UNIX™ TIME-SHARING SYSTEM:

# UNIX PROGRAMMER'S MANUAL

*Seventh Edition, Volume 2B*

*January, 1979*

Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

# Yacc: Yet Another Compiler-Compiler

*Stephen C. Johnson*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an ''input language'' which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

July 31, 1978

# Yacc: Yet Another Compiler-Compiler

*Stephen C. Johnson*

Bell Laboratories
Murray Hill, New Jersey 07974

## 0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] 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 will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
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 which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2,3,4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

## 1: 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 Section 3, 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 ‘‘%’’ 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  ;
```

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 NUL character (´\0´ or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ‘‘|’’ can be used to avoid rewriting the left hand 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
        ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input 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 done by writing

```
%token   name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). 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 hand 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.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as ‘‘end-of-file’’ or ‘‘end-of-record’’.

**2: Actions**

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ‘‘{’’ and ‘‘}’’. For example,

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

and

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

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ‘‘dollar sign’’ ‘‘$’’ is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ‘‘$$’’ to some value. For example, an action that does nothing but return the value 1 is

```
        {  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B  C  D   ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :    ´(´  expr  ´)´  ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr    :    ´(´  expr  ´)´          {  $$ = $2 ;  }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A       :    B   ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A       :    B
                     {  $$ = 1;  }
             C
                     {   x = $2;   y = $3;  }
        ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT    :    /* empty */
                     {  $$ = 1;  }
        ;

A       :    B  $ACT  C
                     {   x = $2;   y = $3;  }
        ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr    :    expr  ´+´  expr
                     {  $$ = node( ´+´, $1, $3 );  }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ‘‘%{’’ and ‘‘%}’’. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in ‘‘yy’’; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

**3: Lexical Analysis**

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ‘‘# define’’ mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case ´0´:
        case ´1´:
          . . .
        case ´9´:
                yylval = c– ´0´;
                return( DIGIT );
                . . .
                }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

**4: How the Parser Works**

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
        IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ‘‘.’’) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
        .       reduce 18
```

refers to *grammar rule* 18, while the action

```
        IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
    A   :       x y z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
        A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action ''turns back the clock'' in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme   :       sound  place
        ;
sound   :       DING  DONG
        ;
place   :       DELL
        ;
```

When Yacc is invoked with the **−v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound   :   DING_DONG

        DONG  shift 6
        .  error

state 4
        rhyme  :   sound  place_    (1)

        .   reduce  1

state 5
        place  :   DELL_    (3)

        .   reduce  3

state 6
        sound   :   DING  DONG_    (2)

        .   reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is ''shift 3'', so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is ''shift 6'', so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound  :   DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound    goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is ''shift 5'', so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by ''$end'' in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

**5: Ambiguity and Conflicts**

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr    :    expr  ´–´  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr  –  expr  –  expr
```

the rule allows this input to be structured as either

```
(  expr  –  expr  )  –  expr
```

or as

```
expr  –  (  expr  –  expr  )
```

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr  –  expr  –  expr
```

When the parser has read the second expr, the input that it has seen:

```
expr  –  expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
–  expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

    expr  –  expr

it could defer the immediate application of the rule, and continue reading the input until it had seen

    expr  –  expr  –  expr

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

    expr  –  expr

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

    expr  –  expr

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict*. Note that there are never any ''Shift/shift'' conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an ''if-then-else'' construction:

```
stat    :       IF  ´(´  cond  ´)´  stat
        |       IF  ´(´  cond  ´)´  stat  ELSE  stat
        ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

    IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2 can be structured according to these rules in two ways:

```
IF ( C1 ) {
        IF ( C2 ) S1
        }
ELSE S2
```

or

```
IF ( C1 ) {
        IF ( C2 ) S1
        ELSE S2
        }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding ''un-*ELSE*'d'' *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (–**v**) option output file. For example, the output corresponding to the above conflict state might be:

23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

```
stat : IF ( cond ) stat_        (18)
stat : IF ( cond ) stat_ELSE stat

ELSE    shift 45
.       reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ‘‘**.**’’, is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF ´(´ cond ´)´ stat
```

Once again, notice that the numbers following ‘‘shift’’ commands refer to other states, while the numbers following ‘‘reduce’’ commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[2,3,4] might be consulted; the services of a local guru might also be appropriate.

**6: Precedence**

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left  ´+´  ´−´
%left  ´*´  ´/´
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A  .LT.  B  .LT.  C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right  ´=´
%left  ´+´  ´−´
%left  ´*´  ´/´

%%

expr    :       expr  ´=´  expr
        |       expr  ´+´  expr
        |       expr  ´−´  expr
        |       expr  ´*´  expr
        |       expr  ´/´  expr
        |       NAME
        ;
```

might be used to structure the input

```
a  =  b  =  c*d  −  e  −  f*g
```

as follows:

```
a = ( b = ( ((c*d)−e) − (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary ´−´; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left  '+'  '-'
%left  '*'  '/'

%%

expr    :       expr  '+'  expr
        |       expr  '-'  expr
        |       expr  '*'  expr
        |       expr  '/'  expr
        |       '-'  expr      %prec  '*'
        |       NAME
        ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially ''cookbook'' fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

**7: Error Handling**

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

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser ''restarted'' after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name ''error'' is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token ''error'' is legal. It then behaves as if the token ''error'' were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :    error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :    error  ´;´
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ´;´. All tokens after the error and before the next ´;´ cannot be shifted, and are discarded. When the ´;´ is seen, this rule will be reduced, and any ‘‘cleanup’’ action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :    error  ´\n´  {  printf( "Reenter last line: " );  }  input
                    {       $$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :    error  ´\n´
                    {       yyerrok;
                            printf( "Reenter last line: " );   }
             input
                    {       $$  =  $4;  }
        ;
```

As mentioned above, the token seen immediately after the ‘‘error’’ symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                        {       resynch();
                                yyerrok ;
                                yyclearin ;   }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

**8: The Yacc Environment**

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse* ; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a −**ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string ‘‘syntax error’’. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

**9: 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 owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of ‘‘knowing who to blame when things go wrong.’’

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

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

**Left Recursion**

The algorithm used by the Yacc parser encourages so called ‘‘left recursive’’ grammar rules: rules of the form

```
        name    :       name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
        list    :       item
                |       list  ','  item
                ;
```

and

```
        seq     :       item
                |       seq  item
                ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
        seq     :       item
                |       item  seq
                ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
        seq     :       /* empty */
                |       seq  item
                ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

**Lexical Tie-ins**

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
  ...  other declarations ...

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;

   ...  other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of ''backdoor'' approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

**Reserved Words**

Some programming languages permit the user to use words like ''if'', which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it ''this instance of 'if' is a keyword, and that instance is a variable''. The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved* ; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

**10: 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. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

**Support for Arbitrary Value Types**

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union  {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **–d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left  <optype>  ´+´  ´–´
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 – see the previous subsection ) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa  {  $<intval>$  =  3;  } bbb
                        {       fun( $<intval>2, $<other>0 );  }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int'* s, as was true historically.

**11: Acknowledgements**

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for ``one more feature''. Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

**References**

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, ''LR Parsing,'' *Comp. Surveys* **6**(2), pp.99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, ''Deterministic Parsing of Ambiguous Grammars,'' *Comm. Assoc. Comp. Mach.* **18**(8), pp.441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, ''Lint, a C Program Checker,'' Comp. Sci. Tech. Rep. No. 65 (December 1977). updated version TM 78-1273-3
6. S. C. Johnson, ''A Portable Compiler: Theory and Practice,'' *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp.97-104 (January 1978).
7. B. W. Kernighan and L. L. Cherry, ''A System for Typesetting Mathematics,'' *Comm. Assoc. Comp. Mach.* **18**, pp.151-157 (March 1975).
8. M. E. Lesk, ''Lex — A Lexical Analyzer Generator,'' Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

**Appendix A: A Simple Example**

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled ‘‘a’’ through ‘‘z’’, and accepts arithmetic expressions made up of the operators +, −, ∗, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
#  include  <stdio.h>
#  include  <ctype.h>

int  regs[26];
int  base;

%}

%start  list

%token  DIGIT  LETTER

%left  ´|´
%left  ´&´
%left  ´+´  ´−´
%left  ´∗´  ´/´  ´%´
%left  UMINUS      /∗  supplies  precedence  for  unary  minus  ∗/

%%      /∗  beginning  of  rules  section  ∗/

list    :       /∗  empty  ∗/
        |       list  stat  ´\n´
        |       list  error  ´\n´
                        {       yyerrok;  }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 );  }
        |       LETTER  ´=´  expr
                        {       regs[$1]  =  $3;  }
        ;

expr    :       ´(´  expr  ´)´
                        {       $$  =  $2;  }
        |       expr  ´+´  expr
                        {       $$  =  $1  +  $3;  }
        |       expr  ´−´  expr
                        {       $$  =  $1  −  $3;  }
        |       expr  ´∗´  expr
                        {       $$  =  $1  ∗  $3;  }
```

```
        |       expr  ´/´  expr
                        {       $$  =  $1  /  $3;  }
        |       expr  ´%´  expr
                        {       $$  =  $1  %  $3;  }
        |       expr  ´&´  expr
                        {       $$  =  $1  &  $3;  }
        |       expr  ´|´  expr
                        {       $$  =  $1  |  $3;  }
        |       ´−´  expr       %prec  UMINUS
                        {       $$  =  −  $2;  }
        |       LETTER
                        {       $$  =  regs[$1];  }
        |       number
        ;

number  :       DIGIT
                        {       $$ = $1;    base  =  ($1==0)  ?  8  :  10;  }
        |       number  DIGIT
                        {       $$  =  base * $1  +  $2;  }
        ;

%%      /*  start  of  programs  */

yylex() {               /*  lexical  analysis  routine  */
          /*  returns  LETTER  for  a  lower  case  letter,  yylval = 0  through  25  */
          /*  return  DIGIT  for  a  digit,  yylval = 0  through  9  */
          /*  all  other  characters  are  returned  immediately  */

        int  c;

        while(  (c=getchar())  ==  ´ ´  )  { /*  skip  blanks  */  }

        /*  c  is  now  nonblank  */

        if(  islower(  c  )  )  {
                yylval  =  c  −  ´a´;
                return  (  LETTER  );
                }
        if(  isdigit(  c  )  )  {
                yylval  =  c  −  ´0´;
                return(  DIGIT  );
                }
        return(  c  );
        }
```

**Appendix B: Yacc Input Syntax**

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /*  grammar  for  the  input  to  Yacc  */

        /*  basic  entities  */
%token  IDENTIFIER      /*  includes  identifiers  and  literals  */
%token  C_IDENTIFIER    /*  identifier  (but  not  literal)  followed  by  colon  */
%token  NUMBER          /*  [0-9]+  */

        /*  reserved  words:  %type  =>  TYPE,  %left  =>  LEFT,  etc.  */

%token  LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token  MARK    /*  the  %%  mark  */
%token  LCURL   /*  the  %{  mark  */
%token  RCURL   /*  the  %}  mark  */

        /*  ascii  character  literals  stand  for  themselves  */

%start  spec

%%

spec    :       defs  MARK  rules  tail
        ;

tail    :       MARK   {  In  this  action,  eat  up  the  rest  of  the  file  }
        |       /*  empty:  the  second  MARK  is  optional  */
        ;

defs    :       /*  empty  */
        |       defs  def
        ;

def     :       START  IDENTIFIER
        |       UNION  {  Copy union  definition  to  output  }
        |       LCURL  {  Copy  C  code  to  output  file  }  RCURL
        |       ndefs  rword  tag  nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
```

```
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       ´<´ IDENTIFIER ´>´
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ´,´ nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       ´{´ { Copy action, translate $$, etc. } ´}´
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ´;´
        ;
```

**Appendix C: An Advanced Example**

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, ''a'' through ''z''. Moreover, it also understands *intervals*, written

( x , y )

where *x* is less than or equal to *y*. There are 26 interval valued variables ''A'' through ''Z'' that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + ( 3.5 − 4. )

and

2.5 + ( 3.5 , 4. )

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the '','' is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
        double lo, hi;
        } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start   lines

%union   {
        int ival;
        double dval;
        INTERVAL vval;
        }

%token <ival> DREG VREG   /* indices into dreg, vreg arrays */

%token <dval> CONST       /* floating point constant */

%type <dval> dexp         /* expression */

%type <vval> vexp         /* interval expression */

        /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS     /* precedence for unary minus */

%%

lines   :       /* empty */
        |       lines line
        ;

line    :       dexp '\n'
                        {       printf( "%15.8f\n", $1 ); }
        |       vexp '\n'
                        {       printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
        |       DREG '=' dexp '\n'
                        {       dreg[$1] = $3; }
        |       VREG '=' vexp '\n'
```

```
                {       vreg[$1] = $3; }
        |       error '\n'
                {       yyerrok; }
        ;

dexp    :       CONST
        |       DREG
                {       $$ = dreg[$1]; }
        |       dexp '+' dexp
                {       $$ = $1 + $3; }
        |       dexp '−' dexp
                {       $$ = $1 − $3; }
        |       dexp '*' dexp
                {       $$ = $1 * $3; }
        |       dexp '/' dexp
                {       $$ = $1 / $3; }
        |       '−' dexp        %prec UMINUS
                {       $$ = − $2; }
        |       '(' dexp ')'
                {       $$ = $2; }
        ;

vexp    :       dexp
                {       $$.hi = $$.lo = $1; }
        |       '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                        printf( "interval out of order\n" );
                        YYERROR;
                        }
                }
        |       VREG
                {       $$ = vreg[$1];   }
        |       vexp '+' vexp
                {       $$.hi = $1.hi + $3.hi;
                        $$.lo = $1.lo + $3.lo;   }
        |       dexp '+' vexp
                {       $$.hi = $1 + $3.hi;
                        $$.lo = $1 + $3.lo;   }
        |       vexp '−' vexp
                {       $$.hi = $1.hi − $3.lo;
                        $$.lo = $1.lo − $3.hi;   }
        |       dexp '−' vexp
                {       $$.hi = $1 − $3.lo;
                        $$.lo = $1 − $3.hi;   }
        |       vexp '*' vexp
                {       $$ = vmul( $1.lo, $1.hi, $3 ); }
        |       dexp '*' vexp
                {       $$ = vmul( $1, $1, $3 ); }
        |       vexp '/' vexp
                {       if( dcheck( $3 ) ) YYERROR;
                        $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
	|	dexp  '/'  vexp
			{	if( dcheck( $3 ) ) YYERROR;
				$$ = vdiv( $1, $1, $3 ); }
	|	'-'  vexp	%prec UMINUS
			{	$$.hi = -$2.lo;   $$.lo = -$2.hi;   }
	|	'(' vexp ')'
			{	$$ = $2; }
	;

%%

# define BSZ 50		/* buffer size for floating point numbers */

	/* lexical analysis */

yylex(){
	register c;

	while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

	if( isupper( c ) ){
		yylval.ival = c - 'A';
		return( VREG );
		}
	if( islower( c ) ){
		yylval.ival = c - 'a';
		return( DREG );
		}

	if( isdigit( c ) || c=='.' ){
		/* gobble up digits, points, exponents */

		char buf[BSZ+1], *cp = buf;
		int dot = 0, exp = 0;

		for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

			*cp = c;
			if( isdigit( c ) ) continue;
			if( c == '.' ){
				if( dot++ || exp ) return( '.' );   /* will cause syntax error */
				continue;
				}

			if( c == 'e' ){
				if( exp++ ) return( 'e' );   /* will cause syntax error */
				continue;
				}

			/* end of number */
			break;
			}
		*cp = '\0';
		if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

```
                else ungetc( c, stdin );   /* push back last char read */
                yylval.dval = atof( buf );
                return( CONST );
                }
        return( c );
        }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a;   v.lo = b; }
        else { v.hi = b;   v.lo = a; }

        if( c>d ) {
                if( c>v.hi ) v.hi = c;
                if( d<v.lo ) v.lo = d;
                }
        else {
                if( d>v.hi ) v.hi = d;
                if( c<v.lo ) v.lo = c;
                }
        return( v );
        }

INTERVAL vmul( a, b, v ) double a, b;   INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return( 1 );
                }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b;   INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

**Appendix D: Old Features Supported but not Encouraged**

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes ‘‘"’’.

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash ‘‘\’’ may be used. In particular, \\ is the same as %%, \left the same as %left, etc.

4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ```
   ={ . . . }
   ```

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

# Lex − A Lexical Analyzer Generator

*M. E. Lesk and E. Schmidt*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

July 21, 1975

# Lex − A Lexical Analyzer Generator

*M. E. Lesk and E. Schmidt*

Bell Laboratories
Murray Hill, New Jersey 07974

**Table of Contents**



## 1. Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ''host languages.'' Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex.* The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

Source → $\boxed{\text{Lex}}$ → yylex

Input → $\boxed{\text{yylex}}$ → Output

An overview of Lex

Figure 1

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$    ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates ''one or more ...''; and the $ indicates ''end of line,'' as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.

```
          lexical        grammar
           rules          rules
             ↓              ↓
          [ Lex ]        [ Yacc ]
             ↓              ↓
Input →  [ yylex ]  →  [ yyparse ]  → Parsed input
```

Lex with Yacc

Figure 2

Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before *cd. . .* Such backup is more costly than the processing of simpler languages.

## 2. Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second *%%* is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions,* program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message ''found keyword INT'' whenever it appears. In this example the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour       printf("color");
mechanise    printf("mechanize");
petrol       printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*; a way of dealing with this will be described later.

## 3. Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D.*

*Operators.* The operator characters are

```
" \ [ ] ^ − ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

*Character classes.* Classes of characters can be specified using the operator pair [ ]. The construction *[abc]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ – and ^. The – character indicates ranges. For example,

[a–z0–9<>_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using – between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0–z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character – in a character class, it should be first or last; thus

[–+0–9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, b, or c, including all special or control characters; or

[^a–zA–Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

*Arbitrary character.* To match almost any character, the operator character

.

is the class of all characters except newline. Escaping into octal is possible although non-portable:

[\40–\176]

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

*Optional expressions.* The operator *?* indicates an optional element of an expression. Thus

ab?c

matches either *ac* or *abc*.

*Repeated expressions.* Repetitions of classes are indicated by the operators ∗ and **+**.

*a*∗

is any number of consecutive *a* characters, including zero; while

a+

is one or more instances of *a.* For example,

[a–z]+

is all strings of lower case letters. And

[A–Za–z][A–Za–z0–9]∗

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

*Alternation and Grouping.* The operator | indicates alternation:

(ab | cd)

matches either *ab* or *cd.* Note that parentheses are used for grouping, although they are not necessary on the outside level;

ab | cd

would have sufficed. Parentheses can be used for more complex expressions:

(ab | cd+)?(ef)∗

matches such strings as *abefef*, *efefef*, *cdef*, or *cddd* ; but not *abc*, *abcd*, or *abcdef*.

*Context sensitivity.* Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and *$*. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is *$*, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the **/** operator character, which indicates trailing context. The expression

ab/cd

matches the string *ab*, but only if followed by *cd.* Thus

ab$

is the same as

ab/\n

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x,* the rule should be prefixed by

<x>

using the angle bracket operator characters. If we considered ‘‘being at the beginning of a line’’ to be start condition *ONE*, then the ^ operator would be equivalent to

<ONE>

Start conditions are explained more fully later.

*Repetitions and Definitions.* The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

{digit}

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of *a*.

Finally, initial % is special, being the separator for Lex source segments.

**4. Lex Actions.**

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, *;* as an action causes this result. A frequent rule is

```
[ \t\n]     ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a−z]+*. Lex leaves this text in an external character array named *yytext.* Thus, to print the name found, a rule like

```
[a−z]+     printf("%s", yytext);
```

will print the string in *yytext.* The C function *printf* accepts a format argument and data to be printed; in this case, the format is ‘‘print string’’ (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext.* So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a−z]+     ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form *[a−z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a−zA−Z]+     {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng−1]
```

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext.* Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

*Example:* Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*    {
           if (yytext[yyleng−1] == '\\')
                yymore();
           else
                ... normal user processing
           }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled ''normal processing''.

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of ''=−a''. Suppose it is desired to treat this as ''=− a'' but print a message. A rule might be

```
=−[a−zA−Z]   {
             printf("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 ''=−''. Alternatively it might be desired to treat this as ''= −a''. To do this, just return the minus sign as well as the letter to the input:

```
=−[a−zA−Z]   {
             printf("Operator (=−) ambiguous\n");
             yyless(yyleng−2);
             ... action for = ...
             }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=−/[A−Za−z]
```

in the first case and

```
=/−[A−Za−z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of ''=−3'', however, makes

```
=−/[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;

2) *output(c)* which writes the character *c* on the output; and

3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * *?* or *$* or containing */* implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap.* In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

**5. Ambiguous Source Rules.**

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.

2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a−z]+     identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because *[a−z]+* matches 8 characters while *integer* matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .∗ dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .∗ stop on the current line. Don't try to defeat this with expressions like *[.\n]+* or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some Lex rules to do this might be

```
she    s++;
he     h++;
\n     |
.      ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that . does not include newline. Since *she* includes *he*, Lex will normally *not* recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means ‘‘go do the next alternative.’’ It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*:

```
she    {s++; REJECT;}
he     {h++; REJECT;}
\n     |
.      ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he* but not vice versa, and omit the REJECT action on *he*; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+    { ... ; REJECT;}
a[cd]+    { ... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%%
[a−z][a−z]    {digram[yytext[0]][yytext[1]]++; REJECT;}
\n            ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

## 6. Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
```

```
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only *%{* and *%}* is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D                   [0−9]
E                   [DEde][−+]?{D}+
%%
{D}+                printf("integer");
{D}+"."{D}*({E})?   |
{D}*"."{D}+({E})?   |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0−9]+/"."EQ     printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under ‘‘Summary of Source Format,’’ section 12.

**7. Usage.**

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library [6].

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same.

*UNIX.* The library is accessed by the loader flag *−ll*. So an appropriate set of commands is

```
lex source cc lex.yy.c −ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided.

**8. Lex and Yacc.**

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex(),* the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex().* In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named ''good'' and the lexical rules to be named ''better'' the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c −ly −ll
```

The Yacc library (−ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

**9. Examples.**

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
            int k;
[0−9]+      {
            k = atoi(yytext);
            if (k%7 == 0)
                 printf("%d", k+3);
            else
                 printf("%d",k);
            }
```

to do just that. The rule [0−9]+ recognizes strings of digits; *atoi* converts the digits to binary and stores the result in *k.* The operator % (remainder) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as *49.63* or *X7*. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
%%
                          int k;
−?[0−9]+                  {
                          k = atoi(yytext);
                          printf("%d", k%7 == 0 ? k+3 : k);
                          }
−?[0−9.]+                 ECHO;
[A-Za-z][A-Za-z0-9]+      ECHO;
```

Numerical strings containing a ''.'' or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if−else* has been replaced by a C conditional expression to save space; the form *a?b:c* means ''if *a* then *b* else *c*''.

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```
                     int lengs[100];
%%
[a−z]+               lengs[yyleng]++;
.                    |
\n                   ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
     if (lengs[i] > 0)
          printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return(1);* indicates that Lex is to perform wrapup. If *yywrap* returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap* that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a     [aA]
b     [bB]
```

```
        c       [cC]
        ...
        z       [zZ]
```

An additional class recognizes white space:

```
        W       [ \t]*
```

The first rule changes ``double precision'' to ``real'', or ``DOUBLE PRECISION'' to ``REAL''.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0]=='d'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
        ^"     "[^ 0]     ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as ``beginning of line, then five blanks, then anything but blank or zero.'' Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+         |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+   |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+   {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' || *p == 'D')
            *p=+ 'e'- 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the *for* loop to find the letter *d* or *D*. The program than adds *'e'–'d'*, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial *d*. By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
  {d}{s}{i}{n}          |
  {d}{c}{o}{s}          |
  {d}{s}{q}{r}{t}       |
  {d}{a}{t}{a}{n}       |
  ...
  {d}{f}{l}{o}{a}{t}    printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
    {d}{l}{o}{g}        |
    {d}{l}{o}{g}10      |
    {d}{m}{i}{n}1       |
    {d}{m}{a}{x}1       {
                        yytext[0] =+ 'a' - 'd';
                        ECHO;
                        }
```

And one routine must have initial *d* changed to initial *r*:

```
  {d}1{m}{a}{c}{h}      {yytext[0] =+ 'r'  - 'd';
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
        [A-Za-z][A-Za-z0-9]*    |
        [0-9]+                  |
        \n                      |
        .                       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

**10. Left Context Sensitivity.**

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as *$* recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
            int flag;
%%
^a          {flag = 'a'; ECHO;}
^b          {flag = 'b'; ECHO;}
^c          {flag = 'c'; ECHO;}
\n          {flag =  0 ; ECHO;}
magic       {
            switch (flag)
            {
            case 'a': printf("first"); break;
            case 'b': printf("second"); break;
            case 'c': printf("third"); break;
            default: ECHO; break;
            }
            }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

```
BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a                    {ECHO; BEGIN AA;}
^b                    {ECHO; BEGIN BB;}
^c                    {ECHO; BEGIN CC;}
\n                    {ECHO; BEGIN 0;}
<AA>magic             printf("first");
<BB>magic             printf("second");
<CC>magic             printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

## 11. Character Set.

The programs generated by Lex handle character I/O only through the routines *input, output,* and *unput.* Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext.* For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only ``%T''. The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example

```
%T
 1    Aa
 2    Bb
...
26    Zz
27    \n
28    +
29    −
30    0
31    1
...
39    9
%T
```

Sample character table.

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and − into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

## 12. Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1) Definitions, in the form ‘‘name space translation’’.

2) Included code, in the form ‘‘space code’’.

3) Included code, in the form

