters enclosed. In addition, **awk** allows parentheses for grouping, the pipe (|) for alternatives, + for "one or more" occurrences, and ? for "zero or one" occurrences. For example, this prints all the records that contain either an x or a y:

```
/x|y/ {print}
```

This program prints all records that contain an a, followed by one or more x's, followed by a b:

```
/ax+b/ {print}
```

This program prints all records that contain an a, followed by zero or one x's, followed by a b:

```
/ax?b/  {print}
```

The characters period (.) and asterisk (*) have the same meaning as they have in **ed**. A period can stand for any character and an asterisk means zero or more occurrences of the character preceding it. For example:

```
/a.b/
```

This matches any record that contains an a, followed by any character, followed by a b. That is, the record must contain an a and a b separated by exactly one character.

Just as in **ed**, you can turn off the special meaning of these metacharacters by preceding them with a backslash. An example of this is the pattern:

```
/\/.*\//
```

This matches any string of characters enclosed in slashes.

You can also specify that any field or variable matches a regular expression (or does not match it) by using the operators or !'. For example, with the input file *countries* the program:

```
$1 ~ /ia$/  {print $1}
```

This prints all countries whose name ends in "ia":

```
Russia
Australia
India
Algeria
```

Combinations of Patterns

A pattern is made up of similar patterns combined with the operators ¦¦ (OR), && (AND), ! (NOT), and parentheses. For example:

```
$2 >= 3000 && $3 >= 100
```

This selects lines where both area AND population are large. For example:

Russia	8650	262	Asia
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America

This program selects lines with Asia or Africa as the fourth field:

```
$4 = = "Asia"  ¦¦ $4 = = "Africa"
```

An alternate way to write this last expression is with a regular expression:

```
$1 ~ /^(Asia¦Africa)$/
```

the operators && and ¦¦ guarantee that their operands are evaluated from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

The pattern that selects an action can consist of two patterns separated by a comma. For example:

```
pattern1, pattern2 { . . . }
```

In this case, the action is performed for each line between an occurrence of pattern1 and the next occurrence of pattern2 (inclusive). As an example with no action:

```
/Canada/,/Brazil/
```

This prints all lines between the one containing Canada and the line containing Brazil. For example:

Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America

This program does the action for lines 2 through 5 of the input:

```
NR = = 2, NR = = 5 { . . . }
```

Different types of patterns are mixed as in:

```
/Canada/,$4 = = "Africa"
```

This prints all lines from the first line containing "Canada" up to and including the next record whose fourth field is "Africa".

Note that patterns in this form occur outside of the action parts of the **awk** programs (outside of the braces that define **awk** actions). If you need to check patterns inside an **awk** action (inside the braces) use a control flow statement such as **if** or **while**. Control flow statements are discussed in the topic *Built-in Functions*.

Actions

An **awk** action is a sequence of action statements separated by newlines or semicolons. These action statements do a variety of bookkeeping and string manipulating tasks.

Variables, Expressions, and Assignments

The **awk** program provides the ability to do arithmetic and to store the results in variables for later use in the program. However, variables can also store strings of characters. You cannot do arithmetic on character strings, but you can stick them together and pull them apart as shown. As an example, consider printing the population density for each country in the file *countries*:

```
{print $1, (1000000 * $3)/($2 * 1000) }
```

(Recall that in this file the population is in millions and the area in thousands.) The result is population density in people per square mile:

Russia	30.289
Canada	6.23053
China	234.561
USA	60.5809
Brazil	35.3013
Australia	4.71698
India	501.97
Argentina	24.2537
Sudan	19.6281
Algeria	19.5652

The formatting is bad; so using *printf* instead gives the program:

```
{printf" %10s %6.1f0,$1, (1000000 * $3)/($2 * 1000) }
```

This produces the output:

Russia	30.3
Canada	6.2
China	234.6
USA	60.6
Brazil	35.3
Australia	4.7
India	502.0
Argentina	24.3
Sudan	19.6
Algeria	19.6

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, / and % (mod or remainder).

To compute the total population and number of countries from Asia write:

```
{pop += $3; ++n}
END {print "total population of", n, "Asian countries is", pop}
```

This produces "total population of three Asian countries is 1765."

Indeed, these operators, ++, —, −=, /e, *=, +=, and %= are available in **awk** as they are in C. Operator x += y has the same effect as x = x + y, but += is shorter and runs faster. The same is true of the ++ operator; it adds one to the value of a variable. The increment operators ++ and — are used as prefix or postfix operators. These operators are also used in expressions.