```
%{
code
%}
```

4) Start conditions, given in the form

```
%S name1 name2 ...
```

5) Character set tables, in the form

```
%T
number space character-string
...
%T
```

6) Changes to internal array sizes, in the form

*%x nnn*

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

| Letter | Parameter |
|---|---|
| p | positions |
| n | states |
| e | tree nodes |
| a | transitions |
| k | packed character classes |
| o | output array size |

Lines in the rules section have the form ‘‘expression action’’ where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

<table>
<tr><td>x</td><td>the character "x"</td></tr>
<tr><td>"x"</td><td>an "x", even if x is an operator.</td></tr>
<tr><td>\x</td><td>an "x", even if x is an operator.</td></tr>
<tr><td>[xy]</td><td>the character x or y.</td></tr>
<tr><td>[x−z]</td><td>the characters x, y or z.</td></tr>
<tr><td>[^x]</td><td>any character but x.</td></tr>
<tr><td>.</td><td>any character but newline.</td></tr>
<tr><td>^x</td><td>an x at the beginning of a line.</td></tr>
<tr><td><y>x</td><td>an x when Lex is in start condition y.</td></tr>
<tr><td>x$</td><td>an x at the end of a line.</td></tr>
<tr><td>x?</td><td>an optional x.</td></tr>
<tr><td>x*</td><td>0,1,2, ... instances of x.</td></tr>
<tr><td>x+</td><td>1,2,3, ... instances of x.</td></tr>
<tr><td>x|y</td><td>an x or a y.</td></tr>
<tr><td>(x)</td><td>an x.</td></tr>
<tr><td>x/y</td><td>an x but only if followed by y.</td></tr>
<tr><td>{xx}</td><td>the translation of xx from the definitions section.</td></tr>
<tr><td>x{m,n}</td><td>m through n occurrences of x</td></tr>
</table>

## 13. Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables 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, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user’s ability to manipulate the not-yet-processed input.

## 14. Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho’s string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

## 15. References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, N. J. (1978).

2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran,* Software – Practice and Experience, **5**, pp. 395-496 (1975).

3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler,* Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.

4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search,* Comm. ACM **18,** 333-340 (1975).

5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor,* Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.

6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library,* Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.

# A Portable Fortran 77 Compiler

*S. I. Feldman*

*P. J. Weinberger*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard on April 3, 1978. We report here on a compiler and run-time system for the new extended language. This is believed to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable, to be correct and complete, and to generate code compatible with calling sequences produced by C compilers. In particular, this Fortran is quite usable on UNIX† systems. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. An appendix describes the Fortran 77 language.

1 August 1978

†UNIX is a Trademark of Bell Laboratories.

# A Portable Fortran 77 Compiler

*S. I. Feldman*

*P. J. Weinberger*

Bell Laboratories
Murray Hill, New Jersey 07974

## 1. INTRODUCTION

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard [1] on April 3, 1978. for the language, known as Fortran 77, is about to be published. Fortran 77 supplants 1966 Standard Fortran [2]. We report here on a compiler and run-time system for the new extended language. The compiler and computation library were written by SIF, the I/O system by PJW. We believe ours to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable to a number of different machines, to be correct and complete, and to generate code compatible with calling sequences produced by compilers for the C language [3]. In particular, it is in use on UNIX† systems. Two families of C compilers are in use at Bell Laboratories, those based on D. M. Ritchie's PDP-11 compiler[4] and those based on S. C. Johnson's portable C compiler [5]. This Fortran compiler can drive the second passes of either family. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. We will describe implementation details in companion papers.

### 1.1. Usage

At present, versions of the compiler run on and compile for the PDP-11, the VAX-11/780, and the Interdata 8/32 UNIX systems. The command to run the compiler is

f77 *flags file* . . .

**f77** is a general-purpose command for compiling and loading Fortran and Fortran-related files. EFL [6] and Ratfor [7] source files will be preprocessed before being presented to the Fortran compiler. C and assembler source files will be compiled by the appropriate programs. Object files will be loaded. (The **f77** and **cc** commands cause slightly different loading sequences to be generated, since Fortran programs need a few extra libraries and a different startup routine than do C programs.) The following file name suffixes are understood:

| | |
|---|---|
| .f | Fortran source file |
| .e | EFL source file |
| .r | Ratfor source file |
| .c | C source file |
| .s | Assembler source file |
| .o | Object file |

The following flags are understood:

–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 on file **x.s**.

†UNIX is a Trademark of Bell Laboratories.

| | |
|---|---|
| −c | Compile but do not load. Output for **x.f, x.e, x.r, x.c,** or **x.s** is put on file **x.o.** |
| −m | Apply the M4 macro preprocessor 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** on **x.f.** Do not compile the resulting Fortran program. |
| −p | Generate code to produce usage profiles. |
| −o *f* | Put executable module on file *f.* (Default is **a.out**). |
| −w | Suppress all warning messages. |
| −w66 | Suppress warnings about Fortran 66 features used. |
| −O | Invoke the C object code optimizer. |
| −C | Compile code the checks that subscripts are within array bounds. |
| −onetrip | Compile code that performs every **do** loop at least once. (see Section 2.10). |
| −U | Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case. |
| −u | Make the default type of a variable **undefined.** (see Section 2.3). |
| −I2 | On machines which support short integers, make the default integer constants and variables short. (−**I4** is the standard value of this option). (see Section 2.14). All logical quantities will be short. |
| −E | The remaining characters in the argument are used as an EFL flag argument. |
| −R | The remaining characters in the argument are used as a Ratfor flag argument. |
| −F | Ratfor and and EFL source programs are pre-processed into Fortran files, but those files are not compiled or removed. |

Other flags, all library names (arguments beginning −**l**), and any names not ending with one of the understood suffixes are passed to the loader.

### 1.2. Documentation Conventions

In running text, we write Fortran keywords and other literal strings in boldface lower case. Examples will be presented in lightface lower case. Names representing a class of values will be printed in italics.

### 1.3. Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. Since there are C compilers running on a variety of machines, relatively small changes will make this Fortran compiler generate code for any of them. Furthermore, this approach guarantees that the resulting programs are compatible with C usage. The runtime computational library is complete. The mathematical functions are computed to at least 63 bit precision. The runtime I/O library makes use of D. M. Ritchie's Standard C I/O package [8] for transferring data. With the few exceptions described below, only documented calls are used, so it should be relatively easy to modify to run on other operating systems.

## 2. LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in the Appendix. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Standard, our compiler implements a few extensions described in this section. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

### 2.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real variables. A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

### 2.2. Internal Files

The Fortran 77 standard introduces ‘‘internal files’’ (memory arrays), but restricts their use to formatted sequential I/O statements. Our I/O system also permits internal files to be used in direct and unformatted reads and writes.

### 2.3. Implicit Undefined statement

Fortran 66 has a fixed rule that the type of a variable that does not appear 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 for overriding this rule. As an aid to good programming practice, we permit an additional type, **undefined.** The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **–u** compiler flag is equivalent to beginning each procedure with this statement.

### 2.4. Recursion

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

### 2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic.** These keywords may appear as ‘‘types’’ in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

### 2.6. Source Input Format

The Standard expects input to the compiler to be in 72 column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next sixty-six are the body of the line. (If there are fewer than seventy-two characters on a line, the compiler pads it with blanks; characters after the seventy-second are ignored).

In order to make it easier to type Fortran programs, our compiler also accepts input in variable length lines. An ampersand (‘‘&’’) in the first position 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.

In the Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary UNIX system usage, our compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the **–U** compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. Regardless of the setting of the flag, keywords will only be recognized in lower case.

### 2.7. Include Statement

The statement

    include ′stuff′

is replaced by the contents of the file **stuff.** **include**s may be nested to a reasonable depth, currently ten.

### 2.8. Binary Initialization Constants

A **logical, real,** or **integer** variable may 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 statements

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

initialize all three elements of **a** to ten.

### 2.9. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

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

Fortran 77 only has one quoting character, the apostrophe. Our compiler and I/O system recognize both the apostrophe ( ′ ) and the double-quote ( " ). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an **integer** word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

### 2.10. Hollerith

Fortran 77 does not have the old Hollerith (*n* **h**) notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In our compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

### 2.11. Equivalence Statements

As a very special and peculiar case, Fortran 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. Fortran 77 does not permit this usage, since subscript lower bounds may now be different from 1. Our compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

### 2.12. 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, as in

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

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the **−onetrip** compiler flag causes non-standard loops to be generated.

### 2.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

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

will read the record

```
–345,.05e–3,12
```

correctly.

### 2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer*2.** (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int**.) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **–I2** flag, all small integer constants will be of type **integer*2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **–I2** command flag is in effect). When the **–I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

### 2.15. Additional Intrinsic Functions

This 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 UNIX command arguments ( **getarg** and **iargc** ).

## 3. VIOLATIONS OF THE STANDARD

We know only thre ways in which our Fortran system violates the new standard:

### 3.1. Double Precision Alignment

The Fortran standards (both 1966 and 1977) permit **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

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

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

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370), run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries on machines with corresponding hardware requirements, and to issue a diagnostic if the source code demands a violation of the rule.

### 3.2. 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 arises as a subtle corollary of the way we represent character string arguments and of the one-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

### 3.3. T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. (Section 6.3.2 in the Appendix.) The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. (People who can make a case for using **tl** should let us know.) A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

## 4. INTER-PROCEDURE INTERFACE

To be able to write C procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

### 4.1. Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

### 4.2. 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]; |

(By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory).

### 4.3. Return Values

A function of type **integer, logical, real,** or **double precision** 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 to be stored. Thus,

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

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. Thus,

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

is equivalent to

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

and could be invoked in C by

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

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

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

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

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

### 4.4. Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is 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 arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

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

is equivalent to that in

```
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 zero, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order, C arrays are stored in row-major order.

## 5. FILE FORMATS

### 5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which 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 must be given, and this is used by the Fortran I/O system to make the file look as if 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 considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record’s length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each newline as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a newline at the end of each record. It is also possible for programs to write newlines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

### 5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following two nonstandard features: The I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the **fseek** routine, so there is a routine **canseek** which determines if **fseek** will have the desired effect. Also, the **inquire** statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form which would enable the program to reopen it. (The UNIX operating system implementation attempts to determine the full pathname.) Therefore there are two routines which depend on facilities of the operating system to provide these two services. In any case, the I/O system runs on the PDP-11, VAX-11/780, and Interdata 8/32 UNIX systems.

### 5.3. Pre-Connected 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 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 will they be created unless their units are used without first executing an **open.** The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end, so a **write** will append to the file and a **read** will result in an end-of-file indication. 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.

**REFERENCES**

1. *Sigplan Notices* **11**, No.3 (1976), as amended in X3J3 internal documents through ''/90.1''.

2. *USA Standard FORTRAN, USAS X3.9-1966*, New York: United States of America Standards Institute, March 7, 1966. Clarified in *Comm. ACM* **12,** 289 (1969) and *Comm. ACM* **14,** 628 (1971).

3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Englewood Cliffs: Prentice-Hall (1978).

4. D. M. Ritchie, private communication.

5. S. C. Johnson, ''A Portable Compiler: Theory and Practice'', Proc. 5th ACM Symp. on Principles of Programming Languages (January 1978).

6. S. I. Feldman, ''An Informal Description of EFL'', internal memorandum.

7. B. W. Kernighan, ''RATFOR — A Preprocessor for a Rational Fortran'', *Bell Laboratories Computing Science Technical Report #55,* (January 1977).

8. D. M. Ritchie, private communication.

## APPENDIX. Differences Between Fortran 66 and Fortran 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. At present the only current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute. The following information is from the ''/92'' document. This draft Standard is written in English rather than a meta-language, but it is forbidding and legalistic. No tutorials or textbooks are available yet.

### 1. Features Deleted from Fortran 66

#### 1.1. Hollerith

All notions of ''Hollerith'' (*n* **h**) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

#### 1.2. Extended Range

In Fortran 66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a **do** loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

### 2. Program Form

#### 2.1. Blank Lines

Completely blank lines are now legal comment lines.

#### 2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

#### 2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick of calling one entry point with a large number of arguments to cause the procedure to ''remember'' the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

### 2.4. DO Loops

**do** variables and range parameters may now be of integer, real, or double precision types. (The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use). The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = l, u, d
```

performs $\max(0, \lfloor (u-l)/d \rfloor)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

### 2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the ‘‘alternate returns’’ is described in section 5.2 of the Appendix.

## 3. Declarations

### 3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function **len** that returns the actual length of a character string). Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

### 3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i, j, k, l, m,** or **n** is of type **integer,** other variables are of type **real,** unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a ,b, c,** or **g** are **real,** those beginning with **w, x, y,** or **z** are assumed **complex,** and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

### 3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

### 3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966). The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common.**

```
real a(−5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *),  b(*), c(0:1, −2:*)
```

### 3.5. SAVE Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a **data** statement and never changed). These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **save**d in every procedure in which it is declared if the desired effect is to occur.

### 3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, ‘‘intrinsic functions’’, rather than being divided into ‘‘intrinsic’’ and ‘‘basic external’’ functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic.** Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

## 4. Expressions

### 4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
′abc′
′ain″t′
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two different quotation marks, ‘‘ ′ ’’’ and ‘‘ " ’’. (See Section 2.9 in the main text.)

### 4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash (‘‘//’’). The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

′ab′ // ′cd′
′abcd′

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a ‘‘*(*)’’ modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments).

### 4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

### 4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

a(i, j) (m:n)

is the string of $(n-m+1)$ characters beginning at the $m^{th}$ character of the character array element $a_{ij}$. Results are undefined unless $m \le n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

### 4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used). Also, multiple exponentiation is now defined:

a**b**c = a ** (b**c)

### 4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in **data** statements. (A constant expression is made up of explicit constants and **parameter**s and the Fortran operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in B common..

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

## 5. Executable Statements

### 5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to Fortran. It is called a ‘‘Block If’’. A Block If begins with a statement of the form

if ( . . . ) then

and ends with an

end if

statement. Two other new statements may appear in a Block If. There may be several

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

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **elseif, else,** or **endif** are executed. Otherwise, the next **elseif** statement in the group is executed. If none of the **elseif** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** must follow all **elseif**s in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures). A case construct may be rendered

```
if (s .eq. ′ab′) then
 . . .
else if (s .eq. ′cd′) then
 . . .
else
 . . .
end if
```

### 5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in

```
call joe(j, ∗10, m, ∗2)
```

A **return** statement may have an integer expression, such as

```
return k
```

If the entry point has $n$ alternate return (asterisk) arguments and if $1 \le k \le n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the **call** is executed.

## 6. Input/Output

### 6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, ′(i5)′) x
```

### 6.2. END=, ERR=, and IOSTAT= Clauses

A **read** or **write** statement may contain **end=, err=,** and **iostat=** clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and **a** and **x** are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the **iostat=** clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

### 6.3. Formatted I/O

#### 6.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6,′(i2,″ isn″″t ″,i1)′) 7, 4
```

produces

```
 7 isn′t 4
```

Here the format is the character constant

```
(i2,′ isn″t ′,i1)
```

and the character constant

```
 isn′t
```

is copied into the output.

#### 6.3.2. Positional Editing Codes

**t, tl, tr,** and **x** codes control where the next character is in the record. **tr***n* or *n***x** specifies that the next character is *n* to the right of the current position. **tl***n* specifies that the next character is *n* to the left of the current position, allowing parts of the record to be reconsidered. **t***n* says that the next character is to be character number *n* in the record. (See section 3.4 in the main text.)

#### 6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x=′("hello", :, " there", i4)′
write(6, x) 12
write(6, x)
```

the first **write** statement prints **hello there 12**, while the second only prints **hello**.

#### 6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The **sp** format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The **ss** format code guarantees that the I/O system will not insert the optional plus signs, and the **s** format code restores the default behavior of the I/O system. (Since we never put out optional plus signs, **ss** and **s** codes have the same effect in our implementation.)

#### 6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks will be ignored following a **bn** code in a format statement, and will be treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. (Blanks are ignored by default.)

#### 6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

### 6.3.7. Iw.m

There is a new integer output code, **i***w.m.* It is the same as **i***w*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case **i***w.*0 is special, in that if the value being printed is 0, the output field is entirely blank. **i***w***.1** is the same as **i***w*.

### 6.3.8. Floating Point

On input, exponents may start with the letter **E, D, e,** or **d.** All have the same meaning. On output we always use **e**. The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. (We do not print it.) There is a **g***w.d* format code which is the same as **e***w.d* and **f***w.d* on input, but which chooses **f** or **e** formats for output depending. on the size of the number and of *d*.

### 6.3.9. ''A'' Format Code

A codes are used for character values. **a***w* use a field width of *w*, while a plain **a** uses the length of the character item.

## 6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

## 6.5. List-Directed Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.)  2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

## 6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=,** which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a.**

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

### 6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file, in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write).** There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,"(a)") x
read(x,"(i3,i4)") n1,n2
```

which reads a card image into **x** and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

(We also support a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed.)

### 6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

#### 6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

**open** takes a variety of arguments with meanings described below.

**unit=** a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 9. If this parameter is the first one in the **open** statement, the **unit=** can be omitted.

**iostat=** is the same as in **read** or **write.**

**err=** is the same as in **read** or **write.**

**file=** a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status=scratch.**

**status=** one of **old, new, scratch,** or **unknown.** If this parameter is not given, **unknown** is assumed. If **scratch** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **new** is given, the file will be created if it doesn't exist, or truncated if it does. The meaning of **unknown** is processor dependent; our system treats it as synonymous with **old.**

**access= sequential** or **direct,** depending on whether the file is to be opened for sequential or direct I/O.

**form= formatted** or **unformatted.**

**recl=** a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

**blank= null** or **zero.** This parameter has meaning only for formatted I/O. The default value is **null. zero** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

### 6.8.2. CLOSE

**close** severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat=** and **err=** with their usual meanings, and **status=** either **keep** or **delete.** Scratch files cannot be kept, otherwise **keep** is the default. **delete** means the file will be removed. A simple example is

```
close(3, err=17)
```

### 6.8.3. INQUIRE

The **inquire** statement gives information about a unit (‘‘inquire by unit’’) or a file (‘‘inquire by file’’). Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', number=n, exist=l)
```

**file=** a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

**unit=** an integer variable specifies the unit the **inquire** is about. Exactly one of **file=** or **unit=** must be used.

**iostat=, err=** are as before.

**exist=** a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

**opened=** a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

**number=** an integer variable to which is assigned the number of the unit connected to the file, if any.

**named=** a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.

**name=** a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.

**access=** a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O. The value becomes undefined if there is no connection.

**sequential=** a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can’t tell.

**direct=** a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **'no'** if the file could not be connected for direct I/O, and **'unknown'** if we can’t tell.

**form=** a character variable to which is assigned the value **'formatted'** if the file is connected for formatted I/O, or **'unformatted'** if the file is connected for unformatted I/O.

**formatted=** a character variable to which is assigned the value **'yes'** if the file could be connected for formatted I/O, **'no'** if the file could not be connected for formatted I/O, and **'unknown'** if we can't tell.

**unformatted=** a character variable to which is assigned the value **'yes'** if the file could be connected for unformatted I/O, **'no'** if the file could not be connected for unformatted I/O, and **'unknown'** if we can't tell.

**recl=** an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.

**nextrec=** an integer variable to which is assigned one more than the number of the the last record read from a file connected for direct access.

**blank=** a character variable to which is assigned the value **'null'** if null blank control is in effect for the file connected for formatted I/O, **'zero'** if blanks are being converted to zeros and the file is connected for formatted I/O.

*The gentle reader* will remember that the people who wrote the standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file="/dev/console")
```

On a UNIX system this statement opens the console for formatted sequential I/O. An **inquire** statement for either unit 1 or file "/dev/console" would reveal that the file exists, is connected to unit 1, has a name, namely "/dev/console", is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the UNIX system environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The **err=** parameter will return system error numbers. The **inquire** statement does not give a way of determining permissions.

# RATFOR — A Preprocessor for a Rational Fortran

*Brian W. Kernighan*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The Ratfor language attempts to conceal the main deficiencies of Fortran while retaining its desirable qualities, by providing decent control flow statements:

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

and some ‘‘syntactic sugar’’:

- free form input (multiple statements/line, automatic continuation)
- unobtrusive comment convention
- translation of >, >=, etc., into .GT., .GE., etc.
- **return**(expression) statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

Ratfor is implemented as a preprocessor which translates this language into Fortran.

Once the control flow and cosmetic deficiencies of Fortran are hidden, the resulting language is remarkably pleasant to use. Ratfor programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their Fortran equivalents.

It is readily possible to write Ratfor programs which are portable to other env ironments. Ratfor is written in itself in this way, so it is also portable; versions of Ratfor are now running on at least two dozen different types of computers at over five hundred locations.

This paper discusses design criteria for a Fortran preprocessor, the Ratfor language and its implementation, and user experience.

# RATFOR — A Preprocessor for a Rational Fortran

*Brian W. Kernighan*

Bell Laboratories
Murray Hill, New Jersey 07974

## 1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs[1]. Finally, Fortran is often the most ''efficient'' language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which 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's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for ''1 to N in steps of 1 (or 2 or ...)'', but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N−1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

## 2. LANGUAGE DESCRIPTION

### Design

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the ''cosmetic'' deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any Fortran.* Any language feature which would require that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a

This paper is a revised and expanded version of one published in *Software—Practice and Experience,* October 1975.
The Ratfor described here is the one in use on UNIX and GCOS at Bell Laboratories, Murray Hill, N. J.

small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

**Statement Grouping**

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction ‘‘if a condition is true, do this group of things,’’ for example,

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

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by 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 the program doesn’t work, or when it must be modified, this must be translated back into a clearer form before one can be sure 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 the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ‘‘>’’ is clearer than ‘‘.GT.’’, so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like "**x>100**"), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of **H**’s: computers count better than people do.

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

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

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), 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. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

**The ‘‘else’’ Clause**

Ratfor provides an **else** statement to handle the construction ‘‘if a condition is true, do this thing, *otherwise* do that thing.’’

```
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 circuitous indeed:

```
          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: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **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*

where the **else** part 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's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to −1 if **x** is less than zero, to +1 if **x** is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
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 will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

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

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) 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** part handles the ''default'' case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

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

### if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

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

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

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

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        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 multi-way branches which 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,* and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression,* and there is a **default** section, the statements with it are done; 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. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch**.)

### The ``do'' Statement

The **do** statement in Ratfor is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

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

is the same as

```
        do 10 i = 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** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do.**

The *Ratfor statement* part will often be enclosed in braces, but as with 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
```

Slightly more complicated,

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

sets the entire array **m** to zero, and

```
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
```

sets the upper triangle of **m** to −1, the diagonal to zero, and the lower triangle to +1. (The operator == is ``equals'', that is, ``.EQ.''.) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

### ``break'' and ``next''

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** 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
}
```

**break** and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

**break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

### The ''while'' Statement

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

```
DO I = 2, 1
```

this will typically be done once with **I** set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

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

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: ''while some condition is true, repeat this group of statements''. It has no preconceptions about why one is looping. 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
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times,* that is, no attempt will be made to compute **x**3** and thus a potential underflow is avoided. Since the test is made at the top of a **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.)

As an aside, a sharp character ''#'' 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 — one can make marginal remarks, which is not possible with Fortran's ''C in column 1'' convention. Blank lines are also permitted anywhere (they are not in Fortran); they should be used 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**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may 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 which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

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

A semicolon by itself is a null statement, which is necessary here to mark 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. Of course the same code can be written in Fortran as

```
100     if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

### The ''for'' Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while.** A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
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** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be re-written with a **for** as

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

The syntax of the **for** statement is

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

*init* is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init, condition,* and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. 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
```

(''!='' is the same as ''.NE.''). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
      DO 10 J = 1, 80
              I = 81 − J
              IF (CARD(I) .NE. BLANK) GO TO 11
10    CONTINUE
      I = 0
11    ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we 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 test at the top of a loop instead of the bottom eliminates a potential boundary error.

### The ''repeat-until'' statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

repeat
*Ratfor statement*
until (*legal Fortran condition*)

The *Ratfor statement* part 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 bare **repeat** 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.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

### More on break and next

**break** exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

**''return'' Statement**

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero 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 (e.g., PL/I) one instead says

return (*expression*)

to return a value from a function. Since this is often clearer, Ratfor provides such a **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. (Another version of **equal** is presented shortly.)

**Cosmetics**

As we said above, 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 cosmetic facilities which may be used to make programs more readable.

**Free-form Input**

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear 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 the 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 upon 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 **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). 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:

"\\\'"

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

| | | | |
|---|---|---|---|
| == | .eq. | != | .ne. |
| > | .gt. | >= | .ge. |
| < | .lt. | <= | .le. |
| & | .and. | \| | .or. |
| ! | .not. | ¬ | .not. |

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

```
[     {          ]     }
$(    {          $)    }
```

**''define'' Statement**

Any string of alphanumeric characters can be defined 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.

**define** 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, definitions may be written as

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

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

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be 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
```

**''include'' Statement**

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. 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

suroutine y
        include commonblocks
        ...
        end
```

This ensures that all copies of the COMMON blocks are identical

**Pitfalls, Botches, Blemishes and other Failings**

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 will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

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

## 3. IMPLEMENTATION

Ratfor was originally written in C[4] on the UNIX operating system[5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

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 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 the code

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

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L       continue
```

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

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

In this latter case, the code

```
L+1     continue
```

is produced after the *statement* part of the **else.** Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else**,

```
        if (i > 0) x = a
```

should be left alone, not converted into

```
        if (.not. (i .gt. 0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of ''inefficiency'' will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v\pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

## 4. EXPERIENCE

### Good Things

''It's so much better than Fortran'' is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read.* The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

### Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, 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, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain that the generated Fortran is ''unreadable'' because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

## 5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in ''features'' — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

### Acknowledgements

C. A. R. Hoare once said that ''One thing [the language designer] should not do is to include untried ideas of his own.'' Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

### References

[1] B. G. Ryder, ''The PFORT Verifier,'' *Software—Practice & Experience,* October 1974.

[2] American National Standard Fortran. American National Standards Institute, New York, 1966.

[3] *For-word: Fortran Development Newsletter,* August 1975.

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

[5] D. M. Ritchie and K. L. Thompson, ''The UNIX Time-sharing System.'' *CACM*, July 1974.

[6] S. C. Johnson, ''YACC — Yet Another Compiler-Compiler.'' Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, ''Structured Programming with goto Statements.'' *Computing Surveys*, December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

[9] B. S. Baker, ''Struct — A Program which Structures Fortran'', Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, ''The Altran System for Rational Function Manipulation — A Survey.'' *CACM*, August 1971.

**Appendix: Usage on UNIX and GCOS.**

Beware — local customs vary. Check with a native before going into the jungle.

**UNIX**

The program **ratfor** is the basic translator; it takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **−6x**, which uses **x** as a continuation character in column 6 (UNIX uses **&** in column 1), and **−C**, which causes Ratfor comments to be copied into the generated Fortran.

The program **rc** provides an interface to the **ratfor** command which is much the same as **cc**. Thus

rc [options] files

compiles the files specified by **files**. Files with names ending in **.r** are Ratfor source; other files are assumed to be for the loader. The flags **−C** and **−6x** described above are recognized, as are

| | |
|---|---|
| −c | compile only; don′t load |
| −f | save intermediate Fortran .f files |
| −r | Ratfor only; implies −c and −f |
| −2 | use big Fortran compiler (for large programs) |
| −U | flag undeclared variables (not universally available) |

Other flags are passed on to the loader.

**GCOS**

The program **./ratfor** is the bare translator, and is identical to the UNIX version, except that the continuation convention is **&** in column 6. Thus

./ratfor files >output

translates the Ratfor source on **files** and collects the generated Fortran on file 'output' for subsequent processing.

**./rc** provides much the same services as **rc** (within the limitations of GCOS), regrettably with a somewhat different syntax. Options recognized by **./rc** include

| | |
|---|---|
| name | Ratfor source or library, depending on type |
| h=/name | make TSS H* file (runnable version); run as /name |
| r=/name | update and use random library |
| a= | compile as ascii (default is bcd) |
| C= | copy comments into Fortran |
| f=name | Fortran source file |
| g=name | gmap source file |

Other options are as specified for the **./cc** command described in [4].

**TSO, TSS, and other systems**

Ratfor exists on various other systems; check with the author for specifics.

# The M4 Macro Processor

*Brian W. Kernighan*

*Dennis M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

M4 is a macro processor available on UNIX† and GCOS. Its primary use has been as a front end for Ratfor for those cases where parameterless macros are not adequately powerful. It has also been used for languages as disparate as C and Cobol. M4 is particularly suited for functional languages like Fortran, PL/I and C since macros are specified in a functional notation.

M4 provides features seldom found even in much larger macro processors, including

- arguments
- condition testing
- arithmetic capabilities
- string and substring functions
- file manipulation

This paper is a user's manual for M4.

July 1, 1977

†UNIX is a Trademark of Bell Laboratories.

# The M4 Macro Processor

*Brian W. Kernighan*

*Dennis M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

## Introduction

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor — replacement of text by other text.

The M4 macro processor is an extension of a macro processor called M3 which was written by D. M. Ritchie for the AP-3 minicomputer; M3 was in turn based on a macro processor implemented for [1]. Readers unfamiliar with the basic ideas of macro processing may wish to read some of the discussion there.

M4 is a suitable front end 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.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric ''token'' (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

## Usage

On UNIX, use

**m4 [files]**

Each argument file is processed in order; if there are no arguments, or if an argument is \`−', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

**m4 [files] >outputfile**

On GCOS, usage is identical, but the program is called **./m4**.

## Defining Macros

The primary built-in function of M4 is **define**, which is used to define new macros. The input

**define(name, stuff)**

causes the string **name** to be defined as **stuff**. All subsequent occurrences of **name** will be replaced by **stuff**. **name** must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). **stuff** is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
 ...
if (i > N)
```

defines **N** to be 100, and uses this \`\`symbolic constant'' in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by `(', it is assumed to have no arguments. This is the situation for **N** above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
 ...
if (NNN > 100)
```

the variable **NNN** is absolutely unrelated to the defined macro **N**, even though it contains a lot of **N**'s.

Things may be defined in terms of other things. For example,

```
define(N, 100)
define(M, N)
```

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 behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string **N** is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it's just as if you had said

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now **M** is defined to be the string **N**, so when you ask for **M** later, you'll always get the value of **N** at that time (because the **M** will be replaced by **N** which will be replaced by 100).

## Quoting

The more general solution is to delay the expansion of the arguments of **define** by *quoting* them. Any text surrounded by the single quotes ` and ´ is not expanded immediately, but has the quotes stripped off. If you say

```
define(N, 100)
define(M, `N´)
```

the quotes around the **N** are stripped off as the argument is being collected, but they have served their purpose, and **M** is defined as the string **N**, not 100. The general rule is that 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 appear in the output, you have to quote it in the input, as in

```
`define´ = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining **N**:

```
define(N, 100)
 ...
define(N, 200)
```

Perhaps regrettably, the **N** in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

```
define(100, 200)
```

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine **N**, you must delay the evaluation by quoting:

```
define(N, 100)
 ...
define(`N´, 200)
```

In M4, it is often wise to quote the first argument of a macro.

If ` and ´ are not convenient for some reason, the quote characters can be changed with the built-in **changequote**:

```
changequote([, ])
```

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to **define**. **undefine** removes the definition of some macro or built-in:

```
undefine(`N´)
```

removes the definition of **N**. (Why are the quotes absolutely necessary?) Built-ins can be removed with **undefine**, as in

**undefine(`define´)**

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to 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. Don't forget the quotes!

**ifdef** actually permits three arguments; if the name is undefined, the value of **ifdef** is then the third argument, as in

**ifdef(`unix´, on UNIX, not on UNIX)**

### Arguments

So far we have discussed the simplest form of macro processing — replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of **$n** will be replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump**, defined as

**define(bump, $1 = $1 + 1)**

generates code to increment its argument by 1:

**bump(x)**

is

**x = x + 1**

A macro can have as many arguments as you want, but only the first nine are accessible, through **$1** to **$9**. (The macro name itself is **$0**, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro **cat** which simply concatenates its arguments, like this:

**define(cat, $1$2$3$4$5$6$7$8$9)**

Thus

**cat(x, y, z)**

is equivalent to

**xyz**

**$4** through **$9** are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus

**define(a,   b   c)**

defines **a** to be **b   c**.

Arguments are separated by commas, but parentheses are counted properly, so a comma ``protected'' by parentheses does not terminate an argument. That is, in

**define(a, (b,c))**

there are only two arguments; the second is literally **(b,c)**. And of course a bare comma or parenthesis can be inserted by quoting it.

### Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is **incr**, which increments its numeric argument by 1. Thus to handle the common programming situation where you want a variable to be defined as ``one more than N'', write

**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 called **eval**, which is capable of arbitrary arithmetic on integers. It provides the operators (in decreasing order of precedence)

| | |
|---|---|
| unary + and − | |
| ** or ^(exponentiation) | |
| * / % (modulus) | |
| + − | |
| == != < <= > >= | |
| ! | (not) |
| & or && | (logical and) |
| \| or \|\| | (logical or) |

Parentheses may be used to group operations where needed. 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 false is 0. The precision in **eval** is 32 bits on UNIX and 36 bits on GCOS.

As a simple example, suppose we want **M** to be **2**N+1**. Then

**define(N, 3)**
**define(M, \`eval(2**N+1)´)**

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

### File Manipulation

You can include a new file in the input at any time by the built-in function **include**:

**include(filename)**

inserts the contents of **filename** in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (\`\`silent include'') says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

**divert(n)**

all subsequent output is put onto the end of a temporary file referred to as **n**. Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

**undivert**

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

### System Command

You can run any program in the local operating system with the **syscmd** built-in. For example,

**syscmd(date)**

on UNIX runs the **date** command. Normally **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function *mktemp:* a string of XXXXX in the argument is replaced by the process id of the current process.

### Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

**ifelse(a, b, c, d)**

compares the two strings **a** and **b**. If these are identical, **ifelse** returns the string **c**; otherwise it returns **d**. Thus we might define a macro called **compare** which compares two strings and returns \`\`yes'' or \`\`no'' if they are the same or different.

**define(compare, \`ifelse($1, $2, yes, no)´)**

Note the quotes, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

**ifelse** can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

**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, so

**ifelse(a, b, c)**

is **c** if **a** matches **b**, and null otherwise.

### String Manipulation

The built-in **len** returns the length of the string that makes up its argument. Thus

**len(abcdef)**

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. **substr(s, i, n)** returns the substring of **s** that starts at the **i**th position (origin zero), and is **n** characters long. If **n** is omitted, the rest of the string is returned, so

**substr(\`now is the time´, 1)**

is

**ow is the time**

If **i** or **n** are out of range, various sensible things happen.

**index(s1, s2)** returns the index (position) in **s1** where the string **s2** occurs, or −1 if it doesn't occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

**translit(s, f, t)**

modifies **s** by replacing any character found in **f** by the corresponding character of **t**. That is,

**translit(s, aeiou, 12345)**

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters which don't have 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)**

deletes vowels from **s**.

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline; it is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, due to J. E. Weythman, is

```
divert(–1)
        define(...)
        ...
divert
```

### Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus you can say

**errprint(\`fatal error´)**

**dumpdef** is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget to quote the names!

### Summary of Built-ins

Each entry is preceded by the page number where it is described.

| | |
|---|---|
| 3 | changequote(L, R) |
| 1 | define(name, replacement) |
| 4 | divert(number) |
| 4 | divnum |
| 5 | dnl |
| 5 | dumpdef(\`name´, \`name´, ...) |
| 5 | errprint(s, s, ...) |
| 4 | eval(numeric expression) |
| 3 | ifdef(\`name´, this if true, this if false) |
| 5 | ifelse(a, b, c, d) |
| 4 | include(file) |
| 3 | incr(number) |
| 5 | index(s1, s2) |
| 5 | len(string) |
| 4 | maketemp(...XXXXX...) |
| 4 | sinclude(file) |
| 5 | substr(string, position, number) |
| 4 | syscmd(s) |
| 5 | translit(str, from, to) |
| 3 | undefine(\`name´) |
| 4 | undivert(number,number,...) |

### Acknowledgements

We are indebted to Rick Becker, John Chambers, Doug McIlroy, and especially Jim Weythman, whose pioneering use of M4 has led to several valuable improvements. We are also deeply grateful to Weythman for several substantial contributions to the code.

**References**

[1] B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, Inc., 1976.

# SED — A Non-interactive Text Editor

*Lee E. McMahon*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

*Sed* is a non-interactive context editor that runs on the UNIX† operating system. *Sed* is designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed.*

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

# SED — A Non-interactive Text Editor

*Lee E. McMahon*

Bell Laboratories
Murray Hill, New Jersey 07974

## Introduction

*Sed* is a non-interactive context editor designed to be especially useful in three cases:

1) To edit files too large for comfortable interactive editing;
2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

*Sed* is a lineal descendant of the UNIX editor, *ed.* Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed;* even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed,* and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

## 1. Overall Operation

*Sed* by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

[address1,address2][function][arguments]

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

### 1.1. Command-line Flags

Three flags are recognized on the command line:

- **-n:** tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);
- **-e:** tells *sed* to take the next argument as an editing command;
- **-f:** tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

### 1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

### 1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

### 1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

**Example:**

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

### 2. ADDRESSES: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces (‘{ }’)(Sec. 3.6.).

### 2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

### 2.2. Context Addresses

A context address is a pattern (‘regular expression’) enclosed in slashes (‘/’). The regular expressions recognized by *sed* are constructed as follows:

1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
2) A circumflex ‘^’ at the beginning of a regular expression matches the null character at the beginning of a line.
3) A dollar-sign ‘$’ at the end of a regular expression matches the null character at the end of a line.
4) The characters ‘\n’ match an imbedded newline character, but not the newline at the end of the pattern space.
5) A period ‘.’ matches any character except the terminal newline of the pattern space.
6) A regular expression followed by an asterisk ‘*’ matches any number (including 0) of adjacent occurrences of the regular expression it follows.
7) A string of characters in square brackets ‘[ ]’ matches any character in the string, and no others. If, however, the first character of the string is circumflex ‘^’, the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.
8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
9) A regular expression between the sequences ‘\(’ and ‘\)’ is identical in effect to the unadorned regular expression, but has side-effects which are described under the *s* command below and specification 10) immediately below.
10) The expression ‘*d*’ means the same string of characters matched by an expression enclosed in ‘\(’ and ‘\)’ earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of ‘\(’ counting from the left. For example, the expression ‘^\(.*\)\1’ matches a line beginning with two repeated occurrences of the same string.
11) The null regular expression standing alone (e.g., ‘//’) is equivalent to the last regular expression compiled.

To use one of the special characters (^ $ . * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash ‘\’.

For a context address to ‘match’ the input requires that the whole pattern within the address match some portion of the pattern space.

### 2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated.

Two addresses are separated by a comma.

**Examples:**

| | |
|---|---|
| /an/ | matches lines 1, 3, 4 in our sample text |
| /an.*an/ | matches line 1 |
| /^an/ | matches no lines |
| /./ | matches all lines |
| /\./ | matches line 5 |
| /r*an/ | matches lines 1,3, 4 (number = zero!) |
| /\(an\).*\1/ | matches line 1 |

## 3. FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

### 3.1. Whole-line Oriented Functions

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n -- next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a\
<text> -- append lines

The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the newline. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)i\
<text> -- insert lines

The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\

<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i, c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with *a* and *i,* <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

*Note:* Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

**Example:**

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n        n
i\       c\
XXXX     XXXX
d
```

**3.2. Substitute Function**

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). The only difference between <pattern> and a context address is that the context address must be delimited by slash (‘/’) characters; <pattern> may be delimited by any character other than space or newline.

By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

& is replaced by the string matched by <pattern>

*d* (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <pattern> enclosed in '\\(' and '\\)'. If nested substrings occur in <pattern>, the *d*th is determined by counting opening delimiters ('\\(').

As in patterns, special characters may be made literal by preceding them with backslash ('\\').

The <flags> argument may contain the following flags:

g -- substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.

p -- print the line if a successful replacement was done. The *p* flag causes the line to be written to the output if and only if a substitution was actually made by the *s* function. Notice that if several *s* functions, each followed by a *p* flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

w <filename> -- write the line to a file if a successful replacement was done. The *w* flag causes lines which are actually substituted by the *s* function to be written to a file named by <filename>. If <filename> exists before *sed* is run, it is overwritten; if not, it is created.

A single space must separate *w* and <filename>.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for *p*.

A maximum of 10 different file names may be mentioned after *w* flags and *w* functions (see below), combined.

**Examples:**

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubhla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file 'changes':

```
Through caverns measureless by man
Down by a sunless sea.
```

If the nocopy option is in effect, the command:

s/[.,;?:]/*P&*/gp

produces:

A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*

Finally, to illustrate the effect of the *g* flag, the command:

/X/s/an/AN/p

produces (assuming nocopy mode):

In XANadu did Kubhla Khan

and the command:

/X/s/an/AN/gp

produces:

In XANadu did Kubhla KhAN

**3.3. Input-output Functions**

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and <filename>.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a* functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and <filename>. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

**Examples**

Assume that the file 'note1' has the following contents:

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

Then the following command:

/Kubla/r note1

produces:

In Xanadu did Kubla Khan
Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

### 3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

Print up to and including the first newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

### 3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h -- hold pattern space

The *h* functions copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

The *H* function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2)g -- get contents of hold area

The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G -- Get contents of hold area

The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

**Example**

The commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

### 3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the adress part.

(2){ -- Grouping

The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching '}' standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *t* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The *t* function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

1) reading a new input line, or
2) executing a *t* function.

### 3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

### Reference

[1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories, 1978.

# Awk — A Pattern Scanning and Processing Language
# (Second Edition)

*Alfred V. Aho*

*Brian W. Kernighan*

*Peter J. Weinberger*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

*Awk* is a programming language whose basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns. *Awk* makes certain data selection and transformation operations easy to express; for example, the *awk* program

```
length > 72
```

prints all input lines whose length exceeds 72 characters; the program

```
NF % 2 == 0
```

prints all lines with an even number of fields; and the program

```
{ $1 = log($1); print }
```

replaces the first field of each line by its logarithm.

*Awk* patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, if-else, while, for statements, and multiple output streams.

This report contains a user's guide, a discussion of the design and implementation of *awk*, and some timing statistics.

September 1, 1978

# Awk — A Pattern Scanning and Processing Language
(Second Edition)

*Alfred V. Aho*

*Brian W. Kernighan*

*Peter J. Weinberger*

Bell Laboratories
Murray Hill, New Jersey 07974

## 1. Introduction

*Awk* is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX† program *grep*[1] will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev      { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

### 1.1. Usage

The command

```
awk  program  [files]
```

executes the *awk* commands in the string program on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file pfile, and executed by the command

```
awk  −f pfile  [files]
```

†UNIX is a Trademark of Bell Laboratories.

### 1.2. Program Structure

An *awk* program is a sequence of statements of the form:

```
pattern     { action }
pattern     { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

### 1.3. Records and Fields

*Awk* input is divided into ‘‘records’’ terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named NR.

Each input record is considered to be divided into ‘‘fields.’’ Fields are normally separated by white space — blanks or tabs — but the input field separator may be changed, as described below. Fields are referred to as $1, $2, and so forth, where $1 is the first field, and $0 is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named NF.

The variables FS and RS refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument −F*c* may also be used to set FS to the character *c* .

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable FILENAME contains the name of the current input file.

### 1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the *awk* command print. The *awk* program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables NF and NR can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, $1, on the file foo1, and the second field on file foo2. The >> notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file foo. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process (on UNIX only); for instance,

```
print | "mail bwk"
```

mails the output to bwk.

The variables OFS and ORS may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the print statement.

*Awk* also provides the printf statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in format and prints them. For example,

```
printf "%8.2f  %10ld\n", $1, $2
```

prints $1 as a floating point number 8 digits wide, with two after the decimal point, and $2 as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of printf is identical to that used with C.[2]

## 2. Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

### 2.1. 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 record has been processed. BEGIN and END thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN    { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by

```
END   { print NR }
```

If BEGIN is present, it must be the first pattern; END must be the last if used.

### 2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name ‘‘smith’’. If a line contains ‘‘smith’’ as part of a larger word, it will also be printed, as in

```
blacksmithing
```

*Awk* regular expressions include the regular expression forms found in the UNIX text editor *ed*[1] and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for ''one or more'', and ? for ''zero or one'', all as in *lex*. Character classes may be abbreviated: [a–zA–Z0–9] is the set of all letters and digits. As an example, the *awk* program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names ''Aho,'' ''Weinberger'' or ''Kernighan,'' whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn of the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches ''john'' or ''John.'' Notice that this will also match ''Johnson'', ''St. Johnsbury'', and so on. To restrict it to exactly [jJ]ohn, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign $ refers to the end.

### 2.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

### 2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with ''s'', but is not ''smith''. && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

### 2.5. Pattern Ranges

The ''pattern'' that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2 { ... }
```

In this case, the action is performed for each line between an occurrence of pat1 and the next occurrence of pat2 (inclusive). For example,

```
/start/, /stop/
```

prints all lines between start and stop, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

## 3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

### 3.1. Built-in Functions

*Awk* provides a ''length'' function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

length by itself is a ''pseudo-variable'' which yields the length of the current record; length(argument) is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

*Awk* also provides the arithmetic functions sqrt, log, exp, and int, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function substr(s, m, n) produces the substring of s that begins at position m (origin 1) and is at most n characters long. If n is omitted, the substring goes to the end of s. The function index(s1, s2) returns the position where the string s2 occurs in s1, or zero if it does not.

The function sprintf(f, e1, e2, ...) produces the value of the expressions e1, e2, etc., in the printf format specified by f. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets x to the string produced by formatting the values of $1 and $2.

## 3.2. Variables, Expressions, and Assignments

*Awk* variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to x. Strings which cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most BEGIN sections. For example, the sums of the first two fields can be computed by

```
      { s1 += $1; s2 += $2 }
END { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (mod). The C increment ++ and decrement −− operators are also available, and so are the assignment operators +=, −=, *=, /=, and %=. These operators may all be used in expressions.

## 3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
      $3 = "too big"
  print
}
```

which replaces the third field by ‘‘too big’’ when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string s into array[1], ..., array[n]. The number of elements found is returned. If the sep argument is provided, it is used as the field separator; otherwise FS is used as the separator.

## 3.4. String Concatenation

Strings may be concatenated. For example

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a print statement,

```
print $1 " is " $2
```

prints the two fields separated by ‘‘ is ’’. Variables and numeric expressions may also appear in concatenations.

## 3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the NR-th element of the array x. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the *awk* program

```
      { x[NR] = $0 }
END { ... program ... }
```

The first action merely records each input line in the array x.

Array elements may be named by non-numeric values, which gives *awk* a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like apple, orange, etc. Then the program

```
/apple/     { x["apple"]++ }
/orange/    { x["orange"]++ }
END         { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

### 3.6. Flow-of-Control Statements

*Awk* provides the basic flow-of-control statements if-else, while, for, and statement grouping with braces, as in C. We showed the if statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the if is done. The else part is optional.

The while statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
      print $i
      ++i
}
```

The for statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
      print $i
```

does the same job as the while statement above.

There is an alternate form of the for statement which is suited for accessing the elements of an associative array:

```
for (i in array)
      statement
```

does *statement* with i set in turn to each element of array. The elements are accessed in an apparently random order. Chaos will ensue if i is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an if, while or for can include relational operators like <, <=, >, >=, == (‘‘is equal to’’), and != (‘‘not equal to’’); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and of course parentheses for grouping.

The break statement causes an immediate exit from an enclosing while or for; the continue statement causes the next iteration to begin.

The statement next causes *awk* to skip immediately to the next record and begin scanning the patterns from the top. The statement exit causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character # and end with the end of the line, as in

```
print x, y # this is a comment
```

## 4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *fgrep* searches for a set of keywords with a particularly fast algorithm. *Sed*[1] provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.

*Lex*[3] provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

*Awk* is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

*Awk* also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn’t do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. We have tried to make the syntax powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called ''report generation'' — processing an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

## 5. Implementation

The actual implementation of *awk* uses the language development tools available on the UNIX operating system. The grammar is specified with *yacc*;[4] the lexical analysis is done by *lex*; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An *awk* program is translated into a parse tree which is then directly executed by a simple interpreter.

*Awk* was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the UNIX programs *wc*, *grep*, *egrep*, *fgrep*, *sed*, *lex*, and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing ''doug''.
3. print all lines containing ''doug'', ''ken'' or ''dmr''.
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing ''doug'', ''ken'', and ''dmr'' to files ''jdoug'', ''jken'', and ''jdmr'', respectively.
7. print each line prefixed by ''line-number : ''.
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls −l*; each line has the form

```
-rw-rw-rw- 1 ava 123 Oct 15 17:05 xxx
```

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*.

## References


1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories (May 1975). Sixth Edition
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
3. M. E. Lesk, ''Lex — A Lexical Analyzer Generator,'' Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (1975).
4. S. C. Johnson, ''Yacc — Yet Another Compiler-Compiler,'' Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

| Program | Task 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| *wc* | 8.6 | | | | | | | |
| *grep* | 11.7 | 13.1 | | | | | | |
| *egrep* | 6.2 | 11.5 | 11.6 | | | | | |
| *fgrep* | 7.7 | 13.8 | 16.1 | | | | | |
| *sed* | 10.2 | 11.6 | 15.8 | 29.0 | 30.5 | 16.1 | | |
| *lex* | 65.1 | 150.1 | 144.2 | 67.7 | 70.3 | 104.0 | 81.7 | 92.8 |
| *awk* | 15.0 | 25.6 | 29.9 | 33.3 | 38.9 | 46.4 | 71.4 | 31.1 |

**Table I.** Execution Times of Programs. (Times are in sec.)

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1. 
```
END {print NR}
```

2. 
```
/doug/
```

3. 
```
/ken|doug|dmr/
```

4. 
```
{print $3}
```

5. 
```
{print $3, $2}
```

6. 
```
/ken/ {print >"jken"}
/doug/      {print >"jdoug"}
/dmr/ {print >"jdmr"}
```

7. 
```
{print NR ": " $0}
```

8. 
```
        {sum = sum + $4}
END {print sum}
```

SED:

1. 
```
$=
```

2. 
```
/doug/p
```

3. 
```
/doug/p
/doug/d
/ken/p
/ken/d
/dmr/p
/dmr/d
```

4. 
```
/[^ ]* [ ]*[^ ]* [ ]*\([^ ]*\) .*/s//\1/p
```

5. 
```
/[^ ]* [ ]*\([^ ]*\) [ ]*\([^ ]*\) .*/s//\2 \1/p
```

6. 
```
/ken/w jken
/doug/w jdoug
/dmr/w jdmr
```

LEX:

1. 
```
%{
int i;
%}
%%
\n      i++;
.       ;
%%
yywrap() {
        printf("%d\n", i);
}
```

2. 
```
%%
^.*doug.*$ printf("%s\n", yytext);
.       ;
\n      ;
```

# DC – An Interactive Desk Calculator

*Robert Morris*

*Lorinda Cherry*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

DC is an interactive desk calculator program implemented on the UNIX† time-sharing system to do arbitrary-precision integer arithmetic. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated is limited only by available core storage. On typical implementations of UNIX, the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

November 15, 1978

# DC – An Interactive Desk Calculator

*Robert Morris*

*Lorinda Cherry*

Bell Laboratories
Murray Hill, New Jersey 07974

DC is an arbitrary precision arithmetic package implemented on the UNIX† time-sharing system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. 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 from the standard input.

## SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

**number**

The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A–F which are treated as digits with values 10–15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

**+ – * % ^**

The top two values on the stack are added (**+**), subtracted (**–**), multiplied (*****), divided (**/**), remaindered (**%**), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

**s***x*

The top of the main stack is popped and stored into a register named *x*, where *x* may be any character. If the **s** is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

---

†UNIX is a Trademark of Bell Laboratories.

**l***x*

The value in register *x* is pushed onto the stack. The register *x* is not altered. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

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**.

**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 and in registers 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 capitalized, 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. For the treatment of numbers with decimal points, see the detailed description below.

**!**

interprets the rest of the line as a UNIX command. Control returns to DC when the UNIX command terminates.

**c**

All values on the stack are popped; the stack becomes empty.

**i**

The top value on the stack is popped and used as the number radix for further input. If **i** is capitalized, 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 capitalized, 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 capitalized, 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.

## DETAILED DESCRIPTION

### 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 0–99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always −1 and all other digits are in the range 0–99. The digit preceding the high order −1 digit is never a 99. The representation of −157 is 43,98,−1. We shall call this 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 that is convenient, as it sometimes is.

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

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done 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 done 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 2. 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. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce 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. This is an implementation of the 'buddy system' of allocation described in [2].

Failing to find a string of the proper length after coalescing, 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 at any time in the process of trying to allocate a string, the allocator runs out of headers, 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. **scale** is the bound on the number of decimal places retained in arithmetic computations. **scale** may be set to the number on the top of the stack truncated to an integer with the **k** command. **K** may be used to push the value of **scale** on the stack. **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

### Addition and Subtraction

The scales 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 proceeding as in addition.

Finally, 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 which are not in the range 0–99 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 mimics 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 which 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 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. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and 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, with propagation of carry as needed. The sign is set from the sign 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 stripped 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 sqrt(y) is Newton's method with successive approximations by the rule

$$x_{n+1} = \tfrac{1}{2}\left(x_n + \frac{y}{x_n}\right)$$

The initial guess is found by taking the integer square root of the top two digits.

**Exponentiation**

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. 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 a _. The hexadecimal digits A–F correspond to the numbers 10–15 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 is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command **I** will push 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 can be output by typing the command **f**. The **o** command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command **O** pushes the value of the output base on the stack.

### Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a \ 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 can be used for decimal-octal or decimal-hexadecimal conversions.

### Internal Registers

Numbers or strings may 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**. *x* can be any character. **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 on 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 **[]** 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 **[]** to store strings, **x** to execute and the testing commands ‘<’, ‘>’, ‘=’, ‘!<’, ‘!>’, ‘!=’ can be used to program DC. The **x** command assumes the top of the stack is an 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 0-9,

```
[lip1+  si  li10>a]sa
0si  lax
```

### Push-Down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push-down 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 push-down stacks. **l** doesn’t effect the top of the register stack, and **s** destroys what was there before.

The commands to work on arrays are **:** and **;**. **:***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. **;***x* is the command to load 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 UNIX command and passes it to UNIX 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 was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% 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 5% 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 was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause 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 and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and 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 the user 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 many places the user wants. In this case only, the user 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.

**References**

[1] L. L. Cherry, R. Morris, *BC – An Arbitrary Precision Desk-Calculator Language.*

[2] K. C. Knowlton, *A Fast Storage Allocator,* Comm. ACM **8**, pp. 623-625 (Oct. 1965).

# BC – An Arbitrary Precision Desk-Calculator Language

*Lorinda Cherry*

*Robert Morris*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

BC is a language and a compiler for doing arbitrary precision arithmetic on the PDP-11 under the UNIX† time-sharing system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two five hundred-digit numbers can be multiplied to give a thousand digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

– to do computation with large integers,

– to do computation accurate to many decimal places,

– conversion of numbers from one base to another base.

November 12, 1978

# BC – An Arbitrary Precision Desk-Calculator Language

*Lorinda Cherry*

*Robert Morris*

Bell Laboratories
Murray Hill, New Jersey 07974

## Introduction

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX† time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is 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 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

## Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators –, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an 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 expression

```
7+–3
```

is interpreted to mean that –3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and –. 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. The two expressions

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions

†UNIX is a Trademark of Bell Laboratories.

```
a*b*c  and  (a*b)*c
```

BC shares with Fortran and C the undesirable convention that

```
a/b*c  is equivalent to  (a/b)*c
```

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed 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 (but see scaling below). The lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

**Bases**

There are special internal quantities, called ‘ibase’ and ‘obase’. The contents of ‘ibase’, initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A–F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10–15 respectively. The statement

```
ibase = A
```

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of ‘obase’, initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting ‘obase’ to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

**Scaling**

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For 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. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. 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 contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of 'scale' by one, and the 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 or octal or any other kind of digits.

**Functions**

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }. 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 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. 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 is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

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: b().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of x to become 60.

**Subscripted Variables**

A single lower-case 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 collide 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 freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition 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 which refer to whole arrays cannot be used in any other contexts.

**Control Statements**

The 'if', the 'while', and the 'for' statements may 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 way

```
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 six relational operators <, >, <=, >=, ==, or !=. The relation == stands for 'equal to' and != stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really 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.

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 tested, and so on. The typical use of the 'for' statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here 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 line

```
f(a)
```

will print *a* factorial if *a* is a positive integer. Here is the definition of a function which will compute 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
        }
}
```

**Some Details**

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2] for their exact workings.