Initialization of Variables

In the previous example, we did not initialize pop or n; yet, everything worked properly. This is because variables by default are initialized to the null string, which has a numerical value of 0. This eliminates the need for most initialization of variables in BEGIN sections. We can use default initialization to advantage in this program that finds the country with the largest population:

```
maxpop < $3 {
   maxpop = $3
   country = $1
}
END  {print country, maxpop}
```

This produces the output:

```
China 866
```

Field Variables

Fields in **awk** share essentially all of the properties of variables. they are used in arithmetic and string operations. They can be assigned to and initialized to the null string. Thus, divide the second field by 1000 to convert the area to millions of square miles using:

```
{ $2 /= 1000; print }
```

You can process two fields into a third using:

```
BEGIN {FS= "      "}
  {$4 = 1000 * $3 / $2; print}
```

Or you can assign strings to a field:

```
/USA/  {$1 = "United States" ; print }
```

This replaces USA by United States and prints the affected line:

```
United States 3615 219 North America
```

Fields are accessed by expressions; thus, $NF is the last field and $(NF–1) is the second to last. Note that the parentheses are needed since $NF–1 is 1 less than the values in the last field.

String Concatenation

Strings are concatenated by writing them one after the other as in the following example:

```
{x = "hello"
 x = x ", world"
  print x
}
```

This prints:

```
hello, world
```

With input from the file *countries*, the following program:

```
/A/    { s = s " " $1 }
END    { print s }
```

This prints:

```
Australia Argentina Algeria
```

Variables, string expressions, and numeric expressions can appear in concatenations; the numeric expressions are treated as strings in this case.

Special Variables

Some variables in **awk** have special meanings. These are detailed here and the complete list is given.

NR	Number of the current record.
NF	Number of fields in the current record.
FS	Input field separator, by default a blank or tab.
RS	Input record separator, by default it is set to the newline character.
$i	The *i*th input field of the current record.
$0	The entire current input record.
OFS	Output field separator, by default it is set to a blank.
ORS	Output record separator, by default it is set to the newline character.
OFMT	The format for printing numbers, with the print statement, be default is "%.6g".
FILENAME	The name of the input file currently being read. This is useful because **awk** commands are typically of the form: **awk –f** *program file1 file1 file3* . . . .

Type

Variables (and fields) take on numeric or string values according to context. For example, pop is presumably a number:

```
pop + = $3
```

While in this example, country is a string:

```
country = $1
```

In this example, the type of maxpop depends on the data found in $3, which is determined when the program is run:

```
maxpop < $3
```

In general, each variable and field is potentially a string or a number or both at any time. When a variable is set by the assignment, its type is set to that of expr:

```
v = expr
```

(Assignment also includes + =, + +, −=, and so on.) An arithmetic expression is of the type, "number"; a concatenation of strings is of type "string". If the assignment is a simply copy as in:

```
v1 = v2
```

then the type of v1 becomes that of v2.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to strings if necessary and the comparison is made on strings.

The type of any expression is coerced to numeric by subterfuges such as:

```
expr + 0
```

Or you can coerce the type to string by:

```
expr ""
```

This last expression is string concatenated with the null string.

Arrays

As well as ordinary variables, **awk** provides one–dimensional arrays. Array elements are not declared; they come into existence by being mentioned. Subscripts can have *any* non–null value including non–numeric strings.

As an example of a conventional numeric subscript, this statement assigns the current input line to the NRth element of the array x:

```
x[NR] = $0
```

In fact, it is possible in principle, although slow, to process the entire input in a random order with the following **awk** program:

```
  { x[NR] = $0 }
END  { ... program ...}
```

The first line of this program records each input line into the array x. In particular, the following program:

```
{ x[NR] = $1}
```

when run on the file *countries* produces an array of elements with:

```
x[1] = "Russia"
x[2] = "Canada"
x[3] = "China"
...
```

Arrays are also indexed by non–numeric values that give **awk** a capability rather like the associative memory of Snobol tables. For example, write:

```
/Asia/  {pop[" Asia"] + = $3 }
/Africa {pop[Africa] + = $3 }
END   print " Asia = " pop["Asia"], "Africa = "pop["Africa"] }
```

This produces the output:

```
Asia 1765 Africa = 37
```

Notice the concatenation. Also, any expression can be used as a subscript in an array reference. So this example uses the first field of a line (as a string) to index the array area:

```
area[$1] = $2
```

Built–in Functions

The function:

```
length
```

is provided by **awk** to compute the length of a string of characters. The following program prints each record preceded by its length:

```
{print length, $0}
```

In this case the variable length means length($0), the length of the present record. In general, length(x) returns the length of x as a string. With input from the file *countries*, the following **awk** program prints the longest country name:

```
length($1) > max  { max = length($1); name = $1 }
END   { print name }
```

The function:

```
split
```

written as split (s, array) assigns the fields of the string "s" to successive elements of the array.

For example:

```
split)"Now is the time", w)
```

This assigns the value "now" to w[1], "is" to w[2], and so on. It is possible to have a character other than a blank as the separator for the elements of w. For this, use split with three elements:

```
n = split(s,array, sep)
```

This splits the string s into array[1],... array[n]. The number of elements found is returned as the value of split. If the *sep* argument is present, its first character is used as the field separator; otherwise, FS is used. This is useful if in the middle of an **awk** script you must change the record separator for one record.

Also provided by awk are the math functions:

```
sqrt
log
exp
int
```

They provide the square root function, the base e logarithm function, exponential and integral part functions. This last function returns the greatest integer less than or equal to its argument. These functions are the same as those of the C library, so they have the same return on error as those in the C library. See the *UTek Command Reference* for information on C library return values.