```
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
```

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

WARNING! 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.

**Three Important Things**

1. To exit a BC program, type ‘quit’.

2. There is a comment convention identical to that of C and of PL/I. Comments begin with ‘/*’ and end with ‘*/’.

3. There is a library of math functions which may be obtained by typing at command level

bc –l

This command will load a set of library functions which, at the time of writing, consists of sine (named ‘s’), cosine (‘c’), arctangent (‘a’), natural logarithm (‘l’), exponential (‘e’) and Bessel functions of integer order (‘j(n,x)’). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [3].

If you type

bc file ...

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

**Acknowledgement**

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

**References**

[1] K. Thompson and D. M. Ritchie, *UNIX Programmer’s Manual,* Bell Laboratories, 1978.

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

[3] R. Morris, *A Library of Reference Standard Mathematical Subroutines,* Bell Laboratories internal memorandum, 1975.

[4] S. C. Johnson, *YACC — Yet Another Compiler-Compiler.* Bell Laboratories Computing Science Technical Report #32, 1978.

[5] R. Morris and L. L. Cherry, *DC – An Interactive Desk Calculator.*

Appendix

## 1. Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [ ] is optional.

## 2. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

### 2.1. Comments

Comments are introduced by the characters /* and terminated by */.

### 2.2. Identifiers

There are three kinds of identifiers – ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case 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 0 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 of which are separate and distinct.

### 2.3. Keywords

The following are reserved keywords:

| | |
|---|---|
| **ibase** | **if** |
| **obase** | **break** |
| **scale** | **define** |
| **sqrt** | **auto** |
| **length** | **return** |
| **while** | **quit** |
| **for** | |

### 2.4. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A**–**F** are also recognized as digits with values 10–15, respectively.

## 3. 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.

### 3.1. Primitive expressions

#### 3.1.1. 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.

##### 3.1.1.1. *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

##### 3.1.1.2. *array-name* [ *expression* ]

Array elements are named expressions. They have an initial value of zero.

##### 3.1.1.3. **scale**, **ibase** and **obase**

The internal registers **scale**, **ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

#### 3.1.2. Function calls

##### 3.1.2.1. *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 zero if no expression is provided or if there is no return statement.

##### 3.1.2.2. **sqrt (** *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.

##### 3.1.2.3. **length (** *expression* **)**

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

##### 3.1.2.4. **scale (** *expression* **)**

The result is the scale of the expression. The scale of the result is zero.

#### 3.1.3. Constants

Constants are primitive expressions.

#### 3.1.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

### 3.2. Unary operators

The unary operators bind right to left.

#### 3.2.1. – *expression*

The result is the negative of the expression.

#### 3.2.2. ++ *named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

#### 3.2.3. – – *named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

#### 3.2.4. *named-expression* ++

The named expression is incremented by one. The result is the value of the named expression before incrementing.

#### 3.2.5. *named-expression* – –

The named expression is decremented by one. The result is the value of the named expression before decrementing.

### 3.3. Exponentiation operator

The exponentiation operator binds right to left.

#### 3.3.1. *expression* ^ *expression*

The result is the first expression raised to the power of 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:

$\min(a \times b, \max(\mathbf{scale}, a))$

### 3.4. Multiplicative operators

The operators *, /, % bind left to right.

#### 3.4.1. *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:

$\min(a+b, \max(\mathbf{scale}, a, b))$

#### 3.4.2. *expression* / *expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

#### 3.4.3. *expression* % *expression*

The % operator produces the remainder of the division of the two expressions. More precisely, $a\%b$ is $a-a/b*b$.

The scale of the result is the sum of the scale of the divisor and the value of **scale**

### 3.5. Additive operators

The additive operators bind left to right.

#### 3.5.1. *expression* **+** *expression*

The result is the sum of the two expressions. The scale of the result is the maximun of the scales of the expressions.

#### 3.5.2. *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.

### 3.6. assignment operators

The assignment operators bind right to left.

#### 3.6.1. *named-expression* **=** *expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

#### 3.6.2. *named-expression* **=+** *expression*

#### 3.6.3. *named-expression* **=–** *expression*

#### 3.6.4. *named-expression* **=*** *expression*

#### 3.6.5. *named-expression* **=/** *expression*

#### 3.6.6. *named-expression* **=%** *expression*

#### 3.6.7. *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 = sign.

## 4. Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

### 4.1. *expression* **<** *expression*

### 4.2. *expression* **>** *expression*

### 4.3. *expression* **<=** *expression*

### 4.4. *expression* **>=** *expression*

### 4.5. *expression* **==** *expression*

### 4.6. *expression* **!=** *expression*

## 5. 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 zero. 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. **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 either C or PL/I. 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 values.

## 6. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

### 6.1. Expression statements

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.

### 6.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

### 6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

### 6.4. If statements

**if (** *relation* **)** *statement*

The substatement is executed if the relation is true.

### 6.5. While statements

**while (** *relation* **)** *statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

### 6.6. For statements

**for (** *expression***;** *relation***;** *expression* **)** *statement*

The for statement is the same as

*first-expression*<br>
**while (***relation* **)** {<br>
    *statement*<br>
    *last-expression*<br>
}

All three expressions must be present.

### 6.7. Break statements

**break**

**break** causes termination of a **for** or **while** statement.

### 6.8. Auto statements

**auto** *identifier* [ **,***identifier* ]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

### 6.9. Define statements

**define(** [*parameter* [ **,** *parameter* . . . ] ] **)** {
*statements* }

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

### 6.10. Return statements

**return**

**return(** *expression* **)**

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

### 6.11. Quit

The quit statement stops execution of a BC program and returns control to UNIX 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.

# Assembler Reference Manual

*Dennis M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

## 0. Introduction

This document describes the usage and input syntax of the UNIX PDP-11 assembler *as*. The details of the PDP-11 are not described.

The input syntax of the UNIX assembler is generally similar to that of the DEC assembler PAL-11R, although its internal workings and output format are unrelated. It may be useful to read the publication DEC-11-ASDB-D, which describes PAL-11R, although naturally one must use care in assuming that its rules apply to *as*.

*As* is a rather ordinary assembler without macro capabilities. It produces an output file that contains relocation information and a complete symbol table; thus the output is acceptable to the UNIX link-editor *ld*, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

## 1. Usage

*as* is used as follows:

as [ **−u** ] [ **−o** *output* ] $file_1$ . . .

If the optional ''−u'' argument is given, all undefined symbols in the current assembly will be made undefined-external. See the **.globl** directive below.

The other arguments name files which are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

The output of the assembler is by default placed on the file *a.out* in the current directory; the ''−o'' flag causes the output to be placed on the named file. If there were no unresolved external references, and no errors detected, the output file is marked executable; otherwise, if it is produced at all, it is made non-executable.

## 2. Lexical conventions

Assembler tokens include identifiers (alternatively, ''symbols'' or ''names''), temporary symbols, constants, and operators.

### 2.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period ''.'', underscore ''_'', and tilde ''~'' as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. When a name begins with a tilde, the tilde is discarded and that occurrence of the identifier generates a unique entry in the symbol table which can match no other occurrence of the identifier. This feature is used by the C compiler to place names of local variables in the output symbol table without having to worry about making them unique.

### 2.2 Temporary symbols

A temporary symbol consists of a digit followed by ‘‘f’’ or ‘‘b’’. Temporary symbols are discussed fully in §5.1.

### 2.3 Constants

An octal constant consists of a sequence of digits; ‘‘8’’ and ‘‘9’’ are taken to have octal value 10 and 11. The constant is truncated to 16 bits and interpreted in two’s complement notation.

A decimal constant consists of a sequence of digits terminated by a decimal point ‘‘**.**’’. The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

A single-character constant consists of a single quote ‘‘′’’ followed by an ASCII character not a new-line. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent new-line and other non-graphics (see *String statements*, §5.5). The constant’s value has the code for the given character in the least significant byte of the word and is null-padded on the left.

A double-character constant consists of a double quote ‘‘"’’ followed by a pair of ASCII characters not including new-line. Certain dual-character escape sequences are acceptable in place of either of the ASCII characters to represent new-line and other non-graphics (see *String statements*, §5.5). The constant’s value has the code for the first given character in the least significant byte and that for the second character in the most significant byte.

### 2.4 Operators

There are several single- and double-character operators; see §6.

### 2.5 Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

### 2.6 Comments

The character ‘‘/’’ introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

## 3. Segments

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed. The UNIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor *ld* (using its ‘‘−n’’ flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment; if the text segment is pure, the data segment begins at the lowest 8K byte boundary after the text segment.

The bss segment may not contain any explicitly initialized code or data. The length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. Typically the bss segment is set up by statements exemplified by

```
lab: . = .+10
```

The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also *Location counter* and *Assignment statements* below.

## 4. The location counter

One special symbol, ‘‘**.**’’, is the location counter. Its value at any time is the offset within the appropriate segment of the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of ‘‘**.**’’ may not decrease. If the effect of the assignment is to increase the value of ‘‘**.**’’, the required number of null bytes are generated (but see *Segments* above).

## 5. Statements

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are five kinds of statements: null statements, expression statements, assignment statements, string statements, and keyword statements.

Any kind of statement may be preceded by one or more labels.

### 5.1 Labels

There are two kinds of label: name labels and numeric labels. A name label consists of a name followed by a colon ( : ). The effect of a name label is to assign the current value and type of the location counter ‘‘**.**’’ to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the ‘‘**.**’’ value assigned changes the definition of the label.

A numeric label consists of a digit *0* to *9* followed by a colon ( : ). Such a label serves to define temporary symbols of the form ‘‘*n* b’’ and ‘‘*n* f’’, where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of ‘‘**.**’’ to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form ‘‘*n* f’’ refer to the first numeric label ‘‘*n* :’’ *f* orward from the reference; ‘‘*n* b’’ symbols refer to the first ‘‘*n* :’’ label *b* ackward from the reference. This sort of temporary label was introduced by Knuth [*The Art of Computer Programming, Vol I: Fundamental Algorithms* ]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

### 5.2 Null statements

A null statement is an empty statement (which may, however, have labels). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

### 5.3 Expression statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its (16-bit) value and places it in the output stream, together with the appropriate relocation bits.

### 5.4 Assignment statements

An assignment statement consists of an identifier, an equals sign ( = ), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter ‘‘**.**’’. It is required, however, that the type of the expression assigned be of the same type as ‘‘**.**’’, and it is forbidden to decrease the value of ‘‘**.**’’. In practice, the most common assignment to ‘‘**.**’’ has the form ‘‘. = . + *n*’’ for some number *n;* this has the effect of generating *n* null bytes.

### 5.5 String statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left string quote ‘‘<’’ followed by a sequence of ASCII characters not including newline, followed by a right string quote ‘‘>’’. Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

| | | |
|---|---|---|
| \n | NL | (012) |
| \s | SP | (040) |
| \t | HT | (011) |
| \e | EOT | (004) |
| \0 | NUL | (000) |
| \r | CR | (015) |
| \a | ACK | (006) |
| \p | PFX | (033) |
| \\ | \ | |
| \> | > | |

The last two are included so that the escape character and the right string quote may be represented. The same escape sequences may also be used within single- and double-character constants (see §2.3 above).

### 5.6 Keyword statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

## 6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, temporary symbols, operators, and brackets. Each expression has a type.

All operators in expressions are fundamentally binary in nature; if an operand is missing on the left, a 0 of absolute type is assumed. Arithmetic is two’s complement and has 16 bits of precision. All operators have equal precedence, and expressions are evaluated strictly left to right except for the effect of brackets.

### 6.1 Expression operators

The operators are:

(blank) when there is no operand between operands, the effect is exactly the same as if a ‘‘+’’ had appeared.

+ addition

– subtraction

* multiplication

\/ division (note that plain ‘‘/’’ starts a comment)

& bitwise **and**

| bitwise **or**

\> logical right shift

\< logical left shift

% modulo

! *a* ! *b* is *a* **or** ( **not** *b* ); i.e., the **or** of the first operand and the one's complement of the second; most common use is as a unary.

^ result has the value of first operand and the type of the second; most often used to define new machine instructions with syntax identical to existing instructions.

Expressions may be grouped by use of square brackets ‘‘[ ]’’. (Round parentheses are reserved for address modes.)

### 6.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined
: Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

undefined external
: A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor *ld* must be used to load the assembler's output with another routine that defines the undefined reference.

absolute
: An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text
: The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of ‘‘**.**’’ is text 0.

data
: The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first **.data** statement, the value of ‘‘**.**’’ is data 0.

bss
: The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first **.bss** statement, the value of ‘‘**.**’’ is bss 0.

external absolute, text, data, or bss
: symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

register
: The symbols

**r0** . . . **r5**
**fr0** . . . **fr5**
**sp**
**pc**

are predefined as register symbols. Either they or symbols defined from them must be used to refer to the six general-purpose, six floating-point, and the 2 special-purpose machine registers. The behavior of the floating register names is identical to that of the corresponding general register names; the former are provided as a mnemonic aid.

other types

Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

### 6.3 Type propagation in expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the ‘‘other types’’ mentioned above, or with a register expression, the result has the register or other type. As a consequence, one can refer to r3 as ‘‘r0+3’’. If two operands of ‘‘other type’’ are combined, the result has the numerically larger type An ‘‘other type’’ combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

– If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

ˆ This operator follows no other rule than that the result has the value of the first operand and the type of the second.

others

It is illegal to apply these operators to any but absolute symbols.

### 7. Pseudo-operations

The keywords listed below introduce statements that generate data in unusual forms or influence the later operations of the assembler. The metanotation

[ stuff ] . . .

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

### 7.1 .byte *expression* [ **,** *expression* ] . . .

The *expression*s in the comma-separated list are truncated to 8 bits and assembled in successive bytes. The expressions must be absolute. This statement and the string statement above are the only ones that assemble data one byte at at time.

**7.2 .even**

If the location counter ``**.**'' is odd, it is advanced by one so the next statement will be assembled at a word boundary.

**7.3 .if** *expression*

The *expression* must be absolute and defined in pass 1. If its value is nonzero, the **.if** is ignored; if zero, the statements between the **.if** and the matching **.endif** (below) are ignored. **.if** may be nested. The effect of **.if** cannot extend beyond the end of the input file in which it appears. (The statements are not totally ignored, in the following sense: **.if**s and **.endif**s are scanned for, and moreover all names are entered in the symbol table. Thus names occurring only inside an **.if** will show up as undefined if the symbol table is listed.)

**7.4 .endif**

This statement marks the end of a conditionally-assembled section of code. See **.if** above.

**7.5 .globl** *name* [ **,** *name* ] ...

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the **.globl** statement were not given; however, the link editor *ld* may be used to combine this routine with other routines that refer these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols. As discussed in §1, it is possible to force the assembler to make all otherwise undefined symbols external.

**7.6 .text**

**7.7 .data**

**7.8 .bss**

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and ``**.**'' moved about by assignment.

**7.9 .comm** *name* **,** *expression*

Provided the *name* is not defined elsewhere, this statement is equivalent to

```
.globl  name
name = expression ^ name
```

That is, the type of *name* is ``undefined external'', and its value is *expression.* In fact the *name* behaves in the current assembly just like an undefined external. However, the link-editor *ld* has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

## 8. Machine instructions

Because of the rather complicated instruction and addressing structure of the PDP-11, the syntax of machine instruction statements is varied. Although the following sections give the syntax in detail, the machine handbooks should be consulted on the semantics.

**8.1 Sources and Destinations**

The syntax of general source and destination addresses is the same. Each must have one of the following forms, where *reg* is a register symbol, and *expr* is any sort of expression:

| syntax | words | mode |
|---|---|---|
| *reg* | 0 | 00+*reg* |
| **(** *reg* **)+** | 0 | 20+*reg* |
| **– (** *reg* **)** | 0 | 40+*reg* |
| *expr* **(** *reg* **)** | 1 | 60+*reg* |
| **(** *reg* **)** | 0 | 10+*reg* |
| * *reg* | 0 | 10+*reg* |
| * **(** *reg* **)+** | 0 | 30+*reg* |
| * **– (** *reg* **)** | 0 | 50+*reg* |
| * **(** *reg* **)** | 1 | 70+*reg* |
| * *expr* **(** *reg* **)** | 1 | 70+*reg* |
| *expr* | 1 | 67 |
| **$** *expr* | 1 | 27 |
| * *expr* | 1 | 77 |
| * **$** *expr* | 1 | 37 |

The *words* column gives the number of address words generated; the *mode* column gives the octal address-mode number. The syntax of the address forms is identical to that in DEC assemblers, except that ‘‘*’’ has been substituted for ‘‘@’’ and ‘‘$’’ for ‘‘#’’; the UNIX typing conventions make ‘‘@’’ and ‘‘#’’ rather inconvenient.

Notice that mode ‘‘*reg’’ is identical to ‘‘(reg)’’; that ‘‘*(reg)’’ generates an index word (namely, 0); and that addresses consisting of an unadorned expression are assembled as pc-relative references independent of the type of the expression. To force a non-relative reference, the form ‘‘*$expr’’ can be used, but notice that further indirection is impossible.

**8.3 Simple machine instructions**

The following instructions are defined as absolute symbols:

**clc**
**clv**
**clz**
**cln**
**sec**
**sev**
**sez**
**sen**

They therefore require no special syntax. The PDP-11 hardware allows more than one of the ‘‘clear’’ class, or alternatively more than one of the ‘‘set’’ class to be **or**-ed together; this may be expressed as follows:

clc | clv

**8.4 Branch**

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot differ from the current location of ‘‘**.**’’ by more than 254 bytes:

| | | |
|---|---|---|
| **br** | **blos** | |
| **bne** | **bvc** | |
| **beq** | **bvs** | |
| **bge** | **bhis** | |
| **blt** | **bec** | (= **bcc**) |
| **bgt** | **bcc** | |
| **ble** | **blo** | |
| **bpl** | **bcs** | |
| **bmi** | **bes** | (= **bcs**) |
| **bhi** | | |

**bes** (‘‘branch on error set’’) and **bec** (‘‘branch on error clear’’) are intended to test the error bit returned by system calls (which is the c-bit).

### 8.5 Extended branch instructions

The following symbols are followed by an expression representing an address in the same segment as ‘‘**.**’’. If the target address is close enough, a branch-type instruction is generated; if the address is too far away, a **jmp** will be used.

| | |
|---|---|
| **jbr** | **jlos** |
| **jne** | **jvc** |
| **jeq** | **jvs** |
| **jge** | **jhis** |
| **jlt** | **jec** |
| **jgt** | **jcc** |
| **jle** | **jlo** |
| **jpl** | **jcs** |
| **jmi** | **jes** |
| **jhi** | |

**jbr** turns into a plain **jmp** if its target is too remote; the others (whose names are contructed by replacing the ‘‘b’’ in the branch instruction’s name by ‘‘j’’) turn into the converse branch over a **jmp** to the target address.

### 8.6 Single operand instructions

The following symbols are names of single-operand machine instructions. The form of address expected is discussed in §8.1 above.

| | |
|---|---|
| clr | sbcb |
| clrb | ror |
| com | rorb |
| comb | rol |
| inc | rolb |
| incb | asr |
| dec | asrb |
| decb | asl |
| neg | aslb |
| negb | jmp |
| adc | swab |
| adcb | tst |
| sbc | tstb |

### 8.7 Double operand instructions

The following instructions take a general source and destination (§8.1), separated by a comma, as operands.

**mov**
**movb**
**cmp**
**cmpb**
**bit**
**bitb**
**bic**
**bicb**
**bis**
**bisb**
**add**
**sub**

### 8.8 Miscellaneous instructions

The following instructions have more specialized syntax. Here *reg* is a register name, *src* and *dst* a general source or destination (§8.1), and *expr* is an expression:

| | | |
|---|---|---|
| **jsr** | *reg,dst* | |
| **rts** | *reg* | |
| **sys** | *expr* | |
| **ash** | *src , reg* | (or, **als**) |
| **ashc** | *src , reg* | (or, **alsc**) |
| **mul** | *src , reg* | (or, **mpy**) |
| **div** | *src , reg* | (or, **dvd**) |
| **xor** | *reg , dst* | |
| **sxt** | *dst* | |
| **mark** | *expr* | |
| **sob** | *reg , expr* | |

**sys** is another name for the **trap** instruction. It is used to code system calls. Its operand is required to be expressible in 6 bits. The expression in **mark** must be expressible in six bits, and the expression in **sob** must be in the same segment as ‘‘**.**’’, must not be external-undefined, must be less than ‘‘**.**’’, and must be within 510 bytes of ‘‘**.**’’.

### 8.9 Floating-point unit instructions

The following floating-point operations are defined, with syntax as indicated:

| | | |
|---|---|---|
| **cfcc** | | |
| **setf** | | |
| **setd** | | |
| **seti** | | |
| **setl** | | |
| **clrf** | *fdst* | |
| **negf** | *fdst* | |
| **absf** | *fdst* | |
| **tstf** | *fsrc* | |
| **movf** | *fsrc, freg* | (= ldf ) |
| **movf** | *freg, fdst* | (= stf ) |
| **movif** | *src, freg* | (= ldcif ) |
| **movfi** | *freg, dst* | (= stcfi ) |
| **movof** | *fsrc, freg* | (= ldcdf ) |
| **movfo** | *freg, fdst* | (= stcfd ) |
| **movie** | *src, freg* | (= ldexp) |
| **movei** | *freg, dst* | (= stexp) |
| **addf** | *fsrc, freg* | |
| **subf** | *fsrc, freg* | |
| **mulf** | *fsrc, freg* | |
| **divf** | *fsrc, freg* | |
| **cmpf** | *fsrc, freg* | |
| **modf** | *fsrc, freg* | |
| **ldfps** | *src* | |
| **stfps** | *dst* | |
| **stst** | *dst* | |

*fsrc*, *fdst*, and *freg* mean floating-point source, destination, and register respectively. Their syntax is identical to that for their non-floating counterparts, but note that only floating registers 0-3 can be a *freg*.

The names of several of the operations have been changed to bring out an analogy with certain fixed-point instructions. The only strange case is **movf**, which turns into either **stf** or **ldf** depending respectively on whether its first operand is or is not a register. Warning: **ldf** sets the floating condition codes, **stf** does not.

## 9. Other symbols

### 9.1 ..

The symbol ‘‘**..**’’ is the *relocation counter*. Just before each assembled word is placed in the output stream, the current value of this symbol is added to the word if the word refers to a text, data or bss segment location. If the output word is a pc-relative address word that refers to an absolute location, the value of ‘‘**..**’’ is subtracted.

Thus the value of ‘‘**..**’’ can be taken to mean the starting memory location of the program. The initial value of ‘‘**..**’’ is 0.

The value of ‘‘**..**’’ may be changed by assignment. Such a course of action is sometimes necessary, but the consequences should be carefully thought out. It is particularly ticklish to change ‘‘**..**’’ midway in an assembly or to do so in a program which will be treated by the loader, which has its own notions of ‘‘**..**’’.

### 9.2 System calls

System call names are not predefined. They may be found in the file */usr/include/sys.s*

## 10. Diagnostics

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

| | |
|---|---|
| ) | parentheses error |
| ] | parentheses error |
| > | string not terminated properly |
| * | indirection ( * ) used illegally |
| **.** | illegal assignment to ‘‘**.**’’ |
| A | error in address |
| B | branch address is odd or too remote |
| E | error in expression |
| F | error in local (‘‘f’’ or ‘‘b’’) type symbol |
| G | garbage (unknown) character |
| I | end of file inside an **.if** |
| M | multiply defined symbol as label |
| O | word quantity assembled at odd address |
| P | phase error— ‘‘**.**’’ different in pass 1 and 2 |
| R | relocation error |
| U | undefined symbol |
| X | syntax error |

# Setting Up Unix – Seventh Edition

*Charles B. Haley*
*Dennis M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

The distribution tape can be used only on a DEC PDP11/45 or PDP11/70 with RP03, RP04, RP05, RP06 disks and with a TU10, TU16, or TE16 tape drive. It consists of some preliminary bootstrapping programs followed by two file system images; if needed, after the initial construction of the file systems individual files can be extracted. (See restor(1))

If you are set up to do it, it might be a good idea immediately to make a copy of the tape to guard against disaster. The tape is 9-track 800 BPI and contains some 512-byte records followed by many 10240-byte records. There are interspersed tapemarks.

The system as distributed contains binary images of the system and all the user level programs, along with source and manual sections for them—about 2100 files altogether. The binary images, along with other things needed to flesh out the file system enough so UNIX will run, are to be put on one file system called the 'root file system'. The file system size required is about 5000 blocks. The file second system has all of the source and documentation. Altogether it amounts to more than 18,000 512-byte blocks.

### Making a Disk From Tape

Perform the following bootstrap procedure to obtain a disk with a root file system on it.

1. Mount the magtape on drive 0 at load point.
2. Mount a formatted disk pack on drive 0.
3. Key in and execute at 100000

| TU10 | TU16/TE16 |
|---|---|
| 012700 | Use the DEC ROM or other |
| 172526 | means to load block 1 |
| 010040 | (i.e. second block) at 800 BPI |
| 012740 | into location 0 and transfer |
| 060003 | to 0. |
| 000777 | |

   The tape should move and the CPU loop. (The TU10 code is *not* the DEC bulk ROM for tape; it reads block 0, not block 1.)

4. If you used the above TU10 code, halt and restart the CPU at 0, otherwise continue to the next step.
5. The console should type

   *Boot*
   *:*

   Copy the magtape to disk by the following procedure. The machine's printouts are shown in italic, explanatory comments are within ( ). Terminate each line you type by carriage return or line-feed. There are two classes of tape drives: the name 'tm' is used for the TU10, and 'ht' is used for the TU16 or TE16. There are also two classes of disks: 'rp' is used for the RP03, and 'hp' is used for the RP04/5/6.

If you should make a mistake while typing, the character '#' erases the last character typed up to the beginning of the line, and the character '@' erases the entire line typed. Some consoles cannot print lower case letters, adjust the instructions accordingly.

(bring in the program mkfs)
*:* tm(0,3) (use 'ht(0,3)' for the TU16/TE16)
*file system size:* 5000
*file system:* rp(0,0) (use 'hp(0,0)' for RP04/5/6)
*isize = XX*
*m/n = XX*
(after a while)
*exit called*
*Boot*
*:*

This step makes an empty file system.

6. The next thing to do is to restore the data onto the new empty file system. To do this you respond to the '*:*' printed in the last step with

   (bring in the program restor)
   *:* tm(0,4) ('ht(0,4)' for TU16/TE16)
   *tape?* tm(0,5) (use 'ht(0,5)' for TU16/TE16)
   *disk?* rp(0,0) (use 'hp(0,0)' for RP04/5/6)
   *Last chance before scribbling on disk.* (you type return)
   (the tape moves, perhaps 5-10 minutes pass)
   *end of tape*
   *Boot*
   *:*

   You now have a UNIX root file system.

**Booting UNIX**

You probably have the bootstrap running, left over from the last step above; if not, repeat the boot process (step 3) again. Then use one of the following:

*:* rp(0,0)rptmunix (for RP03 and TU10)
*:* rp(0,0)rphtunix (for RP03 and TU16/TE16)
*:* hp(0,0)hptmunix (for RP04/5/6 and TU10)
*:* hp(0,0)hphtunix (for RP04/5/6 and TU16/TE16)

The machine should type the following:

*mem = xxx*
#

The *mem* message gives the memory available to user programs in bytes.

UNIX is now running, and the 'UNIX Programmer's manual' applies; references below of the form X(Y) mean the subsection named X in section Y of the manual. The '#' is the prompt from the Shell, and indicates you are the super-user. The user name of the super-user is 'root' if you should find yourself in multi-user mode and need to log in; the password is also 'root'.

To simplify your life later, rename the appropriate version of the system as specified above plain 'unix.' For example, use mv (1) as follows if you have an RP04/5/6 and a TU16 tape:

mv hphtunix unix

In the future, when you reboot, you can type just

```
hp(0,0)unix
```

to the ‘:’ prompt. (Choose appropriately among ‘hp’, ‘rp’, ‘ht’, ‘tm’ according to your configuration).

You now need to make some special file entries in the dev directory. These specify what sort of disk you are running on, what sort of tape drive you have, and where the file systems are. For simplicity, this recipe creates fixed device names. These names will be used below, and some of them are built into various programs, so they are most convenient. However, the names do not always represent the actual major and minor device in the manner suggested in section 4 of the Programmer’s Manual. For example, ‘rp3’ will be used for the name of the file system on which the user file system is put, even though it might be on an RP06 and is not logical device 3. Also, this sequence will put the user file system on the same disk drive as the root, which is not the best place if you have more than one drive. Thus the prescription below should be taken only as one example of where to put things. See also the section on ‘Disk layout’ below.

In any event, change to the dev directory (cd(1)) and, if you like, examine and perhaps change the makefile there (make (1)).

```
cd /dev
cat makefile
```

Then, use one of

```
make rp03
make rp04
make rp05
make rp06
```

depending on which disk you have. Then, use one of

```
make tm
make ht
```

depending on which tape you have. The file ‘rp0’ refers to the root file system; ‘swap’ to the swap-space file system; ‘rp3’ to the user file system. The devices ‘rrp0’ and ‘rrp3’ are the ‘raw’ versions of the disks. Also, ‘mt0’ is tape drive 0, at 800 BPI; ‘rmt0’ is the raw tape, on which large records can be read and written; ‘nrmt0’ is raw tape with the quirk that it does not rewind on close, which is a subterfuge that permits multifile tapes to be handled.

The next thing to do is to extract the rest of the data from the tape. Comments are enclosed in ( ); don’t type these. The number in the first command is the size of the file system; it differs between RP03, RP04/5, and RP06.