The subtract function is:

```
substr
```

The form substr(s,m,n) produces the substring of *s* that begins at position *m* and is at most *n* characters long. If the third argument is omitted, the substring goes to the end of *s*. For example, you can abbreviate the country names in the file *countries*:

```
{ $1 = substr($1,1,3); print }
```

This produces the output:

```
Rus   8650   262   Asia
Can   3852   24    North America
Chi   3692   866   Asia
USA   3615   219   North America
Bra   3286   116   South America
Aus   2968   14    Australia
Ind   1269   637   Asia
Arg   1072   26    South America
Sud   968    19    Africa
Alg   920    18    Africa
```

If *s* is a number, substr uses its printed image.

The function:

index:

in the form index(*s1,s2*) it returns the leftmost position where the string*s2* occurs in *s1*, or zero if *s2* does not occur in *s1*.

This function:

sprintf

formats expressions as the printf statement does, but assigns the resulting expression to a variable instead of sending the results to the standard output. For example:

```
x = sprintf("%10s %6d ", $1, $2 )
```

This sets x to the string produced by formatting the values of $1 and $2. The x is then used in subsequent computations.

The function:

getline

immediately reads the next input record. Fields NR and $0 are all set, but control is left at exactly the same spot in the **awk** program. Getline returns 0 for the end of a file and a 1 for a normal record.

Control Flow

The **awk** provides the basic control flow statements:

- **if–else**
- **while**
- **for**

These statements are grouped as in the C language.

The **if** statement is used as follows:

if *(condition) statement1* **else** *statement2*

The *condition* is evaluated, and if it is true *statement1* is executed; otherwise, *statement2* is executed. The **else** part is optional. Several statements enclosed in braces are treated as a single statement. You can rewrite the maximum population computation for the file *countries* using an **if** statement:

```
{  if (maxpop < $3) {
        maxpop= $3
        country= $1
        }    }
END    {print country, maxpop }
```

Awk also provides a **while** statment:

while *(condition) statement*

The *condition* is evaluated; if it is true, the statement is executed. The condition is evaluated again, and if true, the statement is executed. The cycle repeats as long as the condition is true. For example, the following prints all input fields, one per line:

```
{  i = 1
   while (i <= NF) {
      print $i
      ++i
      }
}
```

Another example is the Euclidean algorithm for finding the greatest common divisor of $1 and $2:

```
{printf"the greatest common divisor of" $1" and ",$2,"is"
while ($1 =!$2) {
     if ($1>$2) $1 = $1 - $2
     else   $2 = $2 - $1
     }
printf $1 " 0
}
```

The **for** statement is like that of C:

for *(expression1; condition; expression2) statement*

This has the same effect as:

```
expression1
while (condition) {
    statement
    expression2
    }
```

So you can write another **awk** program that prints all input fields one per line:

```
{  for (i=1; i <= NF; i++)
    print $i
}
```

This is an alternate form of the **or** statement. It is suited for accessing the elements of an associative array, as in **awk**:

for (*i* in array) *statement*

This executes *statement* with the variable *i* set in turn to each subscript of the array. The subscripts are each accessed once but in random order. Chaos ensues if the variable *i* is altered or if any new elements are created within the loop. For example, you could use the **for** statement to print the record number, followed by the record of all input records after the main program is executed:

```
    { x[NR] = $0 }
END   { for(i in x) { print i, x[i] }
```

A more practical example is the following use of strings to index arrays to add the populations of countries by continents:

```
BEGIN  {FS=""}
    {population[$4] =+ $3}
END   {for(i in population)
        print i, population[i]
}
```

In this program, the body of the **for** loop is executed for *i* equal to setting the string "Asia", then for *i* equal to the string "North America", and so forth until all the possible values of *i* are exhausted; that is, until all the strings of names of countries are used. Note, however, the order in which the loops are executed is not specified. If the loop associated with "Canada" is executed before the loop associated with the string "Russia", such a program produces the output:

```
South America 26
Africa 16
Asia 637
Australia 14
North America 219
```

Note that the expression in the condition part of an **if, while,** or, **for** statement can include relational operators like <, <=, >, >=, ==, and !=. It can include regular expressions that are used with the matching operators ~ and !~; it can include the logical operators ||, &&, and !; and it also includes parentheses for grouping.

The **break** statement, if it occurs within a **while** or **for** loop, causes an immediate exit from the **while** or **for** loop.

The **continue** statement, when it occurs within a **while** or **for** loop, causes the next iteration of the loop to begin.

The **next** statement in an **awk** program causes **awk** to skip immediately to the next record and begin scanning patterns from the top of the program. (Note the difference between getline and next. Getline does not skip to the top of the **awk** program.)

If an **exit** statement occurs in the BEGIN section of an **awk** program, the program stops execution and the END section is not executed.

An **exit** that occurs in the main body of the **awk** program causes execution of the main body of the **awk** program to stop. No more records are read, and the END section is executed. An **exit** in the END section causes execution to terminate at that point.

Report Generation

The control flow statements in the last discussion are especially useful when you use **awk** as a report generator. **Awk** is useful for tabulating, summarizing, and formatting information. We have seen an example of **awk** in the tabulation of populations. Here is another example of this capability. Suppose you have a file called *prog.usage* that contains lines of three fields; name, program and usage:

```
Smith    draw    3
Brown    eqn     1
Jones    nroff   4
Smith    nroff   1
Jones    spell   5
Brown    spell   9
Smith    draw    6
```

The first line indicates that Smith used the draw program three times. If you want to create a program that has the total usage o each program along with the names in alphabetical order and the total usage, use the following program, called *list.a*:

```
    { use[$1 "" $2] + = $3}
END  {for (np in use)
        print np "       " use[np] ¦ "sort +0 +2nr" }
```

This program produces the following output when *prog.usage* is used as the input file:

```
Brown    eqn     1
Brown    spell   0
Jones    nroff   4
Jones    spell   5
Smith    draw    9
Smith    nroff   1
```

To format the previous output so that each name is printed only once, pipe the output of the previous **awk** program into the following program, called *format.a*:

```
{  if ($1 != prev) }
       print $1 ":"
       rev = $1
       }
    print " " $2" " $3
}
```

The variable *prev* prints the unique values of the first field. The command:

awk –f *list.a prog.usage* ¦ **awk –f** *format.a*

This gives the output:

```
Brown:
        eqn     1
        spell   9
Jones:
        nroff   4
        spell   5
Smith:
        draw    9
        nroff   1
```

It is often useful to combine different **awk** scripts and other shell commands, such as **sort**, as we did in the last program.

Cooperation with the Shell

Normally, an **awk** program is either contained in a file or enclosed within single quotes:

awk '{print $1}' ...

Awk uses many of the same characters that the shell does, such as $ and the double quote. Surrounding the program with single quotes ensures that the shell passes the **awk** program to **awk** intact.

Consider writing an **awk** program to print the *n*th field, where *n* is a parameter determined when the program is run. That is, we want a program called *field* such that:

```
field n
```

runs the **awk** program:

```
awk '{print $n}'
```

How does the value of n get into the **awk** program?

There are several ways to do this. One is to define field as follows:

```
awk '{print $'$1'}'
```

Spaces are critical here: as written, there is only one argument, even though there are two sets of quotes. The $1 is outside the quotes, visible to the shell, and therefore substituted properly when field is invoked.

Another way to do this job relies on the fact that the shell substituted for $ parameters within double quotes:

```
awk " {print $1} "
```

Here the trick is to protect the first $ with a \\; the $1 is again replaced by the number when field is invoked.

Miscellaneous Hints

You can simulate the effect of multi-dimensional arrays by creating your own subscripts. For example:

```
for (i = 1; i <= 10; i++)
    for (j = 1; j <= 10; j++)
      mult[i "," j] =...
```

This creates an array whose subscripts have the form i,j; that is, 1,1; 1,2; and so forth, thus simulating a two-dimensional array.

Interactive Desk Calculator (DC)

Introduction

The DC program is an interactive desk calculator program implemented on the UTek operating system to do arbitrary-precision integer arithmetic. It has provisions for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated by DC is limited only by available core storage. On typical implementations of the UTek system, the size of numbers that can be handled varies from several hundred on the smallest systems to several thousand on the largest.

The DC program works like a stacking calculator using reverse Polish notation. Ordinarily, DC operates on decimal integers; but an input base, output base, and a number of fractional digits to be maintained can be specified.

A language called BC has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles the output which is interpreted for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a pushdown stack. The DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end; then it is taken from the standard input.

DC Commands

Any number of commands are permitted on a line. Blanks and newline characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number (such as 244)

The value of a number is pushed onto the stack. A number is an unbroken string of digits 0 through 9 and uppercase letters A through F (treated as digits with values 10 through 15, respectively). The number may be preceded by an underscore (_) to input a negative number and numbers may contain decimal points.

The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^) by using

+ – * / % ^

The two entries are popped off the stack, and the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. An exponent must not have any digits after the decimal point.

In the following example, the top of the main stack is popped and stored in a register named *x* (where *x* may be any character):

s*x*

If **s** is uppercase, *x* is treated as a stack; and the value is pushed onto it. Any character, even blank or newline, is a valid register name.

The value of register *x* is pushed onto the stack. Register *x* is not altered. If the **l** in the following example is uppercase, register *x* is treated as a stack, and its top value is popped onto the main stack:

l*x*

All registers start with empty value which is treated as a zero by the command **l** and is treated as an error by the command **L**.

The following characters perform the stated tasks:

d	The top value on the stack is duplicated.
p	The top value on the stack is printed. The top value remains unchanged.
f	All values on the stack are printed.
x	Treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.
[...]	Puts the bracketed character string onto the top of the stack.
q	Exits the program. If executing a string, the recursion level is popped by two. If **q** is uppercase, the top value on the stack is popped; and the string execution level is popped by that value.
$<x>x=x!<x!>x!=x$	The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.
v	Replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer.
!	Interprets the rest of the line as a UTek software command. Control returns to DC when the command terminates.
c	All values on the stack are popped; the stack becomes empty.

i The top value of the stack is popped and used as the number radix for further input. If **i** is uppercase, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o The top value on the stack is popped and used as the number radix for further output. If **o** is uppercase, the value of the output base is pushed onto the stack.

k The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is uppercase, the value of the scale factor is pushed onto the stack.

z The value of the stack level is pushed onto the stack.