```
/etc/mkfs /dev/rp3 74000  (153406 if on RP04/5, 322278 on RP06)
(The above command takes about 2-3 minutes on an RP03)
dd if=/dev/nrmt0 of=/dev/null bs=20b files=6        (skip 6 files on the tape)
restor rf /dev/rmt0 /dev/rp3        (restore the file system)
(Reply with a ‘return’ (CR) to the ‘Last chance’ message)
(The restor takes about 20-30 minutes)
```

All of the data on the tape has been extracted.

You may at this point mount the source file system (mount(1)). To do this type the following:

```
/etc/mount /dev/rp3 /usr
```

The source and manual pages are now available in subdirectories of /usr.

The above mount command is only needed if you intend to play around with source on a single user system, which you are going to do next. The file system is mounted automatically when multi-user mode is entered, by a command in the file /etc/rc. (See ‘Disk Layout’ below).

Before anything further is done the bootstrap block on the disk (block 0) should be filled in. This is done using the command

```
dd if=/usr/mdec/rpuboot of=/dev/rp0 count=1
```

if you have the RP03, or

```
dd if=/usr/mdec/hpuboot of=/dev/rp0 count=1
```

if you have an RP04/5/6. Now the DEC disk bootstraps are usable. See Boot Procedures(8) for further information.

Before UNIX is turned up completely, a few configuration dependent exercises must be performed. At this point, it would be wise to read all of the manuals (especially 'Regenerating System Software') and to augment this reading with hand to hand combat.

**Reconfiguration**

The UNIX system running is configured to run with the given disk and tape, a console, and no other device. This is certainly not the correct configuration. You will have to correct the configuration table to reflect the true state of your machine.

It is wise at this point to know how to recompile the system. Print (cat(1)) the file /usr/sys/conf/makefile. This file is input to the program 'make(1)' which if invoked with 'make all' will recompile all of the system source and install it in the correct libraries.

The program mkconf(1) prepares files that describe a given configuration (See mkconf(1)). In the /usr/sys/conf directory, the four files *xy*conf were input to mkconf to produce the four versions of the system *xy*unix. Pick the appropriate one, and edit it to add lines describing your own configuration. (Remember the console typewriter is automatically included; don't count it in the kl specification.) Then run mkconf; it will generate the files l.s (trap vectors) c.c (configuration table), and mch0.s. Take a careful look at l.s to make sure that all the devices that you have are assembled in the correct interrupt vectors. If your configuration is non-standard, you will have to modify l.s to fit your configuration.

There are certain magic numbers and configuration parameters imbedded in various device drivers that you may want to change. The device addresses of each device are defined in each driver. In case you have any non-standard device addresses, just change the address and recompile. (The device drivers are in the directory /usr/sys/dev.)

The DC11 driver is set to run 4 lines. This can be changed in dc.c.

The DH11 driver is set to handle 3 DH11's with a full complement of 48 lines. If you have less, or more, you may want to edit dh.c.

The DN11 driver will handle 4 DN's. Edit dn.c.

The DU11 driver can only handle a single DU. This cannot be easily changed.

The KL/DL driver is set up to run a single DL11-A, -B, or -C (the console) and no DL11-E's. To change this, edit kl.c to have NKL11 reflect the total number of DL11-ABC's and NDL11 to reflect the number of DL11-E's. So far as the driver is concerned, the difference between the devices is their address.

All of the disk and tape drivers (rf.c, rk.c, rp.c, tm.c, tc.c, hp.c, ht.c) are set up to run 8 drives and should not need to be changed. The big disk drivers (rp.c and hp.c) have partition tables in them which you may want to experiment with.

After all the corrections have been made, use 'make(1)' to recompile the system (or recompile individually if you wish: use the makefile as a guide). If you compiled individually, say 'make unix' in the directory /usr/sys/conf. The final object file (unix) should be moved to the root, and then booted to try it out. It is best to name it /nunix so as not to destroy the working system until you're sure it does work. See Boot Procedures(8) for a discussion of booting. Note: before taking the system down, always (!!) perform a sync(1) to force delayed output to the disk.

**Special Files**

Next you must put in special files for the new devices in the directory /dev using mknod(1). Print the configuration file c.c created above. This is the major device switch of each device class (block and character). There is one line for each device configured in your system and a null line for place holding for those devices not configured. The essential block special files were installed above; for any new devices, the major device number is selected by counting the line number (from zero) of the device's entry in the block configuration table. Thus the first entry in the table bdevsw would be major device zero. This number is also printed in the table along the right margin.

The minor device is the drive number, unit number or partition as described under each device in section 4 of the manual. For tapes where the unit is dial selectable, a special file may be made for each possible selection. You can also add entries for other disk drives.

In reality, device names are arbitrary. It is usually convenient to have a system for deriving names, but it doesn't have to be the one presented above.

Some further notes on minor device numbers. The hp driver uses the 0100 bit of the minor device number to indicate whether or not to interleave a file system across more than one physical device. See hp(4) for more detail. The tm and ht drivers use the 0200 bit to indicate whether or not to rewind the tape when it is closed. The 0100 bit indicates the density of the tape on TU16 drives. By convention, tape special files with the 0200 bit on have an 'n' prepended to their name, as in /dev/nmt0 or /dev/nrmt1. Again, see tm(4) or ht(4).

The naming of character devices is similar to block devices. Here the names are even more arbitrary except that devices meant to be used for teletype access should (to avoid confusion, no other reason) be named /dev/ttyX, where X is some string (as in '00' or 'library'). The files console, mem, kmem, and null are already correctly configured.

The disk and magtape drivers provide a 'raw' interface to the device which provides direct transmission between the user's core and the device and allows reading or writing large records. The raw device counts as a character device, and should have the name of the corresponding standard block special file with 'r' prepended. (The 'n' for no rewind tapes violates this rule.) Thus the raw magtape files would be called /dev/rmtX. These special files should be made.

When all the special files have been created, care should be taken to change the access modes (chmod(1)) on these files to appropriate values (probably 600 or 644).

**Floating Point**

UNIX only supports (and really expects to have) the FP11-B/C floating point unit. For machines without this hardware, there is a user subroutine available that will catch illegal instruction traps and interpret floating point operations. (See fptrap(3).) To install this subroutine in the library, change to /usr/src/libfpsim and execute the shell files

```
compall
mklib
```

The system as delivered does not have this code included in any command, although the operating system adapts automatically to the presence or absence of the FP11.

Next, a floating-point version of the C compiler in /usr/src/cmd/c should be compiled using the commands:

```
cd /usr/src/cmd/c
make fc1
mv fc1 /lib/fc1
```

This allows programs with floating point constants to be compiled. To compile floating point programs use the '−f' flag to cc(1). This flag ensures that the floating point interpreter is loaded with the program and that the floating point version of 'cc' is used.

**Time Conversion**

If your machine is not in the Eastern time zone, you must edit (ed(1)) the file /usr/sys/h/param.h to reflect your local time. The manifest 'TIMEZONE' should be changed to reflect the time difference between local time and GMT in minutes. For EST, this is 5*60; for PST it would be 8*60. Finally, there is a 'DSTFLAG' manifest; when it is 1 it causes the time to shift to Daylight Savings automatically between the last Sundays in April and October (or other algorithms in 1974 and 1975). Normally this will not have to be reset. When the needed changes are done, recompile and load the system using make(1) and install it. (As a general rule, when a system header file is changed, the entire system should be recompiled. As it happens, the only uses of these flags are in /usr/sys/sys/sys4.c, so if this is all that was changed it alone needs to be recompiled.)

You may also want to look at timezone(3) (/usr/src/libc/gen/timezone.c) to see if the name of your timezone is in its internal table. If needed, edit the changes in. After timezone.c has been edited it should be compiled and installed in its library. (See /usr/src/libc/(mklib and compall)) Then you should (at your leisure) recompile and reinstall all programs that use it (such as date(1)).

**Disk Layout**

If there are to be more file systems mounted than just the root and /usr, use mkfs(1) to create any new file system and put its mounting in the file /etc/rc (see init(8) and mount(1)). (You might look at /etc/rc anyway to see what has been provided for you.)

There are two considerations in deciding how to adjust the arrangement of things on your disks: the most important is making sure there is adequate space for what is required; secondarily, throughput should be maximized. Swap space is a critical parameter. The system as distributed has 8778 (hpunix) or 2000 (rpunix) blocks for swap space. This should be large enough so running out of swap space never occurs. You may want to change these if local wisdom indicates otherwise.

The system as distributed has all of the binaries in /bin. Most of them should be moved to /usr/bin, leaving only the ones required for system maintenance (such as icheck, dcheck, cc, ed, restor, etc.) and the most heavily used in /bin. This will speed things up a bit if you have only one disk, and also free up space on the root file system for temporary files. (See below).

Many common system programs (C, the editor, the assembler etc.) create intermediate files in the /tmp directory, so the file system where this is stored also should be made large enough to accommodate most high-water marks. If you leave the root file system as distributed (except as discussed above) there should be no problem. All the programs that create files in /tmp take care to delete them, but most are not immune to events like being hung up upon, and can leave dregs. The directory should be examined every so often and the old files deleted.

Exhaustion of user-file space is certain to occur now and then; the only mechanisms for controlling this phenomenon are occasional use of du(1), df(1), quot(1), threatening messages of the day, and personal letters.

The efficiency with which UNIX is able to use the CPU is largely dictated by the configuration of disk controllers. For general time-sharing applications, the best strategy is to try to split user files, the root directory (including the /tmp directory) and the swap area among three controllers.

Once you have decided how to make best use of your hardware, the question is how to initialize it. If you have the equipment, the best way to move a file system is to dump it (dump(1)) to magtape, use mkfs(1) to create the new file system, and restore (restor(1)) the tape. If for some reason you don't want to use magtape, dump accepts an argument telling where to put the dump; you might use another disk. Sometimes a file system has to be increased in logical size without copying. The super-block of the device has a word giving the highest address which can be allocated. For relatively small increases, this word can be patched using the debugger (adb(1)) and the free list reconstructed using icheck(1). The size should not be increased very greatly by this technique, however, since although the allocatable space will increase the maximum number of files will not (that is, the i-list size can't be changed). Read and understand the description given in file system(5) before playing around in this way. You may want to see section rp(4) for some suggestions on how to lay out the information on RP disks.

If you have to merge a file system into another, existing one, the best bet is to use tar(1). If you must shrink a file system, the best bet is to dump the original and restor it onto the new filesystem. However, this might not work if the i-list on the smaller filesystem is smaller than the maximum allocated inode on the larger. If this is the case, reconstruct the filesystem from scratch on another filesystem (perhaps using tar(1)) and then dump it. If you are playing with the root file system and only have one drive the procedure is more complicated. What you do is the following:

1. GET A SECOND PACK!!!!
2. Dump the current root filesystem (or the reconstructed one) using dump(1).
3. Bring the system down and mount the new pack.
4. Retrieve the WECo distribution tape and perform steps 1 through 5 at the beginning of this document, substituting the desired file system size instead of 5000 when asked for ‘file system size’.
5. Perform step 6 above up to the point where the ‘tape’ question is asked. At this point mount the tape you made just a few minutes ago. Continue with step 6 above substituting a 0 (zero) for the 5.

**New Users**

Install new users by editing the password file /etc/passwd (passwd(5)). This procedure should be done once multi-user mode is entered (see init(8)). You’ll have to make a current directory for each new user and change its owner to the newly installed name. Login as each user to make sure the password file is correctly edited. For example:

```
ed /etc/passwd
$a
joe::10:1::/usr/joe:
w
q
mkdir /usr/joe
chown joe /usr/joe
login joe
ls −la
login root
```

This will make a new login entry for joe, who should be encouraged to use passwd(1) to give himself a password. His default current directory is /usr/joe which has been created. The delivered password file has the user *bin* in it to be used as a prototype.

**Multiple Users**

If UNIX is to support simultaneous access from more than just the console terminal, the file /etc/ttys (ttys(5)) has to be edited. To add a new terminal be sure the device is configured and the special file exists, then set the first character of the appropriate line of /etc/ttys to 1 (or add a new line). Note that init.c will have to be recompiled if there are to be more than 100 terminals. Also note that if the special file is inaccessible when init tries to create a process for it, the system will thrash trying and retrying to open it.

**File System Health**

Periodically (say every day or so) and always after a crash, you should check all the file systems for consistency (icheck, dcheck(1)). It is quite important to execute sync (8) before rebooting or taking the machine down. This is done automatically every 30 seconds by the update program (8) when a multiple-user system is running, but you should do it anyway to make sure.

Dumping of the file system should be done regularly, since once the system is going it is very easy to become complacent. Complete and incremental dumps are easily done with dump(1). Dumping of files by name is best done by tar(1) but the number of files is somewhat limited. Finally if there are enough drives entire disks can be copied using cp(1), or preferably with dd(1) using the raw special files and an appropriate block size.

**Converting Sixth Edition Filesystems**

The best way to convert file systems from 6th edition (V6) to 7th edition (V7) format is to use tar(1). However, a special version of tar must be prepared to run on V6. The following steps will do this:

1. change directories to /usr/src/cmd/tar
2. At the shell prompt respond

   ```
   make v6tar
   ```

   This will leave an executable binary named ‘v6tar’.
3. Mount a scratch tape.
4. Use tp(1) to put ‘v6tar’ on the scratch tape.
5. Bring down V7 and bring up V6.
6. Use tp (on V6) to read in ‘v6tar’. Put it in /bin or /usr/bin (or perhaps some other preferred location).
7. Use v6tar to make tapes of all that you wish to convert. You may want to read the manual section on tar(1) to see whether you want to use blocking or not. Try to avoid using full pathnames when making the tapes. This will simplify moving the hierarchy to some other place on V7 if desired. For example

   ```
   chdir /usr/ken
   v6tar c .
   ```

   is preferable to

   ```
   v6tar c /usr/ken
   ```
8. After all of the desired tapes are made, bring down V6 and reboot V7. Use tar(1) to read in the tapes just made.

**Odds and Ends**

The programs dump, icheck, quot, dcheck, ncheck, and df (source in /usr/source/cmd) should be changed to reflect your default mounted file system devices. Print the first few lines of these programs and the changes will be obvious. Tar should be changed to reflect your desired default tape drive.

Good Luck

Charles B. Haley
Dennis M. Ritchie

# Regenerating System Software

*Charles B. Haley*

*Dennis. M. Ritchie*
*Bell Laboratories*
*Murray Hill, New Jersey 07974*

## Introduction

This document discusses how to assemble or compile various parts of the UNIX† system software. This may be necessary because a command or library is accidentally deleted or otherwise destroyed; also, it may be desirable to install a modified version of some command or library routine. A few commands depend to some degree on the current configuration of the system; thus in any new system modifications to some commands are advisable. Most of the likely modifications relate to the standard disk devices contained in the system. For example, the df(1) ('disk free') command has built into it the names of the standardly present disk storage drives (e.g. '/dev/rf0', '/dev/rp0'). Df(1) takes an argument to indicate which disk to examine, but it is convenient if its default argument is adjusted to reflect the ordinarily present devices. The companion document 'Setting up UNIX' discusses which commands are likely to require changes.

## Where Commands and Subroutines Live

The source files for commands and subroutines reside in several subdirectories of the directory /usr/src. These subdirectories, and a general description of their contents, are

| | |
|---|---|
| cmd | Source files for commands. |
| libc/stdio | Source files making up the 'standard i/o package'. |
| libc/sys | Source files for the C system call interfaces. |
| libc/gen | Source files for most of the remaining routines described in section 3 of the manual. |
| libc/crt | Source files making up the C runtime support package, as in call save-return and long arithmetic. |
| libc/csu | Source for the C startup routines. |
| games | Source for (some of) the games. No great care has been taken to try to make it obvious how to compile these; treat it as a game. |
| libF77 | Source for the Fortran 77 runtime library, exclusive of IO. |
| libI77 | Source for the Fortran 77 IO runtime routines. |
| libdbm | Source for the 'data-base manager' package *dbm* (3). |
| libfpsim | Source for the floating-point simulator routine. |
| libm | Source for the mathematical library. |
| libplot | Source for plotting routines. |

†UNIX is a Trademark of Bell Laboratories.

**Commands**

The regeneration of most commands is straightforward. The 'cmd' directory will contain either a source file for the command or a subdirectory containing the set of files that make up the command. If it is a single file the command

```
cd /usr/src/cmd
cmake cmd_name
```

suffices. (Cmd_name is the name of the command you are playing with.) The result of the cmake command will be an executable version. If you type

```
cmake −cp cmd_name
```

the result will be copied to /bin (or perhaps /etc or other places if appropriate).

If the source files are in a subdirectory there will be a 'makefile' (see make(1)) to control the regeneration. After changing to the proper directory (cd(1)) you type one of the following:

make all The program is compiled and loaded; the executable is left in the current directory.

make cp The program is compiled and loaded, and the executable is installed. Everything is cleaned up afterwards; for example .o files are deleted.

make cmp The program is compiled and loaded, and the executable is compared against the one in /bin.

Some of the makefiles have other options. Print (cat(1)) the ones you are interested in to find out.

**The Assembler**

The assembler consists of two executable files: /bin/as and /lib/as2. The first is the 0-th pass: it reads the source program, converts it to an intermediate form in a temporary file '/tmp/atm0?', and estimates the final locations of symbols. It also makes two or three other temporary files which contain the ordinary symbol table, a table of temporary symbols (like 1:) and possibly an overflow intermediate file. The program /lib/as2 acts as an ordinary multiple pass assembler with input taken from the files produced by /bin/as.

The source files for /bin/as are named '/usr/src/cmd/as/as1?.s' (there are 9 of them); /lib/as2 is produced from the source files '/usr/src/cmd/as/as2?.s'; they likewise are 9 in number. Considerable care should be exercised in replacing either component of the assembler. Remember that if the assembler is lost, the only recourse is to replace it from some backup storage; a broken assembler cannot assemble itself.

**The C Compiler**

The C compiler consists of seven routines: '/bin/cc', which calls the phases of the compiler proper, the compiler control line expander '/lib/cpp', the assembler ('as'), and the loader ('ld'). The phases of the C compiler are '/lib/c0', which is the first phase of the compiler; '/lib/c1', which is the second phase of the compiler; and '/lib/c2', which is the optional third phase optimizer. The loss of the C compiler is as serious as that of the assembler.

The source for /bin/cc resides in '/usr/src/cmd/cc.c'. Its loss alone (or that of c2) is not fatal. If needed, prog.c can be compiled by

```
/lib/cpp prog.c >temp0
/lib/c0 temp0 temp1 temp2
/lib/c1 temp1 temp2 temp3
as − temp3
ld −n /lib/crt0.o a.out −lc
```

The source for the compiler proper is in the directory /usr/src/cmd/c. The first phase (/lib/c0) is generated from the files c00.c, ..., c05.c, which must be compiled by the C compiler. There is also c0.h, a header file *included* by the C programs of the first phase. To make a new /lib/c0 use

```
make c0
```

Before installing the new c0, it is prudent to save the old one someplace.

The second phase of C (/lib/c1) is generated from the source files c10.c, ..., c13.c, the include-file c1.h, and a set of object-code tables combined into table.o. To generate a new second phase use

```
make c1
```

It is likewise prudent to save c1 before installing a new version. In fact in general it is wise to save the object files for the C compiler so that if disaster strikes C can be reconstituted without a working version of the compiler.

In a similar manner, the third phase of the C compiler (/lib/c2) is made up from the files c20.c and c21.c together with c2.h. Its loss is not critical since it is completely optional.

The set of tables mentioned above is generated from the file table.s. This '.s' file is not in fact assembler source; it must be converted by use of the *cvopt* program, whose source and object are located in the C directory. Normally this is taken care of by make(1). You might want to look at the makefile to see what it does.

**UNIX**

The source and object programs for UNIX are kept in four subdirectories of */usr/sys*. In the subdirectory *h* there are several files ending in '.h'; these are header files which are picked up (via '#include ...') as required by each system module. The subdirectory *dev* consists mostly of the device drivers together with a few other things. The subdirectory *sys* is the rest of the system. There are files of the form LIBx in the directories sys and dev. These are archives (ar(1)) which contain the object versions of the routines in the directory.

Subdirectory *conf* contains the files which control device configuration of the system. *L.s* specifies the contents of the interrupt vectors; *c.c* contains the tables which relate device numbers to handler routines. A third file, *mch.s*, contains all the machine-language code in the system. A fourth file, *mch0.s*, is generated by mkconf(1) and contains flags indicating what sort of tape drive is available for taking crash dumps.

There are two ways to recreate the system. Use

```
cd /usr/sys/conf
make unix
```

if the libraries /usr/sys/dev/LIB2 and /usr/sys/sys/LIB1, and also c.o and l.o, are correct. Use

```
cd /usr/sys/conf
make all
```

to recompile everything and recreate the libraries from scratch. This is needed, for example, when a header included in several source files is changed. See 'Setting Up UNIX' for other information about configuration and such.

When the make is done, the new system is present in the current directory as 'unix'. It should be tested before destroying the currently running '/unix', this is best done by doing something like

```
mv /unix /ounix
mv unix /unix
```

If the new system doesn't work, you can still boot 'ounix' and come up (see boot(8)). When you have satisfied yourself that the new system works, remove /ounix.

To install a new device driver, compile it and put it into its library. The best way to put it into the library is to use the command

```
ar uv LIB2 x.o
```

where x is the routine you just compiled. (All the device drivers distributed with the system are already in the library.)

Next, the device's interrupt vector must be entered in l.s. This is probably already done by the routine mkconf(1), but if the device is esoteric or nonstandard you will have to massage l.s by hand. This involves placing a pointer to a callout routine and the device's priority level in the vector. Use some other device (like the console) as a guide. Notice that the entries in l.s must be in order as the assembler does not permit moving the location counter '.' backwards. The assembler also does not permit assignation of an absolute number to '.', which is the reason for the '. = ZERO+100' subterfuge. If a constant smaller than 16(10) is added to the priority level, this number will be available as the first argument of the interrupt routine. This stratagem is used when several similar devices share the same interrupt routine (as in dl11's).

If you have to massage l.s, be sure to add the code to actually transfer to the interrupt routine. Again use the console as a guide. The apparent strangeness of this code is due to running the kernel in separate I&D space. The *call* routine saves registers as required and prepares a C-style call on the actual interrupt routine named after the 'jmp' instruction. When the routine returns, *call* restores the registers and performs an rti instruction. As an aside, note that external names in C programs have an underscore ('_') prepended to them.

The second step which must be performed to add a device unknown to mkconf is to add it to the configuration table /usr/sys/conf/c.c. This file contains two subtables, one for block-type devices, and one for character-type devices. Block devices include disks, DECtape, and magtape. All other devices are character devices. A line in each of these tables gives all the information the system needs to know about the device handler; the ordinal position of the line in the table implies its major device number, starting at 0.

There are four subentries per line in the block device table, which give its open routine, close routine, strategy routine, and device table. The open and close routines may be nonexistent, in which case the name 'nulldev' is given; this routine merely returns. The strategy routine is called to do any I/O, and the device table contains status information for the device.

For character devices, each line in the table specifies a routine for open, close, read, and write, and one which sets and returns device-specific status (used, for example, for stty and gtty on typewriters). If there is no open or close routine, 'nulldev' may be given; if there is no read, write, or status routine, 'nodev' may be given. Nodev sets an error flag and returns.

The final step which must be taken to install a device is to make a special file for it. This is done by mknod(1), to which you must specify the device class (block or character), major device number (relative line in the configuration table) and minor device number (which is made available to the driver at appropriate times).

The documents 'Setting up Unix' and 'The Unix IO system' may aid in comprehending these steps.

**The Library libc.a**

The library /lib/libc.a is where most of the subroutines described in sections 2 and 3 of the manual are kept. This library can be remade using the following commands:

```
cd /usr/src/libc
sh compall
sh mklib
mv libc.a /lib/libc.a
```

If single routines need to be recompiled and replaced, use

```
cc −c −O x.c
ar vr /lib/libc.a x.o
rm x.o
```

The above can also be used to put new items into the library. See ar(1), lorder(1), and tsort(1).

The routines in /usr/src/cmd/libc/csu (C start up) are not in libc.a. These are separately assembled and put into /lib. The commands to do this are