? A line of input is taken from the input source (usually the console) and executed.

Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100, stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0 to 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100s complement notation, which is analogous to twos complement notation for binary numbers. The high-order digit of a negative number is always –1 and all other digits are in the range 0 to 99. The digit preceding the high-order –1 digit is never a 99. The representation of –157 is 43,98,–1. This is called the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when it is convenient.

An additional byte is stored with each number beyond the high-order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high-order digit. The value of this extra byte is called the *scale factor* of the number.

The Allocator

The DC program uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of two. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Leftover strings are put on the free list. If there are no larger strings, the allocator tries to combine smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long.

If a string of the proper length cannot be found, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If the allocator runs out of headers at any time in the process of trying to allocate a string, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end of string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. The **scale** register limits the number of decimal places retained in arithmetic computations. The **scale** register can be set to the number on the top of the stack truncated to an integer with the **k** command. The **K** command can be used to push the value of scale on the stack. The value of **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations includes the exact effect of **scale** on computations.

Addition and Subtraction

The scale of the two numbers are compared, and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and preceding as in addition.

The addition is performed digit by digit from the low–order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers, replacing the high–order configuration 99,–1 by the digit –1. In any case, digits that are not in the range 0 to 99 must be brought into that range, propagating any carries or borrows that result.

Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly follows the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low–order digit. The intermediate products are accumulated into a partial sum that becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

Division

The scales are removed from the two operands. Zeros are appended, or digits are removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise, the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high–order) digit of the quotient. If it turns out to be one unit too low, the next trial quotient is larger than 99; and this is adjusted at the end of the process. The trial digit is multiplied by the divisor, the result subtracted from the dividend, and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form of the operands.

Remainder

The division routine is called, and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

Square Root

The scale is removed from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand. The method used to compute the square root is Newton's method with successive approximations by the rule.

The initial guess is found by taking the integer square root of the top two digits.

Exponentiation

Only exponents with 0 scale factor are handled. If the exponent is 0, then the result is 1. If the exponent is negative, then it is made positive; and the base is divided into 1. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared, and the result is obtained as a product of those powers of the base that correspond to the positions of the one–bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with an underscore (_). The hexadecimal digits A through F correspond to the numbers 10 through 15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base (**ibase**) is initialized to 10 (decimal) but can, for example, be changed to 8 or 16 for octal, or hexadecimal to decimal conversions. The command **I** pushes the value of the input base on the stack.

Output Commands

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers are output by typing the command **f**. The **o** command is used to change the output base (**obase**). This command uses the top of the stack truncated to an integer as the base for all further output. The output base is initialized to 10 (decimal). It works correctly for any base. The command **O** pushes the value of the output base of the stack.

Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no affect on arithmetic computations. Large numbers are output with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 are used for decimal–octal or decimal–hexadecimal conversions.

Internal Registers

Numbers or strings can be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register *x*. The *x* can be any character. The command **l***x* puts the contents of register *x* on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

Stack Commands

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack onto the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

Subroutine Definitions and Calls

Enclosing a string in brackets (**[ ]**) pushes the ASCII string on the stack. The **q** command quits or (in executing a string) pops the recursion levels by two.

Internal Registers — Programming DC

The load and store commands, together with brackets (**[ ]**) to store strings, the **x** command to execute, and the testing commands (**<**, **>**, **=**, **!<**, **!>**, **!=**), can be used to program DC. The **x** command assumes the top of the stack is a string of DC commands and executes it. The testing commands compare the top two elements on the stack and, if the relation holds, execute the register that follows the relation. For example, to print the numbers 0 through 9, input the following:

```
[lip1+ si li10>a]sa
0si lax
```

Pushdown Registers and Arrays

These commands are designed for use by a compiler, not directly by programmers. They involve pushdown registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **L***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers but not as pushdown stacks. The command **l** does not affect the top of the register stack, but **s** destroys what was there before.

The commands to work on arrays are : and ;. The command :*x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. The command ;*x* loads the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

Miscellaneous Commands

The command ! interprets the rest of the line as a UTek software command and passes it to the UTek operating system to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

Design Choices

The real reason for the use of a dynamic storage allocator is that a general purpose program can be used for a variety of other tasks. The allocator has some value for input and for compiling (such as the bracket **[...]** commands) where it cannot be known in advance how long a string will be. The result is, that at a modest cost in execution time:

- All considerations of string allocation and sizes of strings are removed from the remainder of the program
- Debugging is made easier
- The allocation method used wastes approximately 25 percent of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and, at the cost of five percent in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack–type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases is to provide an understandable means of continuing after a change of base or scale (when numbers had already been entered). An earlier implementation which had global notations of scale and base did not work out well. If the value of **scale** is interpreted in the current input or output base, then a change of base or scale in the midst of a computation causes great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other

operations. The value of **scale** is not used for any essential purpose by any part of the program. It is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The rationale for the choices for the scales of the results of arithmetic is that in no case should any significant digits be thrown away if, on appearances, you actually wanted them. So, if you want to add the numbers 1.5 and 3.517, it seemed reasonable to give them the result 5.017 without requiring to unnecessarily specify rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands. It seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless you asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places, and there is simply no way to guess how may places you want. In this case only, you must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

Arbitrary Precision Desk Calculator Language (BC)

Introduction

The arbitrary precision desk calculator language (BC) is a language and compiler for doing arbitrary precision arithmetic under the UTek operating system. The output of the compiler is interpreted and executed by a collection of routines that can input, output, and do arithmetic on infinitely large integers and on scaled fixed-point numbers. These routines are based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The BC language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution. A small collection of library functions is also available, including **sin, cos, arctan, log, exponential**, and **Bessel** functions of integer order.

The BC compiler was written to make conveniently available a collection of routines (called DC) that are capable of doing arithmetic on integers of arbitrary size. The compiler is not intended to provide a complete programming language. It is a minimal language facility.

Some of the uses of this compiler are:

- Compile large integers
- Compute accurately to many decimal places
- Convert numbers from one base to another base

There is a scaling provision that permits the use of decimal point notation. Provision is also made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal eight.

The actual limit on the number of digits that can be handled depends on the amount of core storage available. This is possible even on the smallest versions of the UTek operating system.

The syntax of BC is very similar to that of the C language. This enables users who are familiar with C language to easily work with BC.

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the addition of two numbers (with the + operator) such as

142857 + 285714

the program responds immediately with the sum

428571.

The operators **–**, *****, **/**, **%**, and **^** can also be used. They indicate subtraction, multiplication, division, remaindering, and integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the **unary** minus sign). The following expression is interpreted to mean that –3 is to be added to 7:

7+–3

More complex expressions with several operators and with parentheses are interpreted in the following order of precedence: power (^) is interpreted first; then *****, **%**, and **/** are read; and finally, + and – are interpreted. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right.

a^b^c
and
a^(b^c)

are equivalent, as are the following two expressions:

a*b*c
and
(a*b)*c

However, BC shares with FORTRAN and C language the undesirable convention that

a/b*c
is equivalent to
(a/b)*c.

Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way. The following statement has the effect of increasing by three the value of the contents of the register named *x*:

x=*x*+**3**

When, as in this case, the outermost operator is an equal sign (=), the assignment is preformed; but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (see the part on SCALING). Entering the lines

```
x=sqrt(191)
x
```

produces the printed result

```
13
```

Bases

There are two special internal quantities; **ibase** (input base) and **obase** (output base). The contents of **ibase**, initially set to 10 (decimal), determines the base used for interpreting numbers read in. For example, the input lines

```
ibase=8
11
```

produces the output line

```
9
```

The system is now ready to do octal to decimal conversions. Beware, however, of trying to change the input base back to decimal by entering

```
ibase=10
```

Because the number 10 is interpreted as octal, this statement has no effect. For dealing in hexadecimal notation, the characters A through F are permitted in numbers (regardless of what base is in effect) and are interpreted as digits having values 10 through 15, respectively. The following statement changes the base to decimal regardless of what the current input base is:

```
ibase=A
```

Negative and large positive input bases are permitted but are useless. No mechanism has been provided for the input of arbitrary numbers in bases less than one and greater than 16.

The content of **obase**, initially 10 (decimal), is used as the base for output numbers. The input lines

```
obase=16
1000
```

produce the following output line:

```
3E8
```

This output is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted and are sometimes useful. For example, large numbers can be output in groups of five digits by setting **obase** to 100000. Strange output bases (such as 1, 0, or negative) are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with a backslash (\). Decimal output conversion is practically instantaneous, but output of very large numbers (such as those with more than 100 digits) with other bases is rather slow. Nondecimal output conversion of a 100–digit number takes about three seconds.

The **ibase** and **obase** have no effect on the course of internal computation or on the evaluation of expressions. They only affect input and output conversions, respectively.

Scaling

A third special internal quantity called **scale** is used to determine the scale of calculated quantities. The number of digits after the decimal point of a number is referred to as its scale. Numbers may have up to 99 digits after the decimal point. This fractional part is retained in further computations.

The contents of **scale** must be no greater than 99 and no less than 0. It is initially set to 0. However, appropriate scaling can be arranged when more than 99 fraction digits are required.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

- *Addition and Subtraction*— The scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result.
- *Multiplication*— The scale of the result is never less than the maximum of the two scales of the operands and never more than the sum of the scales of the operands. Subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity **scale**.
- *Division*— The scale of a quotient is the contents of the internal quantity **scale**. The scale of the remainder is the sum of the scales of the quotient and the divisor.
- *Exponentiation*— The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.
- *Square Root*— The scale of a square root is set to the maximum of the scale of the argument and the contents of **scale**.

All of the internal operations are actually carried out in terms of integers with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The internal quantities **scale, ibase,** and **obase** can be used in expressions just like other variables. The input line

scale=scale + 1

increases the value of **scale** by one. The input line

scale

causes the current value of **scale** to be printed.

The value of **scale** retains its meaning as a number of decimal digits to be retained in internal computation even when **ibase** or **obase** are not equal to 10. The internal computations (which are still conducted in decimal regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal, octal, or any other kind of digits.

Functions

The name of a function is a single lowercase letter. Function names are permitted to coincide with simple variable names. 26 different defined functions are permitted in addition to the 26 variable names. The following input line begins the definition of a function with one argument:

define a(x){

This line must be followed by one or more statements which make up the body of the function ending with a right brace (}). The general form of a function is as follows:

```
define a(x){
        ...
        ...
        return
}
```

Return of control from a function occurs when a **return** statement is executed or when the end of the function is reached. The **return** statement can take either of the two forms:

return
return(*x***)**

In the first case, the value of the function is 0; and in the second, the value of the function is the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

auto *x,y,z*

There can be only one **auto** statement in a function, and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return (exit). The

values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition follows:

```
define a(x,y){
        auto z
        z=x*y
        return(z)
}
```

The value of this function **a**, when called, is the product of its two arguments, x and y.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: **()**.

If the function **a** above has been defined, then the line

```
a(7,3.14)
```

causes the result **21.98** to be printed. The line

```
z=a(a(3,4),5)
```

causes the result **60** to be printed.

Subscripted Variables

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name, and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to coincide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be used in expressions, in function calls, and in return statements.

An array name can be used as an argument to a function or may be declared as automatic in a function by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function and thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other contexts.

Control Statements

The **if, while** and **for** statements can be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following ways:

if*(relation) statement*
while*(relation) statement*
for*(expression1;relation;expression2) statement*

or

if*(relation)* {*statements*}
while*(relation)* {*statements*}
for*(expression1;relation;expression2)* {*statements*}

A relation in one of the control statements is an expression of the form

x>y

where two expressions are related by one of the following six relational operators:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

NOTE

Beware of using one equal sign (=) instead of two equal signs (==) as a relational operator. Unfortunately, both of these are legal, so there will be no diagnostic message; but, the single equal sign (=) will not do a comparison.

The **if** statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The **while** statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range; and if the relation is false, control passes to the next statement beyond the range of the **while** statement.

The **for** statement begins by executing *expression1*. Then the relation is tested; and if true, the statements in the range of the **for** are executed. Then *expression2* is executed. The relation is then tested, and so forth. The typical use of the **for** statement is for a controlled iteration, as in the following statement:

for(*i*=1;*i*<=10;*i*=*i*+1)*i*

This example prints the integers from one to ten. The following are some examples of the use of the control statements:

```
define f(n){
auto i,x
x=1
for(i=1;i<=n;i=i+1)
x=x*i
return (x)
}
```

The input line

f(*a***)**

prints *a* factorial if *a* is a positive integer. The following is the definition of a function that computes values of the binomial coefficient (*m* and *n* are assumed to be positive integers):

```
define b(n,m){
auto x,j
x=1
for (j=1;j<=m;j=j=1)
x=x*(n-j+1)/j
return (x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale=20
define e(x){
        auto a,b,c,d,n
        a=1
        b=1
        c=1
        d=0
        n=1
        while (1==1){
                a=a*x
                b=b*n
                c=c+a/b
                n=n+1
                if(c==d)return(c)
                d=c
        }
}
```

Additional Features

There are some additional language features that every user should know.

Normally, statements are entered one to a line. It is also permissible, however, to enter several statements on a line by separating the statements by semicolons.

If an assignment statment is parenthesized, it then has a value; and it can be used anywhere that an expression can. For example, the following input line not only makes the indicated assignment, but also prints the resulting value:

(*x*=*y*+17)

The following is an example of a use of the value of an assignment statement even when it is not parenthesized. The input line causes a value to be assigned to *x* and also increments *i* before it is used as a subscript:

x=**a[***i*=*i*+**1]**

The next examples construct work in BC in exactly the same manner as they do in the C language:

`x=y=z`	is the same as	`x=(y=z)`
`x=+y`	"	`x=x+y`
`x=-y`	"	`x=x-y`
`x=*y`	"	`x=x*y`
`x=/y`	"	`x=x/y`
`x=%y`	"	`x=x%y`
`x=^y`	"	`x=x^y`
`x++`	"	`(x=x+1)-1`
`x--`	"	`(x=x-1)+1`
`++x`	"	`x=x+1`
`--x`	"	`x=x-1`

NOTE

In some of these constructions, spaces are significant. There is a real difference between **x=-y** *and* **x= -y**. *The first replaces* **x** *by* **x-y** *and the second by* **-y**.