```
cd /usr/src/libc/csu
as − x.s
mv a.out /lib/x
```

where x is the routine you want.

**Other Libraries**

Likewise, the directories containing the source for the other libraries have files compall (that recompiles everything) and mklib (that recreates the library).

**System Tuning**

There are several tunable parameters in the system. These set the size of various tables and limits. They are found in the file /usr/sys/h/param.h as manifests ('#define's). Their values are rather generous in the system as distributed. Our typical maximum number of users is about 20, but there are many daemon processes.

When any parameter is changed, it is prudent to recompile the entire system, as discussed above. A brief discussion of each follows:

NBUF This sets the size of the disk buffer cache. Each buffer is 512 bytes. This number should be around 25 plus NMOUNT, or as big as can be if the above number of buffers cause the system to not fit in memory.

NFILE This sets the maximum number of open files. An entry is made in this table every time a file is 'opened' (see open(2), creat(2)). Processes share these table entries across forks (fork(2)). This number should be about the same size as NINODE below. (It can be a bit smaller.)

NMOUNT This indicates the maximum number of mounted file systems. Make it big enough that you don't run out at inconvenient times.

MAXMEM This sets an administrative limit on the amount of memory a process may have. It is set automatically if the amount of physical memory is small, and thus should not need to be changed.

MAXUPRC This sets the maximum number of processes that any one user can be running at any one time. This should be set just large enough that people can get work done but not so large that a user can hog all the processes available (usually by accident!).

NPROC This sets the maximum number of processes that can be active. It depends on the demand pattern of the typical user; we seem to need about 8 times the number of terminals.

NINODE This sets the size of the inode table. There is one entry in the inode table for every open device, current working directory, sticky text segment, open file, and mounted device. Note that if two users have a file open there is still only one entry in the inode table. A reasonable rule of thumb for the size of this table is

NPROC + NMOUNT + (number of terminals)

SSIZE The initial size of a process stack. This may be made bigger if commonly run processes have large data areas on the stack.

SINCR The size of the stack growth increment.

NOFILE This sets the maximum number of files that any one process can have open. 20 is plenty.

CANBSIZ This is the size of the typewriter canonicalization buffer. It is in this buffer that erase and kill processing is done. Thus this is the maximum size of an input typewriter line. 256 is usually plenty.

CMAPSIZ The number of fragments that memory can be broken into. This should be big enough that it never runs out. The theoretical maximum is twice the number of processes, but this is a vast overestimate in practice. 50 seems enough.

SMAPSIZ Same as CMAPSIZ except for secondary (swap) memory.

NCALL This is the size of the callout table. Callouts are entered in this table when some sort of internal system timing must be done, as in carriage return delays for terminals. The number must be big enough to handle all such requests.

NTEXT The maximum number of simultaneously executing pure programs. This should be big enough so as to not run out of space under heavy load. A reasonable rule of thumb is about

(number of terminals) + (number of sticky programs)

NCLIST The number of clist segments. A clist segment is 6 characters. NCLIST should be big enough so that the list doesn't become exhausted when the machine is busy. The characters that have arrived from a terminal and are waiting to be given to a process live here. Thus enough space should be left so that every terminal can have at least one average line pending (about 30 or 40 characters).

TIMEZONE The number of minutes westward from Greenwich. See 'Setting Up UNIX'.

DSTFLAG See 'Setting Up UNIX' section on time conversion.

MSGBUFS The maximum number of characters of system error messages saved. This is used as a circular buffer.

NCARGS The maximum number of characters in an exec(2) arglist. This number controls how many arguments can be passed into a process. 5120 is practically infinite.

HZ Set to the frequency of the system clock (e.g., 50 for a 50 Hz. clock).

# UNIX Implementation

*K. Thompson*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

This paper describes in high-level terms the implementation of the resident UNIX† kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

## 1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression ‘‘the UNIX operating system.’’ The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on ‘‘the way things should be done.’’ Even so, if ‘‘the way’’ is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

## 2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

---

†UNIX is a Trademark of Bell Laboratories.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

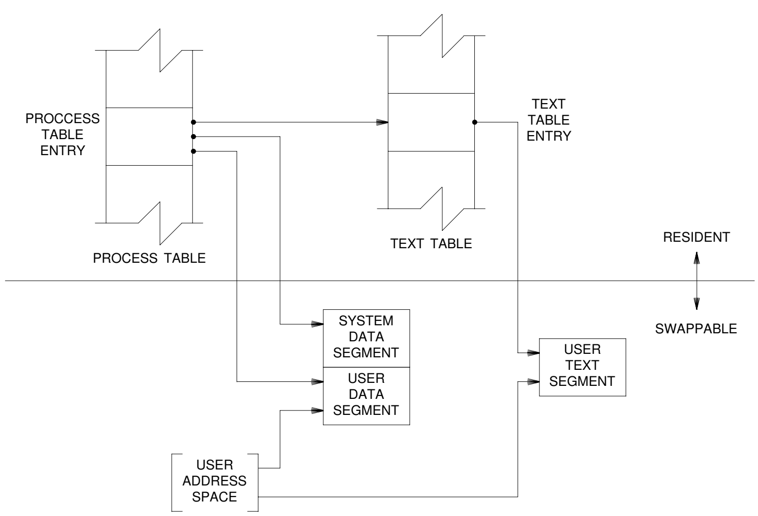


Fig. 1—Process control data structure.

### 2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **exec**s the second program. This is analogous to a ''goto.'' If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a ''call.'' Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

### 2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

### 2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of ''how much'' can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[3]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

## 3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been ''structured I/O'' and ''unstructured I/O,'' respectively; while the term ''block I/O'' has some meaning, ''character I/O'' is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

### 3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as ''dirty.'' The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is ''cured'' by allowing only one outstanding write request per drive.

### 3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the ''classical'' character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means ''everything other than block.'' I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

#### 3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

#### 3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

### 3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or ‘‘borrow’’ block I/O buffers for a while and then give them back.

## 4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a ‘‘disk’’ is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called ‘‘super-block.’’ This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases.

Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a ‘‘triple indirect’’ address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.,* 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

### 4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2.

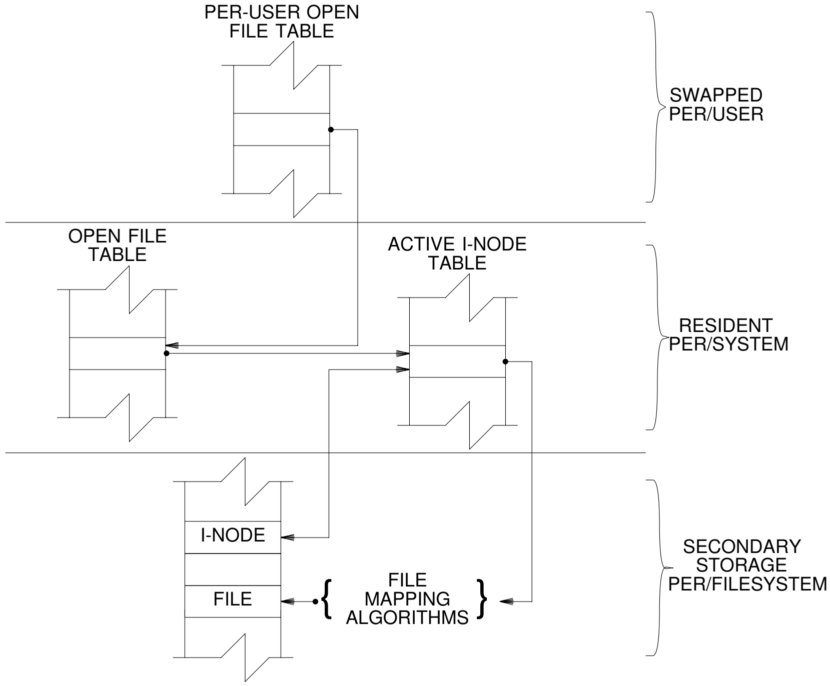


Fig. 2—File system data structure.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **fork**s) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe.** Implementation of **pipe**s consists of implied **seek**s before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

### 4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

### 4.3. Other system functions

There are some other things that the system does for the user – a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[5] Each user may have his own command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing ''system'' code with general user primitives comes directly from MULTICS.[6]

### References

1. R. E. Griswold and D. R. Hanson, ''An Overview of SL5,'' *SIGPLAN Notices* **12**(4), pp.40-50 (April 1977).

2. E. W. Dijkstra, ''Cooperating Sequential Processes,'' pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, New York (1968).

3. J. A. Hawley and W. B. Meyer, ''MUNIX, A Multiprocessing Version of UNIX,'' M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).

4. D. M. Ritchie and K. Thompson, ''The UNIX Time-Sharing System,'' *Bell Sys. Tech. J.* **57**(6), pp.1905-1929 (1978).

5. S. R. Bourne, ''UNIX Time-Sharing System: The UNIX Shell,'' *Bell Sys. Tech. J.* **57**(6), pp.1971-1990 (1978).

6. E. I. Organick, *The MULTICS System,* M.I.T. Press, Cambridge, Mass. (1972).

# The UNIX I/O System

*Dennis M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

This paper gives an overview of the workings of the UNIX† I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper ''The UNIX Time-sharing System.'' A more detailed discussion appears in ''UNIX Implementation;'' the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

## Device Classes

There are two classes of device: *block* and *character.* The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

## Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u.* This table is indexed by the file descriptor returned by the *open* or *creat,* and is accessed during a *read, write,* or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat.* This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or

†UNIX is a Trademark of Bell Laboratories.

write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

Certain open files can be designated ''multiplexed'' files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

**Character Device Drivers**

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which performs the final *close.*

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied that indicates, if *on,* that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count.*

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset, count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write,* except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The ''special-functions'' routine is invoked by the *stty* and *gtty* system calls as follows: *(*p) (dev, v)* where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2].*

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the ''new PS'' word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their ''top half'' (read/write) and ''bottom half'' (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc.* A queue header has the structure

```
struct {
        int     c_cc;   /* character count */
        char    *c_cf;  /* first character */
        char    *c_cl;  /* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority.*

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area ‘‘*u.*’’ should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup,* (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( ), spl5( ), spl6( ), spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg).* Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

**The Block-device Interface**

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf.* Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are ‘‘free,’’ that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made ‘‘busy,’’ so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The *breada* routine is used to implement read-ahead. it is logically similar to *bread,* but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread.* There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

*Bwrite* is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_PHYS This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

B_MAP This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.

B_WANTEDThis flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_AGE This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_ASYNC This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRI This bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

**Block Device Drivers**

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup.* In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back),* and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

**Raw Block-device I/O**

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the ''appropriate information'' to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat,* the buffer pointer *bp,* the device number *dev,* and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE.* *Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

# A Tour through the UNIX† C Compiler

*D. M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

### The Intermediate Language

Communication between the two phases of the compiler proper is carried out by means of a pair of intermediate files. These files are treated as having identical structure, although the second file contains only the code generated for strings. It is convenient to write strings out separately to reduce the need for multiple location counters in a later assembly phase.

The intermediate language is not machine-independent; its structure in a number of ways reflects the fact that C was originally a one-pass compiler chopped in two to reduce the maximum memory requirement. In fact, only the latest version of the compiler has a complete intermediate language at all. Until recently, the first phase of the compiler generated assembly code for those constructions it could deal with, and passed expression parse trees, in absolute binary form, to the second phase for code generation. Now, at least, all inter-phase information is passed in a describable form, and there are no absolute pointers involved, so the coupling between the phases is not so strong.

The areas in which the machine (and system) dependencies are most noticeable are

1. Storage allocation for automatic variables and arguments has already been performed, and nodes for such variables refer to them by offset from a display pointer. Type conversion (for example, from integer to pointer) has already occurred using the assumption of byte addressing and 2-byte words.

2. Data representations suitable to the PDP-11 are assumed; in particular, floating point constants are passed as four words in the machine representation.

As it happens, each intermediate file is represented as a sequence of binary numbers without any explicit demarcations. It consists of a sequence of conceptual lines, each headed by an operator, and possibly containing various operands. The operators are small numbers; to assist in recognizing failure in synchronization, the high-order byte of each operator word is always the octal number 376. Operands are either 16-bit binary numbers or strings of characters representing names. Each name is terminated by a null character. There is no alignment requirement for numerical operands and so there is no padding after a name string.

The binary representation was chosen to avoid the necessity of converting to and from character form and to minimize the size of the files. It would be very easy to make each operator-operand ‘line’ in the file be a genuine, printable line, with the numbers in octal or decimal; this in fact was the representation originally used.

The operators fall naturally into two classes: those which represent part of an expression, and all others. Expressions are transmitted in a reverse-Polish notation; as they are being read, a tree is built which is isomorphic to the tree constructed in the first phase. Expressions are passed as a whole, with no non-expression operators intervening. The reader maintains a stack; each leaf of the expression tree (name, constant) is pushed on the stack; each unary operator replaces the top of the stack by a node whose operand is the old top-of-stack; each binary operator replaces the top pair on the stack with a single entry. When the expression is complete there is exactly one item on the stack. Following each

---

†UNIX is a Trademark of Bell Laboratories.

expression is a special operator which passes the unique previous expression to the 'optimizer' described below and then to the code generator.

Here is the list of operators not themselves part of expressions.

**EOF**

marks the end of an input file.

**BDATA** *flag data ...*

specifies a sequence of bytes to be assembled as static data. It is followed by pairs of words; the first member of the pair is non-zero to indicate that the data continue; a zero flag is not followed by data and terminates the operator. The data bytes occupy the low-order part of a word.

**WDATA** *flag data ...*

specifies a sequence of words to be assembled as static data; it is identical to the BDATA operator except that entire words, not just bytes, are passed.

**PROG**

means that subsequent information is to be compiled as program text.

**DATA**

means that subsequent information is to be compiled as static data.

**BSS**

means that subsequent information is to be compiled as unitialized static data.

**SYMDEF** *name*

means that the symbol *name* is an external name defined in the current program. It is produced for each external data or function definition.

**CSPACE** *name size*

indicates that the name refers to a data area whose size is the specified number of bytes. It is produced for external data definitions without explicit initialization.

**SSPACE** *size*

indicates that *size* bytes should be set aside for data storage. It is used to pad out short initializations of external data and to reserve space for static (internal) data. It will be preceded by an appropriate label.

**EVEN**

is produced after each external data definition whose size is not an integral number of words. It is not produced after strings except when they initialize a character array.

**NLABEL** *name*

is produced just before a BDATA or WDATA initializing external data, and serves as a label for the data.

**RLABEL** *name*

is produced just before each function definition, and labels its entry point.

**SNAME** *name number*

is produced at the start of each function for each static variable or label declared therein. Subsequent uses of the variable will be in terms of the given number. The code generator uses this only to produce a debugging symbol table.

**ANAME** *name number*

Likewise, each automatic variable's name and stack offset is specified by this operator. Arguments count as automatics.

**RNAME** *name number*

Each register variable is similarly named, with its register number.

**SAVE** *number*

produces a register-save sequence at the start of each function, just after its label (RLABEL).

**SETREG** *number*

is used to indicate the number of registers used for register variables. It actually gives the register number of the lowest free register; it is redundant because the RNAME operators could be counted instead.

**PROFIL**

is produced before the save sequence for functions when the profile option is turned on. It produces code to count the number of times the function is called.

**SWIT** *deflab line label value ...*

is produced for switches. When control flows into it, the value being switched on is in the register forced by RFORCE (below). The switch statement occurred on the indicated line of the source, and the label number of the default location is *deflab.* Then the operator is followed by a sequence of label-number and value pairs; the list is terminated by a 0 label.

**LABEL** *number*

generates an internal label. It is referred to elsewhere using the given number.

**BRANCH** *number*

indicates an unconditional transfer to the internal label number given.

**RETRN**

produces the return sequence for a function. It occurs only once, at the end of each function.

**EXPR** *line*

causes the expression just preceding to be compiled. The argument is the line number in the source where the expression occurred.

**NAME** *class type name*

**NAME** *class type number*

indicates a name occurring in an expression. The first form is used when the name is external; the second when the name is automatic, static, or a register. Then the number indicates the stack offset, the label number, or the register number as appropriate. Class and type encoding is described elsewhere.

**CON** *type value*

transmits an integer constant. This and the next two operators occur as part of expressions.

**FCON** *type 4-word-value*

transmits a floating constant as four words in PDP-11 notation.

**SFCON** *type value*

transmits a floating-point constant whose value is correctly represented by its high-order word in PDP-11 notation.

**NULL**

indicates a null argument list of a function call in an expression; call is a binary operator whose second operand is the argument list.

**CBRANCH** *label cond*

produces a conditional branch. It is an expression operator, and will be followed by an EXPR. The branch to the label number takes place if the expression's truth value is the same as that of *cond.* That is, if *cond=1* and the expression evaluates to true, the branch is taken.

**binary-operator** *type*

There are binary operators corresponding to each such source-language operator; the type of the result of each is passed as well. Some perhaps-unexpected ones are: COMMA, which is a right-associative operator designed to simplify right-to-left evaluation of function arguments; prefix and postfix ++ and −−, whose second operand is the increment amount, as a CON; QUEST and COLON, to express the conditional expression as 'a?(b:c)'; and a sequence of special operators for expressing relations between pointers, in case pointer comparison is different from integer comparison (e.g. unsigned).

**unary-operator** *type*

There are also numerous unary operators. These include ITOF, FTOI, FTOL, LTOF, ITOL, LTOI which convert among floating, long, and integer; JUMP which branches indirectly through a label expression; INIT, which compiles the value of a constant expression used as an initializer; RFORCE, which is used before a return sequence or a switch to place a value in an agreed-upon register.

**Expression Optimization**

Each expression tree, as it is read in, is subjected to a fairly comprehensive analysis. This is performed by the *optim* routine and a number of subroutines; the major things done are

1. Modifications and simplifications of the tree so its value may be computed more efficiently and conveniently by the code generator.
2. Marking each interior node with an estimate of the number of registers required to evaluate it. This register count is needed to guide the code generation algorithm.

One thing that is definitely not done is discovery or exploitation of common subexpressions, nor is this done anywhere in the compiler.

The basic organization is simple: a depth-first scan of the tree. *Optim* does nothing for leaf nodes (except for automatics; see below), and calls *unoptim* to handle unary operators. For binary operators, it calls itself to process the operands, then treats each operator separately. One important case is commutative and associative operators, which are handled by *acommute.*

Here is a brief catalog of the transformations carried out by by *optim* itself. It is not intended to be complete. Some of the transformations are machine-dependent, although they may well be useful on machines other than the PDP-11.

1. As indicated in the discussion of *unoptim* below, the optimizer can create a node type corresponding to the location addressed by a register plus a constant offset. Since this is precisely the implementation of automatic variables and arguments, where the register is fixed by convention, such variables are changed to the new form to simplify later processing.
2. Associative and commutative operators are processed by the special routine *acommute.*
3. After processing by *acommute,* the bitwise & operator is turned into a new *andn* operator; 'a & b' becomes 'a *andn* ~b'. This is done because the PDP-11 provides no *and* operator, but only *andn.* A similar transformation takes place for '=&'.
4. Relationals are turned around so the more complicated expression is on the left. (So that '2 > f(x)' becomes 'f(x) < 2'). This improves code generation since the algorithm prefers to have the right operand require fewer registers than the left.
5. An expression minus a constant is turned into the expression plus the negative constant, and the *acommute* routine is called to take advantage of the properties of addition.
6. Operators with constant operands are evaluated.
7. Right shifts (unless by 1) are turned into left shifts with a negated right operand, since the PDP-11 lacks a general right-shift operator.
8. A number of special cases are simplified, such as division or multiplication by 1, and shifts by 0.

The *unoptim* routine performs the same sort of processing for unary operators.

1. '*&x' and '&*x' are simplified to 'x'.
2. If *r* is a register and *c* is a constant or the address of a static or external variable, the expressions '*(r+c)' and '*r' are turned into a special kind of name node which expresses the name itself and the offset. This simplifies subsequent processing because such constructions can appear as the the address of a PDP-11 instruction.
3. When the unary '&' operator is applied to a name node of the special kind just discussed, it is reworked to make the addition explicit again; this is done because the PDP-11 has no 'load address' instruction.
4. Constructions like '*r++' and '*−−r' where *r* is a register are discovered and marked as being implementable using the PDP-11 auto-increment and -decrement modes.
5. If '!' is applied to a relational, the '!' is discarded and the sense of the relational is reversed.
6. Special cases involving reflexive use of negation and complementation are discovered.
7. Operations applying to constants are evaluated.

The *acommute* routine, called for associative and commutative operators, discovers clusters of the same operator at the top levels of the current tree, and arranges them in a list: for 'a+((b+c)+(d+f))' the list would be'a,b,c,d,e,f'. After each subtree is optimized, the list is sorted in decreasing difficulty of computation; as mentioned above, the code generation algorithm works best when left operands are the difficult ones. The 'degree of difficulty' computed is actually finer than the mere number of registers required; a constant is considered simpler than the address of a static or external, which is simpler than reference to a variable. This makes it easy to fold all the constants together, and also to merge together the sum of a constant and the address of a static or external (since in such nodes there is space for an 'offset' value). There are also special cases, like multiplication by 1 and addition of 0.

A special routine is invoked to handle sums of products. *Distrib* is based on the fact that it is better to compute ‘c1*c2*x + c1*y’ as ‘c1*(c2*x + y)’ and makes the divisibility tests required to assure the correctness of the transformation. This transformation is rarely possible with code directly written by the user, but it invariably occurs as a result of the implementation of multi-dimensional arrays.

Finally, *acommute* reconstructs a tree from the list of expressions which result.

**Code Generation**

The grand plan for code-generation is independent of any particular machine; it depends largely on a set of tables. But this fact does not necessarily make it very easy to modify the compiler to produce code for other machines, both because there is a good deal of machine-dependent structure in the tables, and because in any event such tables are non-trivial to prepare.

The arguments to the basic code generation routine *rcexpr* are a pointer to a tree representing an expression, the name of a code-generation table, and the number of a register in which the value of the expression should be placed. *Rcexpr* returns the number of the register in which the value actually ended up; its caller may need to produce a *mov* instruction if the value really needs to be in the given register. There are four code generation tables.

*Regtab* is the basic one, which actually does the job described above: namely, compile code which places the value represented by the expression tree in a register.

*Cctab* is used when the value of the expression is not actually needed, but instead the value of the condition codes resulting from evaluation of the expression. This table is used, for example, to evaluate the expression after *if.* It is clearly silly to calculate the value (0 or 1) of the expression ‘a==b’ in the context ‘if (a==b) ... ’

The *sptab* table is used when the value of an expression is to be pushed on the stack, for example when it is an actual argument. For example in the function call ‘f(a)’ it is a bad idea to load *a* into a register which is then pushed on the stack, when there is a single instruction which does the job.

The *efftab* table is used when an expression is to be evaluated for its side effects, not its value. This occurs mostly for expressions which are statements, which have no value. Thus the code for the statement ‘a = b’ need produce only the approoriate *mov* instruction, and need not leave the value of *b* in a register, while in the expression ‘a + (b = c)’ the value of ‘b = c’ will appear in a register.

All of the tables besides *regtab* are rather small, and handle only a relatively few special cases. If one of these subsidiary tables does not contain an entry applicable to the given expression tree, *rcexpr* uses *regtab* to put the value of the expression into a register and then fixes things up; nothing need be done when the table was *efftab,* but a *tst* instruction is produced when the table called for was *cctab,* and a *mov* instruction, pushing the register on the stack, when the table was *sptab.*

The *rcexpr* routine itself picks off some special cases, then calls *cexpr* to do the real work. *Cexpr* tries to find an entry applicable to the given tree in the given table, and returns $-1$ if no such entry is found, letting *rcexpr* try again with a different table. A successful match yields a string containing both literal characters which are written out and pseudo-operations, or macros, which are expanded. Before studying the contents of these strings we will consider how table entries are matched against trees.

Recall that most non-leaf nodes in an expression tree contain the name of the operator, the type of the value represented, and pointers to the subtrees (operands). They also contain an estimate of the number of registers required to evaluate the expression, placed there by the expression-optimizer routines. The register counts are used to guide the code generation process, which is based on the Sethi-Ullman algorithm.

The main code generation tables consist of entries each containing an operator number and a pointer to a subtable for the corresponding operator. A subtable consists of a sequence of entries, each with a key describing certain properties of the operands of the operator involved; associated with the key is a code string. Once the subtable corresponding to the operator is found, the subtable is searched linearly until a key is found such that the properties demanded by the key are compatible with the operands of the tree node. A successful match returns the code string; an unsuccessful search, either for the operator in the main table or a compatble key in the subtable, returns a failure indication.

The tables are all contained in a file which must be processed to obtain an assembly language program. Thus they are written in a special-purpose language. To provided definiteness to the following discussion, here is an example of a subtable entry.

```
%n,aw
        F
        add     A2,R
```

The '%' indicates the key; the information following (up to a blank line) specifies the code string. Very briefly, this entry is in the subtable for '+' of *regtab;* the key specifies that the left operand is any integer, character, or pointer expression, and the right operand is any word quantity which is directly addressible (e.g. a variable or constant). The code string calls for the generation of the code to compile the left (first) operand into the current register ('F') and then to produce an 'add' instruction which adds the second operand ('A2') to the register ('R'). All of the notation will be explained below.

Only three features of the operands are used in deciding whether a match has occurred. They are:

1. Is the type of the operand compatible with that demanded?
2. Is the 'degree of difficulty' (in a sense described below) compatible?
3. The table may demand that the operand have a '*' (indirection operator) as its highest operator.

As suggested above, the key for a subtable entry is indicated by a '%,' and a comma-separated pair of specifications for the operands. (The second specification is ignored for unary operators). A specification indicates a type requirement by including one of the following letters. If no type letter is present, any integer, character, or pointer operand will satisfy the requirement (not float, double, or long).

b A byte (character) operand is required.

w A word (integer or pointer) operand is required.

f A float or double operand is required.

d A double operand is required.

l A long (32-bit integer) operand is required.

Before discussing the 'degree of difficulty' specification, the algorithm has to be explained more completely. *Rcexpr* (and *cexpr)* are called with a register number in which to place their result. Registers 0, 1, ... are used during evaluation of expressions; the maximum register which can be used in this way depends on the number of register variables, but in any event only registers 0 through 4 are available since r5 is used as a stack frame header and r6 (sp) and r7 (pc) have special hardware properties. The code generation routines assume that when called with register *n* as argument, they may use *n+1,* ... (up to the first register variable) as temporaries. Consider the expression 'X+Y', where both X and Y are expressions. As a first approximation, there are three ways of compiling code to put this expression in register *n.*

1. If Y is an addressible cell, (recursively) put X into register *n* and add Y to it.
2. If Y is an expression that can be calculated in *k* registers, where *k* smaller than the number of registers available, compile X into register *n,* Y into register *n+1,* and add register *n+1* to *n.*
3. Otherwise, compile Y into register *n,* save the result in a temporary (actually, on the stack) compile X into register *n,* then add in the temporary.

The distinction between cases 2 and 3 therefore depends on whether the right operand can be compiled in fewer than *k* registers, where *k* is the number of free registers left after registers 0 through *n* are taken: 0 through *n–1* are presumed to contain already computed temporary results; *n* will, in case 2, contain the value of the left operand while the right is being evaluated.

These considerations should make clear the specification codes for the degree of difficulty, bearing in mind that a number of special cases are also present:

z is satisfied when the operand is zero, so that special code can be produced for expressions like 'x = 0'.

1 is satisfied when the operand is the constant 1, to optimize cases like left and right shift by 1, which can be done efficiently on the PDP-11.

c is satisfied when the operand is a positive (16-bit) constant; this takes care of some special cases in long arithmetic.

a is satisfied when the operand is addressible; this occurs not only for variables and constants, but also for some more complicated constructions, such as indirection through a simple variable, '*p++' where *p* is a register variable (because of the PDP-11's auto-increment address mode), and '*(p+c)' where *p* is a register and *c* is a constant. Precisely, the requirement is that the operand refers to a cell whose address can be written as a source or destination of a PDP-11 instruction.

e is satisfied by an operand whose value can be generated in a register using no more than *k* registers, where *k* is the number of registers left (not counting the current register). The 'e' stands for 'easy.'

n is satisfied by any operand. The 'n' stands for 'anything.'

These degrees of difficulty are considered to lie in a linear ordering and any operand which satisfies an earlier-mentioned requirement will satisfy a later one. Since the subtables are searched linearly, if a '1' specification is included, almost certainly a 'z' must be written first to prevent expressions containing the constant 0 to be compiled as if the 0 were 1.

Finally, a key specification may contain a '*' which requires the operand to have an indirection as its leading operator. Examples below should clarify the utility of this specification.

Now let us consider the contents of the code string associated with each subtable entry. Conventionally, lower-case letters in this string represent literal information which is copied directly to the output. Upper-case letters generally introduce specific macro-operations, some of which may be followed by modifying information. The code strings in the tables are written with tabs and new-lines used freely to suggest instructions which will be generated; the table-compiling program compresses tabs (using the 0200 bit of the next character) and throws away some of the new-lines. For example the macro 'F' is ordinarily written on a line by itself; but since its expansion will end with a new-line, the new-line after 'F' itself is dispensable. This is all to reduce the size of the stored tables.

The first set of macro-operations is concerned with compiling subtrees. Recall that this is done by the *cexpr* routine. In the following discussion the 'current register' is generally the argument register to *cexpr;* that is, the place where the result is desired. The 'next register' is numbered one higher than the current register. (This explanation isn't fully true because of complications, described below, involving operations which require even-odd register pairs.)

F causes a recursive call to the *rcexpr* routine to compile code which places the value of the first (left) operand of the operator in the current register.

F1 generates code which places the value of the first operand in the next register. It is incorrectly used if there might be no next register; that is, if the degree of difficulty of the first operand is not 'easy;' if not, another register might not be available.

FS generates code which pushes the value of the first operand on the stack, by calling *rcexpr* specifying *sptab* as the table.

Analogously,

S, S1, SScompile the second (right) operand into the current register, the next register, or onto the stack.

To deal with registers, there are

R which expands into the name of the current register.

R1 which expands into the name of the next register.

R+ which expands into the the name of the current register plus 1. It was suggested above that this is the same as the next register, except for complications; here is one of them. Long integer variables have 32 bits and require 2 registers; in such cases the next register is the current register plus 2. The code would like to talk about both halves of the long quantity, so R refers to the register with the high-order part and R+ to the low-order part.

R– This is another complication, involving division and mod. These operations involve a pair of registers of which the odd-numbered contains the left operand. *Cexpr* arranges that the current register is odd; the R– notation allows the code to refer to the next lower, even-numbered register.

To refer to addressible quantities, there are the notations:

A1 causes generation of the address specified by the first operand. For this to be legal, the operand must be addressible; its key must contain an 'a' or a more restrictive specification.

A2 correspondingly generates the address of the second operand providing it has one.

We now have enough mechanism to show a complete, if suboptimal, table for the + operator on word or byte operands.

```
%n,z
        F

%n,1
        F
        inc     R

%n,aw
        F
        add     A2,R

%n,e
        F
        S1
        add     R1,R

%n,n
        SS
        F
        add     (sp)+,R
```

The first two sequences handle some special cases. Actually it turns out that handling a right operand of 0 is unnecessary since the expression-optimizer throws out adds of 0. Adding 1 by using the 'increment' instruction is done next, and then the case where the right operand is addressible. It must be a word quantity, since the PDP-11 lacks an 'add byte' instruction. Finally the cases where the right operand either can, or cannot, be done in the available registers are treated.

The next macro-instructions are conveniently introduced by noticing that the above table is suitable for subtraction as well as addition, since no use is made of the commutativity of addition. All that is needed is substitution of 'sub' for 'add' and 'dec' for 'inc.' Considerable saving of space is achieved by factoring out several similar operations.

I is replaced by a string from another table indexed by the operator in the node being expanded. This secondary table actually contains two strings per operator.

I′ is replaced by the second string in the side table entry for the current operator.

Thus, given that the entries for '+' and '–' in the side table (which is called *instab)* are 'add' and 'inc,' 'sub' and 'dec' respectively, the middle of of the above addition table can be written

```
%n,1
        F
        I'      R

%n,aw
        F
        I       A2,R
```

and it will be suitable for subtraction, and several other operators, as well.

Next, there is the question of character and floating-point operations.

B1 generates the letter ‘b’ if the first operand is a character, ‘f’ if it is float or double, and nothing otherwise. It is used in a context like ‘movB1’ which generates a ‘mov’, ‘movb’, or ‘movf’ instruction according to the type of the operand.

B2 is just like B1 but applies to the second operand.

BE generates ‘b’ if either operand is a character and null otherwise.

BF generates ‘f’ if the type of the operator node itself is float or double, otherwise null.