The following are three important things to remember when using BC programs:

- To exit a BC program, enter **quit**.
- There is a comment convention identical to that of the C language. Comments begin with **/*** and end with ***/**.
- There is a library of math functions that may be obtained by entering at the command level:

  bc-l

This command loads a set of library functions that includes sine (**s**), cosine (**c**), arctangent (**a**), natural logarithm (**l**), exponential (**e**), and Bessel functions of integer order [**j(n,x)**]. The library sets the scale to 20, but it can be reset to another value.

If you enter

bc *filename...*

the BC program reads and executes the named file or files before accepting commands from the keyboard. In this way, programs and function definitions are loaded.

Summary

Notation

In the following pages, syntactic categories are in italics and literals are in bold. Material in brackets **[ ]** is optional.

Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs, or comments. Newline characters or semicolons separate statements.

Comments are introduced by the characters **/*** and terminated by ***/**.

There are three kinds of identifiers: ordinary, array, and function. All three types consist of single lowercase letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from zero to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict. A program can have a variable named *x*, an array named *x*, and a function named *x*; all are separate and distinct.

The following are reserved keywords:

ibase	if
obase	break
scale	define
sqrt	auto
length	return
while	quit
for	

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits A through F are also recognized as digits with values 10 through 15, respectively.

Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

Named Expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

Identifiers

Simple identifiers are named expressions. They have an initial value of 0.

Array-name[expression]

Array elements are named expressions. They have an initial value of 0.

Scale, Ibase, and Obase

The internal registers **scale, ibase,** and **obase** are all named expressions. The **scale** register is the number of digits after the decimal point to be retained in arithmetic operations. It has an initial value of 0. The **ibase** and **obase** registers are the input and output number radix, respectively. Both **ibase** and **obase** have initial values of 10.

Function Calls

Function Name ([expression[,expression..]])

A function call consists of a function name followed by parentheses containing a comma–separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by *value.* As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a **return** statement, the value of the function is the value of the expression in the parentheses of the **return** statement or is 0 if no expression is provided or if there is no **return** statement.

squrt(expression)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

length(expression)

The result is the total number of significant decimal digits in the expression. The scale of the result is 0.

scale(expression)

The result is the scale of the expression. The scale of the result is 0.

Constants
Constants are primitive expressions.

Parentheses
An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

The unary operators bind right to left.

-expression
The result is the negative of the expression.

+ +named-expression
The named expression is incremented by one. The result is the value of the named expression after incrementing.

- -named-expression
The named expression is decremented by one. The result is the value of the named expression before incrementing.

named-expression- -
The named expression is decremented by one. The result is the value of the named expression before decrementing.

The exponentiation operator binds right to left.

expression ^ expression
The result is the first expression raised to the power to the second expression. The second expression must be an integer. If a is the scale of the left expression and b is the absolute value of the right expression, then the scale of the result is as follows:

$$min(a \times b,\ max(scale,a))$$

The operators *, /, and % bind left to right.

*expression*expression*
The result is the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result is as follows:

$$min(a + b, max(scale, a, b))$$

expression / expression
The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

expression % expression
The **%** operator produces the remainder of the division of the two expressions. More precisely, **a%b** is equal to **a–a/b*b**.

The scale of the result is the sum of the scale of the divisor and the value of **scale**.

The additive operators bind left to right.

expression + expression
The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

expression - expression
The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

The assignment operators bind right to left.

named-expression = expression
This expression results in assigning the value of the expression on the right to the named expression on the left.

named-expression = + *expression*
named-expression = - *expression*
named-expression = * *expression*
named-expression = / *expression*
named-expression = % *expression*
named-expression = ^ *expression*

The result of the above expressions is equivalent to *named-expression* = *named expression OP expression*, where *OP* is the operator after the equal sign (=).

Relational Operators

Unlike all other operators, the relational operators are only valid as the object of an **if** or **while** statement or inside a **for** statement.

expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*
expression = = *expression*
expression ! = *expression*

Storage Classes

There are only two storage classes in BC–global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of 0. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. The **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in C language. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new value.

Statements

Statements must be separated by a semicolon or newline. Except where altered by control statements, execution is sequential.

When a statement is an expression (unless the main operator is an assignment), the value of the expression is printed followed by a newline character.

Statements may be grouped together and used when one statement is expected by surrounding them with braces { }.

The following statement prints the string inside the quotes:

"any string"

The substatement is executed if the relation is true:

if(*relation*)*statement*

The **while** statement is executed while the relation is true. The test occurs before each execution of the statement:

for(*expression;relation;expression*)*statement*

The **for** statement is the same as the following (all three expressions must be present):

first-expression
while*(relation)* {
    statement
    last-expression
}

The **break** statement causes termination of a **for** or **while** statement:

break

The **auto** statement cause the values of the identifiers to be pushed down:

auto *identifier[,identifier]*

The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name with empty square brackets. The **auto** statement must be the first statement in a function definition.

The **define** statement defines a function:

define([*parameter***[,***parameter* . . . **]])**{
    statements}

The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

The **return** statement has the following form:

return
return(*expression***)**

The **return** statment causes the following:

- Termination of a function
- Popping of the auto variables on the stack
- Specifies the results of the function

The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

The **quit** statement stops execution of a BC program and returns control to the UTek system software when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if, for** or **while** statement.