For example, there is an entry in *efftab* for the ‘=’ operator

```
%a,aw
%ab,a
        IBE     A2,A1
```

Note first that two key specifications can be applied to the same code string. Next, observe that when a word is assigned to a byte or to a word, or a word is assigned to a byte, a single instruction, a *mov* or *movb* as appropriate, does the job. However, when a byte is assigned to a word, it must pass through a register to implement the sign-extension rules:

```
%a,n
        S
        IB1     R,A1
```

Next, there is the question of handling indirection properly. Consider the expression ‘X + *Y’, where X and Y are expressions, Assuming that Y is more complicated than just a variable, but on the other hand qualifies as ‘easy’ in the context, the expression would be compiled by placing the value of X in a register, that of *Y in the next register, and adding the registers. It is easy to see that a better job can be done by compiling X, then Y (into the next register), and producing the instruction symbolized by ‘add (R1),R’. This scheme avoids generating the instruction ‘mov (R1),R1’ required actually to place the value of *Y in a register. A related situation occurs with the expression ‘X + *(p+6)’, which exemplifies a construction frequent in structure and array references. The addition table shown above would produce

```
[put X in register R]
mov     p,R1
add     $6,R1
mov     (R1),R1
add     R1,R
```

when the best code is

```
[put X in R]
mov     p,R1
add     6(R1),R
```

As we said above, a key specification for a code table entry may require an operand to have an indirection as its highest operator. To make use of the requirement, the following macros are provided.

F* the first operand must have the form *X. If in particular it has the form *(Y + c), for some constant *c,* then code is produced which places the value of Y in the current register. Otherwise, code is produced which loads X into the current register.

F1* resembles F* except that the next register is loaded.

S* resembles F* except that the second operand is loaded.

S1* resembles S* except that the next register is loaded.

FS* The first operand must have the form '*X'. Push the value of X on the stack.

SS* resembles FS* except that it applies to the second operand.

To capture the constant that may have been skipped over in the above macros, there are

#1 The first operand must have the form *X; if in particular it has the form *(Y + c) for *c* a constant, then the constant is written out, otherwise a null string.

#2 is the same as #1 except that the second operand is used.

Now we can improve the addition table above. Just before the '%n,e' entry, put

```
%n,ew*
        F
        S1*
        add     #2(R1),R
```

and just before the '%n,n' put

```
%n,nw*
        SS*
        F
        add     *(sp)+,R
```

When using the stacking macros there is no place to use the constant as an index word, so that particular special case doesn't occur.

The constant mentioned above can actually be more general than a number. Any quantity acceptable to the assembler as an expression will do, in particular the address of a static cell, perhaps with a numeric offset. If *x* is an external character array, the expression 'x[i+5] = 0' will generate the code

```
mov     i,r0
clrb    x+5(r0)
```

via the table entry (in the '=' part of *efftab)*

```
%e*,z
        F
        I'B1    #1(R)
```

Some machine operations place restrictions on the registers used. The divide instruction, used to implement the divide and mod operations, requires the dividend to be placed in the odd member of an even-odd pair; other peculiarities of multiplication make it simplest to put the multiplicand in an odd-numbered register. There is no theory which optimally accounts for this kind of requirement. *Cexpr* handles it by checking for a multiply, divide, or mod operation; in these cases, its argument register number is incremented by one or two so that it is odd, and if the operation was divide or mod, so that it is a member of a free even-odd pair. The routine which determines the number of registers required estimates, conservatively, that at least two registers are required for a multiplication and three for the other peculiar operators. After the expression is compiled, the register where the result actually ended up is returned. (Divide and mod are actually the same operation except for the location of the result).

These operations are the ones which cause results to end up in unexpected places, and this possibility adds a further level of complexity. The simplest way of handling the problem is always to move the result to the place where the caller expected it, but this will produce unnecessary register moves in many simple cases; 'a = b*c' would generate

```
mov     b,r1
mul     c,r1
mov     r1,r0
mov     r0,a
```

The next thought is used the passed-back information as to where the result landed to change the notion of the current register. While compiling the '=' operation above, which comes from a table entry like

```
%a,e
        S
        mov     R,A1
```

it is sufficient to redefine the meaning of 'R' after processing the 'S' which does the multiply. This technique is in fact used; the tables are written in such a way that correct code is produced. The trouble is that the technique cannot be used in general, because it invalidates the count of the number of registers required for an expression. Consider just 'a*b + X' where X is some expression. The algorithm assumes that the value of a*b, once computed, requires just one register. If there are three registers available, and X requires two registers to compute, then this expression will match a key specifying '%n,e'. If a*b is computed and left in register 1, then there are, contrary to expectations, no longer two registers available to compute X, but only one, and bad code will be produced. To guard against this possibility, *cexpr* checks the result returned by recursive calls which implement F, S and their relatives. If the result is not in the expected register, then the number of registers required by the other operand is checked; if it can be done using those registers which remain even after making unavailable the unexpectedly-occupied register, then the notions of the 'next register' and possibly the 'current register' are redefined. Otherwise a register-copy instruction is produced. A register-copy is also always produced when the current operator is one of those which have odd-even requirements.

Finally, there are a few loose-end macro operations and facts about the tables. The operators:

V is used for long operations. It is written with an address like a machine instruction; it expands into 'adc' (add carry) if the operation is an additive operator, 'sbc' (subtract carry) if the operation is a subtractive operator, and disappears, along with the rest of the line, otherwise. Its purpose is to allow common treatment of logical operations, which have no carries, and additive and subtractive operations, which generate carries.

T generates a 'tst' instruction if the first operand of the tree does not set the condition codes correctly. It is used with divide and mod operations, which require a sign-extended 32-bit operand. The code table for the operations contains an 'sxt' (sign-extend) instruction to generate the high-order part of the dividend.

H is analogous to the 'F' and 'S' macros, except that it calls for the generation of code for the current tree (not one of its operands) using *regtab.* It is used in *cctab* for all the operators which, when executed normally, set the condition codes properly according to the result. It prevents a 'tst' instruction from being generated for constructions like 'if (a+b) ...' since after calculation of the value of 'a+b' a conditional branch can be written immediately.

All of the discussion above is in terms of operators with operands. Leaves of the expression tree (variables and constants), however, are peculiar in that they have no operands. In order to regularize the matching process, *cexpr* examines its operand to determine if it is a leaf; if so, it creates a special 'load' operator whose operand is the leaf, and substitutes it for the argument tree; this allows the table entry for the created operator to use the 'A1' notation to load the leaf into a register.

Purely to save space in the tables, pieces of subtables can be labelled and referred to later. It turns out, for example, that rather large portions of the the *efftab* table for the '=' and '=+' operators are identical. Thus '=' has an entry

```
%[move3:]
%a,aw
%ab,a
        IBE     A2,A1
```

while part of the '=+' table is

```
%aw,aw
%       [move3]
```

Labels are written as '%[ ... : ]', before the key specifications; references are written with '% [ ... ]' after the key. Peculiarities in the implementation make it necessary that labels appear before references to them.

The example illustrates the utility of allowing separate keys to point to the same code string. The assignment code works properly if either the right operand is a word, or the left operand is a byte; but since there is no 'add byte' instruction the addition code has to be restricted to word operands.

**Delaying and reordering**

Intertwined with the code generation routines are two other, interrelated processes. The first, implemented by a routine called *delay,* is based on the observation that naive code generation for the expression 'a = b++' would produce

```
mov     b,r0
inc     b
mov     r0,a
```

The point is that the table for postfix ++ has to preserve the value of *b* before incrementing it; the general way to do this is to preserve its value in a register. A cleverer scheme would generate

```
mov     b,a
inc     b
```

*Delay* is called for each expression input to *rcexpr,* and it searches for postfix ++ and −− operators. If one is found applied to a variable, the tree is patched to bypass the operator and compiled as it stands; then the increment or decrement itself is done. The effect is as if 'a = b; b++' had been written. In this example, of course, the user himself could have done the same job, but more complicated examples are easily constructed, for example 'switch (x++)'. An essential restriction is that the condition codes not be required. It would be incorrect to compile 'if (a++) ...' as

```
tst     a
inc     a
beq     ...
```

because the 'inc' destroys the required setting of the condition codes.

Reordering is a similar sort of optimization. Many cases which it detects are useful mainly with register variables. If *r* is a register variable, the expression 'r = x+y' is best compiled as

```
mov     x,r
add     y,r
```

but the codes tables would produce

```
mov     x,r0
add     y,r0
mov     r0,r
```

which is in fact preferred if *r* is not a register. (If *r* is not a register, the two sequences are the same size, but the second is slightly faster.) The scheme is to compile the expression as if it had been written 'r = x; r =+ y'. The *reorder* routine is called with a pointer to each tree that *rcexpr* is about to compile; if it has the right characteristics, the 'r = x' tree is constructed and passed recursively to *rcexpr;* then the original tree is modified to read ‘r =+ y’ and the calling instance of *rcexpr* compiles that instead. Of course the whole business is itself recursive so that more extended forms of the same phenomenon are handled, like ‘r = x + y | z’.

Care does have to be taken to avoid ‘optimizing’ an expression like ‘r = x + r’ into ‘r = x; r =+ r’. It is required that the right operand of the expression on the right of the ‘=’ be a ’, distinct from the register variable.

The second case that *reorder* handles is expressions of the form ‘r = X’ used as a subexpression. Again, the code out of the tables for ‘x = r = y’ would be

```
mov     y,r0
mov     r0,r
mov     r0,x
```

whereas if *r* were a register it would be better to produce

```
mov     y,r
mov     r,x
```

When *reorder* discovers that a register variable is being assigned to in a subexpression, it calls *rcexpr* recursively to compile the subexpression, then fiddles the tree passed to it so that the register variable itself appears as the operand instead of the whole subexpression. Here care has to be taken to avoid an infinite regress, with *rcexpr* and *reorder* calling each other forever to handle assignments to registers.

A third set of cases treated by *reorder* comes up when any name, not necessarily a register, occurs as a left operand of an assignment operator other than ‘=’ or as an operand of prefix ‘++’ or ‘−−’. Unless condition-code tests are involved, when a subexpression like ‘(a =+ b)’ is seen, the assignment is performed and the argument tree modified so that *a* is its operand; effectively ‘x + (y =+ z)’ is compiled as ‘y =+ z; x + y’. Similarly, prefix increment and decrement are pulled out and performed first, then the remainder of the expression.

Throughout code generation, the expression optimizer is called whenever *delay* or *reorder* change the expression tree. This allows some special cases to be found that otherwise would not be seen.

# A Tour Through the Portable C Compiler

*S. C. Johnson*

Bell Laboratories
Murray Hill, New Jersey 07974

## Introduction

A C compiler has been implemented that has proved to be quite portable, serving as the basis for C compilers on roughly a dozen machines, including the Honeywell 6000, IBM 370, and Interdata 8/32. The compiler is highly compatible with the C language standard.[1]

Among the goals of this compiler are portability, high reliability, and the use of state-of-the-art techniques and tools wherever practical. Although the efficiency of the compiling process is not a primary goal, the compiler is efficient enough, and produces good enough code, to serve as a production compiler.

The language implemented is highly compatible with the current PDP-11 version of C. Moreover, roughly 75% of the compiler, including nearly all the syntactic and semantic routines, is machine independent. The compiler also serves as the major portion of the program *lint*, described elsewhere.[2]

A number of earlier attempts to make portable compilers are worth noting. While on CO-OP assignment to Bell Labs in 1973, Alan Snyder wrote a portable C compiler which was the basis of his Master's Thesis at M.I.T.[3] This compiler was very slow and complicated, and contained a number of rather serious implementation difficulties; nevertheless, a number of Snyder's ideas appear in this work.

Most earlier portable compilers, including Snyder's, have proceeded by defining an intermediate language, perhaps based on three-address code or code for a stack machine, and writing a machine independent program to translate from the source code to this intermediate code. The intermediate code is then read by a second pass, and interpreted or compiled. This approach is elegant, and has a number of advantages, especially if the target machine is far removed from the host. It suffers from some disadvantages as well. Some constructions, like initialization and subroutine prologs, are difficult or expensive to express in a machine independent way that still allows them to be easily adapted to the target assemblers. Most of these approaches require a symbol table to be constructed in the second (machine dependent) pass, and/or require powerful target assemblers. Also, many conversion operators may be generated that have no effect on a given machine, but may be needed on others (for example, pointer to pointer conversions usually do nothing in C, but must be generated because there are some machines where they are significant).

For these reasons, the first pass of the portable compiler is not entirely machine independent. It contains some machine dependent features, such as initialization, subroutine prolog and epilog, certain storage allocation functions, code for the *switch* statement, and code to throw out unneeded conversion operators.

As a crude measure of the degree of portability actually achieved, the Interdata 8/32 C compiler has roughly 600 machine dependent lines of source out of 4600 in Pass 1, and 1000 out of 3400 in Pass 2. In total, 1600 out of 8000, or 20%, of the total source is machine dependent (12% in Pass 1, 30% in Pass 2). These percentages can be expected to rise slightly as the compiler is tuned. The percentage of machine-dependent code for the IBM is 22%, for the Honeywell 25%. If the assembler format and structure were the same for all these machines, perhaps another 5-10% of the code would become machine independent.

These figures are sufficiently misleading as to be almost meaningless. A large fraction of the machine dependent code can be converted in a straightforward, almost mechanical way. On the other hand, a certain amount of the code requres hard intellectual effort to convert, since the algorithms embodied in this part of the code are typically complicated and machine dependent.

To summarize, however, if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished!

**Overview**

This paper discusses the structure and organization of the portable compiler. The intent is to give the big picture, rather than discussing the details of a particular machine implementation. After a brief overview and a discussion of the source file structure, the paper describes the major data structures, and then delves more closely into the two passes. Some of the theoretical work on which the compiler is based, and its application to the compiler, is discussed elsewhere.[4] One of the major design issues in any C compiler, the design of the calling sequence and stack frame, is the subject of a separate memorandum.[5]

The compiler consists of two passes, *pass1* and *pass2*, that together turn C source code into assembler code for the target machine. The two passes are preceded by a preprocessor, that handles the **#define** and **#include** statements, and related features (e.g., **#ifdef**, etc.). It is a nearly machine independent program, and will not be further discussed here.

The output of the preprocessor is a text file that is read as the standard input of the first pass. This produces as standard output another text file that becomes the standard input of the second pass. The second pass produces, as standard output, the desired assembler language source code. The preprocessor and the two passes all write error messages on the standard error file. Thus the compiler itself makes few demands on the I/O library support, aiding in the bootstrapping process.

Although the compiler is divided into two passes, this represents historical accident more than deep necessity. In fact, the compiler can optionally be loaded so that both passes operate in the same program. This ''one pass'' operation eliminates the overhead of reading and writing the intermediate file, so the compiler operates about 30% faster in this mode. It also occupies about 30% more space than the larger of the two component passes.

Because the compiler is fundamentally structured as two passes, even when loaded as one, this document primarily describes the two pass version.

The first pass does the lexical analysis, parsing, and symbol table maintenance. It also constructs parse trees for expressions, and keeps track of the types of the nodes in these trees. Additional code is devoted to initialization. Machine dependent portions of the first pass serve to generate subroutine prologs and epilogs, code for switches, and code for branches, label definitions, alignment operations, changes of location counter, etc.

The intermediate file is a text file organized into lines. Lines beginning with a right parenthesis are copied by the second pass directly to its output file, with the parenthesis stripped off. Thus, when the first pass produces assembly code, such as subroutine prologs, etc., each line is prefaced with a right parenthesis; the second pass passes these lines to through to the assembler.

The major job done by the second pass is generation of code for expressions. The expression parse trees produced in the first pass are written onto the intermediate file in Polish Prefix form: first, there is a line beginning with a period, followed by the source file line number and name on which the expression appeared (for debugging purposes). The successive lines represent the nodes of the parse tree, one node per line. Each line contains the node number, type, and any values (e.g., values of constants) that may appear in the node. Lines representing nodes with descendants are immediately followed by the left subtree of descendants, then the right. Since the number of descendants of any node is completely determined by the node number, there is no need to mark the end of the tree.

There are only two other line types in the intermediate file. Lines beginning with a left square bracket ('[') represent the beginning of blocks (delimited by { ... } in the C source); lines beginning with right square brackets (']') represent the end of blocks. The remainder of these lines tell how much stack space, and how many register variables, are currently in use.

Thus, the second pass reads the intermediate files, copies the ')' lines, makes note of the information in the '[' and ']' lines, and devotes most of its effort to the '.' lines and their associated expression trees, turning them turns into assembly code to evaluate the expressions.

In the one pass version of the compiler, the expression trees that are built by the first pass have been declared to have room for the second pass information as well. Instead of writing the trees onto an intermediate file, each tree is transformed in place into an acceptable form for the code generator. The code generator then writes the result of compiling this tree onto the standard output. Instead of '[' and ']' lines in the intermediate file, the information is passed directly to the second pass routines. Assembly code produced by the first pass is simply written out, without the need for ')' at the head of each line.

**The Source Files**

The compiler source consists of 22 source files. Two files, *manifest* and *macdefs*, are header files included with all other files. *Manifest* has declarations for the node numbers, types, storage classes, and other global data definitions. *Macdefs* has machine-dependent definitions, such as the size and alignment of the various data representations. Two machine independent header files, *mfile1* and *mfile2*, contain the data structure and manifest definitions for the first and second passes, respectively. In the second pass, a machine dependent header file, *mac2defs*, contains declarations of register names, etc.

There is a file, *common*, containing (machine independent) routines used in both passes. These include routines for allocating and freeing trees, walking over trees, printing debugging information, and printing error messages. There are two dummy files, *comm1.c* and *comm2.c*, that simply include *common* within the scope of the appropriate pass1 or pass2 header files. When the compiler is loaded as a single pass, *common* only needs to be included once: *comm2.c* is not needed.

Entire sections of this document are devoted to the detailed structure of the passes. For the moment, we just give a brief description of the files. The first pass is obtained by compiling and loading *scan.c*, *cgram.c*, *xdefs.c*, *pftn.c*, *trees.c*, *optim.c*, *local.c*, *code.c*, and *comm1.c*. *Scan.c* is the lexical analyzer, which is used by *cgram.c*, the result of applying *Yacc*[6] to the input grammar *cgram.y*. *Xdefs.c* is a short file of external definitions. *Pftn.c* maintains the symbol table, and does initialization. *Trees.c* builds the expression trees, and computes the node types. *Optim.c* does some machine independent optimizations on the expression trees. *Comm1.c* includes *common*, that contains service routines common to the two passes of the compiler. All the above files are machine independent. The files *local.c* and *code.c* contain machine dependent code for generating subroutine prologs, switch code, and the like.

The second pass is produced by compiling and loading *reader.c*, *allo.c*, *match.c*, *comm1.c*, *order.c*, *local.c*, and *table.c*. *Reader.c* reads the intermediate file, and controls the major logic of the code generation. *Allo.c* keeps track of busy and free registers. *Match.c* controls the matching of code templates to subtrees of the expression tree to be compiled. *Comm2.c* includes the file *common*, as in the first pass. The above files are machine independent. *Order.c* controls the machine dependent details of the code generation strategy. *Local2.c* has many small machine dependent routines, and tables of opcodes, register types, etc. *Table.c* has the code template tables, which are also clearly machine dependent.

**Data Structure Considerations.**

This section discusses the node numbers, type words, and expression trees, used throughout both passes of the compiler.

The file *manifest* defines those symbols used throughout both passes. The intent is to use the same symbol name (e.g., MINUS) for the given operator throughout the lexical analysis, parsing, tree building, and code generation phases; this requires some synchronization with the *Yacc* input file, *cgram.y*, as well.

A token like MINUS may be seen in the lexical analyzer before it is known whether it is a unary or binary operator; clearly, it is necessary to know this by the time the parse tree is constructed. Thus, an operator (really a macro) called UNARY is provided, so that MINUS and UNARY MINUS are both distinct node numbers. Similarly, many binary operators exist in an assignment form (for example, −=), and the operator ASG may be applied to such node names to generate new ones, e.g. ASG MINUS.

It is frequently desirable to know if a node represents a leaf (no descendants), a unary operator (one descendant) or a binary operator (two descendants). The macro *optype(o)* returns one of the manifest constants LTYPE, UTYPE, or BITYPE, respectively, depending on the node number *o*. Similarly, *asgop(o)* returns true if *o* is an assignment operator number (=, +=, etc. ), and *logop(o)* returns true if *o* is a relational or logical (&&, ||, or !) operator.

C has a rich typing structure, with a potentially infinite number of types. To begin with, there are the basic types: CHAR, SHORT, INT, LONG, the unsigned versions known as UCHAR, USHORT, UNSIGNED, ULONG, and FLOAT, DOUBLE, and finally STRTY (a structure), UNIONTY, and ENUMTY. Then, there are three operators that can be applied to types to make others: if *t* is a type, we may potentially have types *pointer to t*, *function returning t*, and *array of t's* generated from *t*. Thus, an arbitrary type in C consists of a basic type, and zero or more of these operators.

In the compiler, a type is represented by an unsigned integer; the rightmost four bits hold the basic type, and the remaining bits are divided into two-bit fields, containing 0 (no operator), or one of the three operators described above. The modifiers are read right to left in the word, starting with the two-bit field adjacent to the basic type, until a field with 0 in it is reached. The macros PTR, FTN, and ARY represent the *pointer to*, *function returning*, and *array of* operators. The macro values are shifted so that they align with the first two-bit field; thus PTR+INT represents the type for an integer pointer, and

```
ARY + (PTR<<2) + (FTN<<4) + DOUBLE
```

represents the type of an array of pointers to functions returning doubles.

The type words are ordinarily manipulated by macros. If *t* is a type word, *BTYPE(t)* gives the basic type. *ISPTR(t)*, *ISARY(t)*, and *ISFTN(t)* ask if an object of this type is a pointer, array, or a function, respectively. *MODTYPE(t,b)* sets the basic type of *t* to *b*. *DECREF(t)* gives the type resulting from removing the first operator from *t*. Thus, if *t* is a pointer to *t'*, a function returning *t'*, or an array of *t'*, then *DECREF(t)* would equal *t'*. *INCREF(t)* gives the type representing a pointer to *t*. Finally, there are operators for dealing with the unsigned types. *ISUNSIGNED(t)* returns true if *t* is one of the four basic unsigned types; in this case, *DEUNSIGN(t)* gives the associated 'signed' type. Similarly, *UNSIGNABLE(t)* returns true if *t* is one of the four basic types that could become unsigned, and *ENUNSIGN(t)* returns the unsigned analogue of *t* in this case.

The other important global data structure is that of expression trees. The actual shapes of the nodes are given in *mfile1* and *mfile2*. They are not the same in the two passes; the first pass nodes contain dimension and size information, while the second pass nodes contain register allocation information. Nevertheless, all nodes contain fields called *op*, containing the node number, and *type*, containing the type word. A function called *talloc()* returns a pointer to a new tree node. To free a node, its *op* field need merely be set to FREE. The other fields in the node will remain intact at least until the next allocation.

Nodes representing binary operators contain fields, *left* and *right*, that contain pointers to the left and right descendants. Unary operator nodes have the *left* field, and a value field called *rval*. Leaf nodes, with no descendants, have two value fields: *lval* and *rval*.

At appropriate times, the function *tcheck()* can be called, to check that there are no busy nodes remaining. This is used as a compiler consistency check. The function *tcopy(p)* takes a pointer *p* that points to an expression tree, and returns a pointer to a disjoint copy of the tree. The function *walkf(p,f)* performs a postorder walk of the tree pointed to by *p*, and applies the function *f* to each node. The function *fwalk(p,f,d)* does a preorder walk of the tree pointed to by *p*. At each node, it calls a function *f*, passing to it the node pointer, a value passed down from its ancestor, and two pointers to values to be passed down to the left and right descendants (if any). The value *d* is the value passed down to the root. *Fwalk* is used for a number of tree labeling and debugging activities.

The other major data structure, the symbol table, exists only in pass one, and will be discussed later.

**Pass One**

The first pass does lexical analysis, parsing, symbol table maintenance, tree building, optimization, and a number of machine dependent things. This pass is largely machine independent, and the machine independent sections can be pretty successfully ignored. Thus, they will be only sketched here.

**Lexical Analysis**

The lexical analyzer is a conceptually simple routine that reads the input and returns the tokens of the C language as it encounters them: names, constants, operators, and keywords. The conceptual simplicity of this job is confounded a bit by several other simple jobs that unfortunately must go on simultaneously. These include

- Keeping track of the current filename and line number, and occasionally setting this information as the result of preprocessor control lines.
- Skipping comments.
- Properly dealing with octal, decimal, hex, floating point, and character constants, as well as character strings.

To achieve speed, the program maintains several tables that are indexed into by character value, to tell the lexical analyzer what to do next. To achieve portability, these tables must be initialized each time the compiler is run, in order that the table entries reflect the local character set values.

**Parsing**

As mentioned above, the parser is generated by Yacc from the grammar on file *cgram.y*. The grammar is relatively readable, but contains some unusual features that are worth comment.

Perhaps the strangest feature of the grammar is the treatment of declarations. The problem is to keep track of the basic type and the storage class while interpreting the various stars, brackets, and parentheses that may surround a given name. The entire declaration mechanism must be recursive, since declarations may appear within declarations of structures and unions, or even within a **sizeof** construction inside a dimension in another declaration!

There are some difficulties in using a bottom-up parser, such as produced by Yacc, to handle constructions where a lot of left context information must be kept around. The problem is that the original PDP-11 compiler is top-down in implementation, and some of the semantics of C reflect this. In a top-down parser, the input rules are restricted somewhat, but one can naturally associate temporary storage with a rule at a very early stage in the recognition of that rule. In a bottom-up parser, there is more freedom in the specification of rules, but it is more difficult to know what rule is being matched until the entire rule is seen. The parser described by *cgram.c* makes effective use of the bottom-up parsing mechanism in some places (notably the treatment of expressions), but struggles against the restrictions in others. The usual result is that it is necessary to run a stack of values ‘‘on the side’’, independent of the Yacc value stack, in order to be able to store and access information deep within inner constructions, where the relationship of the rules being recognized to the total picture is not yet clear.

In the case of declarations, the attribute information (type, etc.) for a declaration is carefully kept immediately to the left of the declarator (that part of the declaration involving the name). In this way, when it is time to declare the name, the name and the type information can be quickly brought together. The ‘‘$0’’ mechanism of Yacc is used to accomplish this. The result is not pretty, but it works. The storage class information changes more slowly, so it is kept in an external variable, and stacked if necessary. Some of the grammar could be considerably cleaned up by using some more recent features of Yacc, notably actions within rules and the ability to return multiple values for actions.

A stack is also used to keep track of the current location to be branched to when a **break** or **continue** statement is processed.

This use of external stacks dates from the time when Yacc did not permit values to be structures. Some, or most, of this use of external stacks could be eliminated by redoing the grammar to use the mechanisms now provided. There are some areas, however, particularly the processing of structure, union, and enum declarations, function prologs, and switch statement processing, when having all the affected data together in an array speeds later processing; in this case, use of external storage seems essential.

The *cgram.y* file also contains some small functions used as utility functions in the parser. These include routines for saving case values and labels in processing switches, and stacking and popping values on the external stack described above.

**Storage Classes**

C has a finite, but fairly extensive, number of storage classes available. One of the compiler design decisions was to process the storage class information totally in the first pass; by the second pass, this information must have been totally dealt with. This means that all of the storage allocation must take place in the first pass, so that references to automatics and parameters can be turned into references to cells lying a certain number of bytes offset from certain machine registers. Much of this transformation is machine dependent, and strongly depends on the storage class.

The classes include EXTERN (for externally declared, but not defined variables), EXTDEF (for external definitions), and similar distinctions for USTATIC and STATIC, UFORTRAN and FORTRAN (for fortran functions) and ULABEL and LABEL. The storage classes REGISTER and AUTO are obvious, as are STNAME, UNAME, and ENAME (for structure, union, and enumeration tags), and the associated MOS, MOU, and MOE (for the members). TYPEDEF is treated as a storage class as well. There are two special storage classes: PARAM and SNULL. SNULL is used to distinguish the case where no explicit storage class has been given; before an entry is made in the symbol table the true storage class is discovered. Similarly, PARAM is used for the temporary entry in the symbol table made before the declaration of function parameters is completed.

The most complexity in the storage class process comes from bit fields. A separate storage class is kept for each width bit field; a $k$ bit bit field has storage class $k$ plus FIELD. This enables the size to be quickly recovered from the storage class.

**Symbol Table Maintenance.**

The symbol table routines do far more than simply enter names into the symbol table; considerable semantic processing and checking is done as well. For example, if a new declaration comes in, it must be checked to see if there is a previous declaration of the same symbol. If there is, there are many cases. The declarations may agree and be compatible (for example, an extern declaration can appear twice) in which case the new declaration is ignored. The new declaration may add information (such as an explicit array dimension) to an already present declaration. The new declaration may be different, but still correct (for example, an extern declaration of something may be entered, and then later the definition may be seen). The new declaration may be incompatible, but appear in an inner block; in this case, the old declaration is carefully hidden away, and the new one comes into force until the block is left. Finally, the declarations may be incompatible, and an error message must be produced.

A number of other factors make for additional complexity. The type declared by the user is not always the type entered into the symbol table (for example, if an formal parameter to a function is declared to be an array, C requires that this be changed into a pointer before entry in the symbol table). Moreover, there are various kinds of illegal types that may be declared which are difficult to check for syntactically (for example, a function returning an array). Finally, there is a strange feature in C that requires structure tag names and member names for structures and unions to be taken from a different logical symbol table than ordinary identifiers. Keeping track of which kind of name is involved is a bit of struggle (consider typedef names used within structure declarations, for example).

The symbol table handling routines have been rewritten a number of times to extend features, improve performance, and fix bugs. They address the above problems with reasonable effectiveness but a singular lack of grace.

When a name is read in the input, it is hashed, and the routine *lookup* is called, together with a flag which tells which symbol table should be searched (actually, both symbol tables are stored in one, and a flag is used to distinguish individual entries). If the name is found, *lookup* returns the index to the entry found; otherwise, it makes a new entry, marks it UNDEF (undefined), and returns the index of the new entry. This index is stored in the *rval* field of a NAME node.

When a declaration is being parsed, this NAME node is made part of a tree with UNARY MUL nodes for each *, LB nodes for each array descriptor (the right descendant has the dimension), and UNARY CALL nodes for each function descriptor. This tree is passed to the routine *tymerge*, along with the attribute type of the whole declaration; this routine collapses the tree to a single node, by calling *tyreduce*, and then modifies the type to reflect the overall type of the declaration.

Dimension and size information is stored in a table called *dimtab*. To properly describe a type in C, one needs not just the type information but also size information (for structures and enums) and dimension information (for arrays). Sizes and offsets are dealt with in the compiler by giving the associated indices into *dimtab*. *Tymerge* and *tyreduce* call *dstash* to put the discovered dimensions away into the *dimtab* array. *Tymerge* returns a pointer to a single node that contains the symbol table index in its *rval* field, and the size and dimension indices in fields *csiz* and *cdim*, respectively. This information is properly considered part of the type in the first pass, and is carried around at all times.

To enter an element into the symbol table, the routine *defid* is called; it is handed a storage class, and a pointer to the node produced by *tymerge*. *Defid* calls *fixtype*, which adjusts and checks the given type depending on the storage class, and converts null types appropriately. It then calls *fixclass*, which does a similar job for the storage class; it is here, for example, that register declarations are either allowed or changed to auto.

The new declaration is now compared against an older one, if present, and several pages of validity checks performed. If the definitions are compatible, with possibly some added information, the processing is straightforward. If the definitions differ, the block levels of the current and the old declaration are compared. The current block level is kept in *blevel*, an external variable; the old declaration level is kept in the symbol table. Block level 0 is for external declarations, 1 is for arguments to functions, and 2 and above are blocks within a function. If the current block level is the same as the old declaration, an error results. If the current block level is higher, the new declaration overrides the old. This is done by marking the old symbol table entry ‘‘hidden’’, and making a new entry, marked ‘‘hiding’’. *Lookup* will skip over hidden entries. When a block is left, the symbol table is searched, and any entries defined in that block are destroyed; if they hid other entries, the old entries are ‘‘unhidden’’.

This nice block structure is warped a bit because labels do not follow the block structure rules (one can do a **goto** into a block, for example); default definitions of functions in inner blocks also persist clear out to the outermost scope. This implies that cleaning up the symbol table after block exit is more subtle than it might first seem.

For successful new definitions, *defid* also initializes a ‘‘general purpose’’ field, *offset*, in the symbol table. It contains the stack offset for automatics and parameters, the register number for register variables, the bit offset into the structure for structure members, and the internal label number for static variables and labels. The offset field is set by *falloc* for bit fields, and *dclstruct* for structures and unions.

The symbol table entry itself thus contains the name, type word, size and dimension offsets, offset value, and declaration block level. It also has a field of flags, describing what symbol table the name is in, and whether the entry is hidden, or hides another. Finally, a field gives the line number of the last use, or of the definition, of the name. This is used mainly for diagnostics, but is useful to *lint* as well.

In some special cases, there is more than the above amount of information kept for the use of the compiler. This is especially true with structures; for use in initialization, structure declarations must have access to a list of the members of the structure. This list is also kept in *dimtab*. Because a structure can be mentioned long before the members are known, it is necessary to have another level of indirection in the table. The two words following the *csiz* entry in *dimtab* are used to hold the alignment of the structure, and the index in dimtab of the list of members. This list contains the symbol table indices for the structure members, terminated by a $-1$.

**Tree Building**

The portable compiler transforms expressions into expression trees. As the parser recognizes each rule making up an expression, it calls *buildtree* which is given an operator number, and pointers to the left and right descendants. *Buildtree* first examines the left and right descendants, and, if they are both constants, and the operator is appropriate, simply does the constant computation at compile time, and returns the result as a constant. Otherwise, *buildtree* allocates a node for the head of the tree, attaches the descendants to it, and ensures that conversion operators are generated if needed, and that the type of the new node is consistent with the types of the operands. There is also a considerable amount of semantic complexity here; many combinations of types are illegal, and the portable compiler makes a strong effort to check the legality of expression types completely. This is done both for *lint* purposes, and to prevent such semantic errors from being passed through to the code generator.

The heart of *buildtree* is a large table, accessed by the routine *opact*. This routine maps the types of the left and right operands into a rather smaller set of descriptors, and then accesses a table (actually encoded in a switch statement) which for each operator and pair of types causes an action to be returned. The actions are logical or's of a number of separate actions, which may be carried out by *buildtree*. These component actions may include checking the left side to ensure that it is an lvalue (can be stored into), applying a type conversion to the left or right operand, setting the type of the new node to the type of the left or right operand, calling various routines to balance the types of the left and right operands, and suppressing the ordinary conversion of arrays and function operands to pointers. An important operation is OTHER, which causes some special code to be invoked in *buildtree*, to handle issues which are unique to a particular operator. Examples of this are structure and union reference (actually handled by the routine *stref*), the building of NAME, ICON, STRING and FCON (floating point constant) nodes, unary * and &, structure assignment, and calls. In the case of unary * and &, *buildtree* will cancel a * applied to a tree, the top node of which is &, and conversely.

Another special operation is PUN; this causes the compiler to check for type mismatches, such as intermixing pointers and integers.

The treatment of conversion operators is still a rather strange area of the compiler (and of C!). The recent introduction of type casts has only confounded this situation. Most of the conversion operators are generated by calls to *tymatch* and *ptmatch*, both of which are given a tree, and asked to make the operands agree in type. *Ptmatch* treats the case where one of the operands is a pointer; *tymatch* treats all other cases. Where these routines have decided on the proper type for an operand, they call *makety*, which is handed a tree, and a type word, dimension offset, and size offset. If necessary, it inserts a conversion operation to make the types correct. Conversion operations are never inserted on the left side of assignment operators, however. There are two conversion operators used; PCONV, if the conversion is to a non-basic type (usually a pointer), and SCONV, if the conversion is to a basic type (scalar).

To allow for maximum flexibility, every node produced by *buildtree* is given to a machine dependent routine, *clocal*, immediately after it is produced. This is to allow more or less immediate rewriting of those nodes which must be adapted for the local machine. The conversion operations are given to *clocal* as well; on most machines, many of these conversions do nothing, and should be thrown away (being careful to retain the type). If this operation is done too early, however, later calls to *buildtree* may get confused about correct type of the subtrees; thus *clocal* is given the conversion ops only after the entire tree is built. This topic will be dealt with in more detail later.

**Initialization**

Initialization is one of the messier areas in the portable compiler. The only consolation is that most of the mess takes place in the machine independent part, where it is may be safely ignored by the implementor of the compiler for a particular machine.

The basic problem is that the semantics of initialization really calls for a co-routine structure; one collection of programs reading constants from the input stream, while another, independent set of programs places these constants into the appropriate spots in memory. The dramatic differences in the local assemblers also come to the fore here. The parsing problems are dealt with by keeping a rather extensive stack containing the current state of the initialization; the assembler problems are dealt with by having a fair number of machine dependent routines.

The stack contains the symbol table number, type, dimension index, and size index for the current identifier being initialized. Another entry has the offset, in bits, of the beginning of the current identifier. Another entry keeps track of how many elements have been seen, if the current identifier is an array. Still another entry keeps track of the current member of a structure being initialized. Finally, there is an entry containing flags which keep track of the current state of the initialization process (e.g., tell if a } has been seen for the current identifier.)

When an initialization begins, the routine *beginit* is called; it handles the alignment restrictions, if any, and calls *instk* to create the stack entry. This is done by first making an entry on the top of the stack for the item being initialized. If the top entry is an array, another entry is made on the stack for the first element. If the top entry is a structure, another entry is made on the stack for the first member of the structure. This continues until the top element of the stack is a scalar. *Instk* then returns, and the parser begins collecting initializers.

When a constant is obtained, the routine *doinit* is called; it examines the stack, and does whatever is necessary to assign the current constant to the scalar on the top of the stack. *gotscal* is then called, which rearranges the stack so that the next scalar to be initialized gets placed on top of the stack. This process continues until the end of the initializers; *endinit* cleans up. If a { or } is encountered in the string of initializers, it is handled by calling *ilbrace* or *irbrace*, respectively.

A central issue is the treatment of the ''holes'' that arise as a result of alignment restrictions or explicit requests for holes in bit fields. There is a global variable, *inoff*, which contains the current offset in the initialization (all offsets in the first pass of the compiler are in bits). *Doinit* figures out from the top entry on the stack the expected bit offset of the next identifier; it calls the machine dependent routine *inforce* which, in a machine dependent way, forces the assembler to set aside space if need be so that the next scalar seen will go into the appropriate bit offset position. The scalar itself is passed to one of the machine dependent routines *fincode* (for floating point initialization), *incode* (for fields, and other initializations less than an int in size), and *cinit* (for all other initializations). The size is passed to all these routines, and it is up to the machine dependent routines to ensure that the initializer occupies exactly the right size.

Character strings represent a bit of an exception. If a character string is seen as the initializer for a pointer, the characters making up the string must be put out under a different location counter. When the lexical analyzer sees the quote at the head of a character string, it returns the token STRING, but does not do anything with the contents. The parser calls *getstr*, which sets up the appropriate location counters and flags, and calls *lxstr* to read and process the contents of the string.

If the string is being used to initialize a character array, *lxstr* calls *putbyte*, which in effect simulates *doinit* for each character read. If the string is used to initialize a character pointer, *lxstr* calls a machine dependent routine, *bycode*, which stashes away each character. The pointer to this string is then returned, and processed normally by *doinit*.

The null at the end of the string is treated as if it were read explicitly by *lxstr*.

**Statements**

The first pass addresses four main areas; declarations, expressions, initialization, and statements. The statement processing is relatively simple; most of it is carried out in the parser directly. Most of the logic is concerned with allocating label numbers, defining the labels, and branching appropriately. An external symbol, *reached*, is 1 if a statement can be reached, 0 otherwise; this is used to do a bit of simple flow analysis as the program is being parsed, and also to avoid generating the subroutine return sequence if the subroutine cannot ''fall through'' the last statement.

Conditional branches are handled by generating an expression node, CBRANCH, whose left descendant is the conditional expression and the right descendant is an ICON node containing the internal label number to be branched to. For efficiency, the semantics are that the label is gone to if the condition is *false*.

The switch statement is compiled by collecting the case entries, and an indication as to whether there is a default case; an internal label number is generated for each of these, and remembered in a big array. The expression comprising the value to be switched on is compiled when the switch keyword is encountered, but the expression tree is headed by a special node, FORCE, which tells the code generator to put the expression value into a special distinguished register (this same mechanism is used for processing the return statement). When the end of the switch block is reached, the array containing the case values is sorted, and checked for duplicate entries (an error); if all is correct, the machine dependent routine *genswitch* is called, with this array of labels and values in increasing order. *Genswitch* can assume that the value to be tested is already in the register which is the usual integer return value register.

**Optimization**

There is a machine independent file, *optim.c*, which contains a relatively short optimization routine, *optim*. Actually the word optimization is something of a misnomer; the results are not optimum, only improved, and the routine is in fact not optional; it must be called for proper operation of the compiler.

*Optim* is called after an expression tree is built, but before the code generator is called. The essential part of its job is to call *clocal* on the conversion operators. On most machines, the treatment of & is also essential: by this time in the processing, the only node which is a legal descendant of & is NAME. (Possible descendants of * have been eliminated by *buildtree.)* The address of a static name is, almost by definition, a constant, and can be represented by an ICON node on most machines (provided that the loader has enough power). Unfortunately, this is not universally true; on some machine, such as the IBM 370, the issue of addressability rears its ugly head; thus, before turning a NAME node into an ICON node, the machine dependent function *andable* is called.

The optimization attempts of *optim* are currently quite limited. It is primarily concerned with improving the behavior of the compiler with operations one of whose arguments is a constant. In the simplest case, the constant is placed on the right if the operation is commutative. The compiler also makes a limited search for expressions such as

$$( x + a ) + b$$

where $a$ and $b$ are constants, and attempts to combine $a$ and $b$ at compile time. A number of special cases are also examined; additions of 0 and multiplications by 1 are removed, although the correct processing of these cases to get the type of the resulting tree correct is decidedly nontrivial. In some cases, the addition or multiplication must be replaced by a conversion op to keep the types from becoming fouled up. Finally, in cases where a relational operation is being done, and one operand is a constant, the operands are permuted, and the operator altered, if necessary, to put the constant on the right. Finally, multiplications by a power of 2 are changed to shifts.

There are dozens of similar optimizations that can be, and should be, done. It seems likely that this routine will be expanded in the relatively near future.

**Machine Dependent Stuff**

A number of the first pass machine dependent routines have been discussed above. In general, the routines are short, and easy to adapt from machine to machine. The two exceptions to this general rule are *clocal* and the function prolog and epilog generation routines, *bfcode* and *efcode*.

*Clocal* has the job of rewriting, if appropriate and desirable, the nodes constructed by *buildtree*. There are two major areas where this is important; NAME nodes and conversion operations. In the case of NAME nodes, *clocal* must rewrite the NAME node to reflect the actual physical location of the name in the machine. In effect, the NAME node must be examined, the symbol table entry found (through the *rval* field of the node), and, based on the storage class of the node, the tree must be rewritten. Automatic variables and parameters are typically rewritten by treating the reference to the variable as a structure reference, off the register which holds the stack or argument pointer; the *stref* routine is set up to be called in this way, and to build the appropriate tree. In the most general case, the tree consists of a unary * node, whose descendant is a + node, with the stack or argument register as left operand, and a constant offset as right operand. In the case of LABEL and internal static nodes, the *rval* field is rewritten to be the negative of the internal label number; a negative *rval* field is taken to be an internal label number. Finally, a name of class REGISTER must be converted into a REG node, and the *rval* field replaced by the register number. In fact, this part of the *clocal* routine is nearly machine independent; only for machines with addressability problems (IBM 370 again!) does it have to be noticeably different,

The conversion operator treatment is rather tricky. It is necessary to handle the application of conversion operators to constants in *clocal*, in order that all constant expressions can have their values known at compile time. In extreme cases, this may mean that some simulation of the arithmetic of the target machine might have to be done in a cross-compiler. In the most common case, conversions from pointer to pointer do nothing. For some machines, however, conversion from byte pointer to short or long pointer might require a shift or rotate operation, which would have to be generated here.

The extension of the portable compiler to machines where the size of a pointer depends on its type would be straightforward, but has not yet been done.

The other major machine dependent issue involves the subroutine prolog and epilog generation. The hard part here is the design of the stack frame and calling sequence; this design issue is discussed elsewhere.[5] The routine *bfcode* is called with the number of arguments the function is defined with, and an array containing the symbol table indices of the declared parameters. *Bfcode* must generate the code to establish the new stack frame, save the return address and previous stack pointer value on the stack, and save whatever registers are to be used for register variables. The stack size and the number of register variables is not known when *bfcode* is called, so these numbers must be referred to by assembler constants, which are defined when they are known (usually in the second pass, after all register variables, automatics, and temporaries have been seen). The final job is to find those parameters which may have been declared register, and generate the code to initialize the register with the value passed on the stack. Once again, for most machines, the general logic of *bfcode* remains the same, but the contents of the *printf* calls in it will change from machine to machine. *efcode* is rather simpler, having just to generate the default return at the end of a function. This may be nontrivial in the case of a function returning a structure or union, however.

There seems to be no really good place to discuss structures and unions, but this is as good a place as any. The C language now supports structure assignment, and the passing of structures as arguments to functions, and the receiving of structures back from functions. This was added rather late to C, and thus to the portable compiler. Consequently, it fits in less well than the older features. Moreover, most of the burden of making these features work is placed on the machine dependent code.

There are both conceptual and practical problems. Conceptually, the compiler is structured around the idea that to compute something, you put it into a register and work on it. This notion causes a bit of trouble on some machines (e.g., machines with 3-address opcodes), but matches many machines quite well. Unfortunately, this notion breaks down with structures. The closest that one can come is to keep the addresses of the structures in registers. The actual code sequences used to move structures vary from the trivial (a multiple byte move) to the horrible (a function call), and are very machine dependent.

The practical problem is more painful. When a function returning a structure is called, this function has to have some place to put the structure value. If it places it on the stack, it has difficulty popping its stack frame. If it places the value in a static temporary, the routine fails to be reentrant. The most logically consistent way of implementing this is for the caller to pass in a pointer to a spot where the called function should put the value before returning. This is relatively straightforward, although a bit tedious, to implement, but means that the caller must have properly declared the function type, even if the value is never used. On some machines, such as the Interdata 8/32, the return value simply overlays the argument region (which on the 8/32 is part of the caller's stack frame). The caller takes care of leaving enough room if the returned value is larger than the arguments. This also assumes that the caller know and declares the function properly.

The PDP-11 and the VAX have stack hardware which is used in function calls and returns; this makes it very inconvenient to use either of the above mechanisms. In these machines, a static area within the called functionis allocated, and the function return value is copied into it on return; the function returns the address of that region. This is simple to implement, but is non-reentrant. However, the function can now be called as a subroutine without being properly declared, without the disaster which would otherwise ensue. No matter what choice is taken, the convention is that the function actually returns the address of the return structure value.

In building expression trees, the portable compiler takes a bit for granted about structures. It assumes that functions returning structures actually return a pointer to the structure, and it assumes that a reference to a structure is actually a reference to its address. The structure assignment operator is rebuilt so that the left operand is the structure being assigned to, but the right operand is the address of the structure being assigned; this makes it easier to deal with

$$a = b = c$$

and similar constructions.

There are four special tree nodes associated with these operations: STASG (structure assignment), STARG (structure argument to a function call), and STCALL and UNARY STCALL (calls of a function with nonzero and zero arguments, respectively). These four nodes are unique in that the size and alignment information, which can be determined by the type for all other objects in C, must be known to carry out these operations; special fields are set aside in these nodes to contain this information, and special intermediate code is used to transmit this information.

### First Pass Summary

There are may other issues which have been ignored here, partly to justify the title ‘‘tour’’, and partially because they have seemed to cause little trouble. There are some debugging flags which may be turned on, by giving the compiler’s first pass the argument

    −X[flags]

Some of the more interesting flags are −Xd for the defining and freeing of symbols, −Xi for initialization comments, and −Xb for various comments about the building of trees. In many cases, repeating the flag more than once gives more information; thus, −Xddd gives more information than −Xd. In the two pass version of the compiler, the flags should not be set when the output is sent to the second pass, since the debugging output and the intermediate code both go onto the standard output.

We turn now to consideration of the second pass.

### Pass Two

Code generation is far less well understood than parsing or lexical analysis, and for this reason the second pass is far harder to discuss in a file by file manner. A great deal of the difficulty is in understanding the issues and the strategies employed to meet them. Any particular function is likely to be reasonably straightforward.

Thus, this part of the paper will concentrate a good deal on the broader aspects of strategy in the code generator, and will not get too intimate with the details.

### Overview.

It is difficult to organize a code generator to be flexible enough to generate code for a large number of machines, and still be efficient for any one of them. Flexibility is also important when it comes time to tune the code generator to improve the output code quality. On the other hand, too much flexibility can lead to semantically incorrect code, and potentially a combinatorial explosion in the number of cases to be considered in the compiler.

One goal of the code generator is to have a high degree of correctness. It is very desirable to have the compiler detect its own inability to generate correct code, rather than to produce incorrect code. This goal is achieved by having a simple model of the job to be done (e.g., an expression tree) and a simple model of the machine state (e.g., which registers are free). The act of generating an instruction performs a transformation on the tree and the machine state; hopefully, the tree eventually gets reduced to a single node. If each of these instruction/transformation pairs is correct, and if the machine state model really represents the actual machine, and if the transformations reduce the input tree to the desired single node, then the output code will be correct.

For most real machines, there is no definitive theory of code generation that encompasses all the C operators. Thus the selection of which instruction/transformations to generate, and in what order, will have a heuristic flavor. If, for some expression tree, no transformation applies, or, more seriously, if the heuristics select a sequence of instruction/transformations that do not in fact reduce the tree, the compiler will report its inability to generate code, and abort.

A major part of the code generator is concerned with the model and the transformations, — most of this is machine independent, or depends only on simple tables. The flexibility comes from the heuristics that guide the transformations of the trees, the selection of subgoals, and the ordering of the computation.

**The Machine Model**

The machine is assumed to have a number of registers, of at most two different types: *A* and *B*. Within each register class, there may be scratch (temporary) registers and dedicated registers (e.g., register variables, the stack pointer, etc.). Requests to allocate and free registers involve only the temporary registers.

Each of the registers in the machine is given a name and a number in the *mac2defs* file; the numbers are used as indices into various tables that describe the registers, so they should be kept small. One such table is the *rstatus* table on file *local2.c*. This table is indexed by register number, and contains expressions made up from manifest constants describing the register types: SAREG for dedicated AREG's, SAREG|STAREG for scratch AREGS's, and SBREG and SBREG|STBREG similarly for BREG's. There are macros that access this information: *isbreg(r)* returns true if register number *r* is a BREG, and *istreg(r)* returns true if register number *r* is a temporary AREG or BREG. Another table, *rnames*, contains the register names; this is used when putting out assembler code and diagnostics.

The usage of registers is kept track of by an array called *busy*. *Busy[r]* is the number of uses of register *r* in the current tree being processed. The allocation and freeing of registers will be discussed later as part of the code generation algorithm.

**General Organization**

As mentioned above, the second pass reads lines from the intermediate file, copying through to the output unchanged any lines that begin with a ')', and making note of the information about stack usage and register allocation contained on lines beginning with ']' and '['. The expression trees, whose beginning is indicated by a line beginning with '.', are read and rebuilt into trees. If the compiler is loaded as one pass, the expression trees are immediately available to the code generator.

The actual code generation is done by a hierarchy of routines. The routine *delay* is first given the tree; it attempts to delay some postfix ++ and −− computations that might reasonably be done after the smoke clears. It also attempts to handle comma (,) operators by computing the left side expression first, and then rewriting the tree to eliminate the operator. *Delay* calls *codgen* to control the actual code generation process. *Codgen* takes as arguments a pointer to the expression tree, and a second argument that, for socio-historical reasons, is called a *cookie*. The cookie describes a set of goals that would be acceptable for the code generation: these are assigned to individual bits, so they may be logically or'ed together to form a large number of possible goals. Among the possible goals are FOREFF (compute for side effects only; don't worry about the value), INTEMP (compute and store value into a temporary location in memory), INAREG (compute into an A register), INTAREG (compute into a scratch A register), INBREG and INTBREG similarly, FORCC (compute for condition codes), and FORARG (compute it as a function argument; e.g., stack it if appropriate).

*Codgen* first canonicalizes the tree by calling *canon*. This routine looks for certain transformations that might now be applicable to the tree. One, which is very common and very powerful, is to fold together an indirection operator (UNARY MUL) and a register (REG); in most machines, this combination is addressable directly, and so is similar to a NAME in its behavior. The UNARY MUL and REG are folded together to make another node type called OREG. In fact, in many machines it is possible to directly address not just the cell pointed to by a register, but also cells differing by a constant offset from the cell pointed to by the register. *Canon* also looks for such cases, calling the machine dependent routine *notoff* to decide if the offset is acceptable (for example, in the IBM 370 the offset must be between 0 and 4095 bytes). Another optimization is to replace bit field operations by shifts and masks if the operation involves extracting the field. Finally, a machine dependent routine, *sucomp*, is called that computes the Sethi-Ullman numbers for the tree (see below).

After the tree is canonicalized, *codgen* calls the routine *store* whose job is to select a subtree of the tree to be computed and (usually) stored before beginning the computation of the full tree. *Store* must return a tree that can be computed without need for any temporary storage locations. In effect, the only store operations generated while processing the subtree must be as a response to explicit assignment operators in the tree. This division of the job marks one of the more significant, and successful, departures from most other compilers. It means that the code generator can operate under the assumption that there are enough registers to do its job, without worrying about temporary storage. If a store into a temporary appears in the output, it is always as a direct result of logic in the *store* routine; this makes debugging easier.

One consequence of this organization is that code is not generated by a treewalk. There are theoretical results that support this decision.[7] It may be desirable to compute several subtrees and store them before tackling the whole tree; if a subtree is to be stored, this is known before the code generation for the subtree is begun, and the subtree is computed when all scratch registers are available.

The *store* routine decides what subtrees, if any, should be stored by making use of numbers, called *Sethi-Ullman numbers*, that give, for each subtree of an expression tree, the minimum number of scratch registers required to compile the subtree, without any stores into temporaries.[8] These numbers are computed by the machine-dependent routine *sucomp*, called by *canon*. The basic notion is that, knowing the Sethi-Ullman numbers for the descendants of a node, and knowing the operator of the node and some information about the machine, the Sethi-Ullman number of the node itself can be computed. If the Sethi-Ullman number for a tree exceeds the number of scratch registers available, some subtree must be stored. Unfortunately, the theory behind the Sethi-Ullman numbers applies only to uselessly simple machines and operators. For the rich set of C operators, and for machines with asymmetric registers, register pairs, different kinds of registers, and exceptional forms of addressing, the theory cannot be applied directly. The basic idea of estimation is a good one, however, and well worth applying; the application, especially when the compiler comes to be tuned for high code quality, goes beyond the park of theory into the swamp of heuristics. This topic will be taken up again later, when more of the compiler structure has been described.

After examining the Sethi-Ullman numbers, *store* selects a subtree, if any, to be stored, and returns the subtree and the associated cookie in the external variables *stotree* and *stocook*. If a subtree has been selected, or if the whole tree is ready to be processed, the routine *order* is called, with a tree and cookie. *Order* generates code for trees that do not require temporary locations. *Order* may make recursive calls on itself, and, in some cases, on *codgen*; for example, when processing the operators &&, ||, and comma (‘,’), that have a left to right evaluation, it is incorrect for *store* examine the right operand for subtrees to be stored. In these cases, *order* will call *codgen* recursively when it is permissible to work on the right operand. A similar issue arises with the ? : operator.

The *order* routine works by matching the current tree with a set of code templates. If a template is discovered that will match the current tree and cookie, the associated assembly language statement or statements are generated. The tree is then rewritten, as specified by the template, to represent the effect of the output instruction(s). If no template match is found, first an attempt is made to find a match with a different cookie; for example, in order to compute an expression with cookie INTEMP (store into a temporary storage location), it is usually necessary to compute the expression into a scratch register first. If all attempts to match the tree fail, the heuristic part of the algorithm becomes dominant. Control is typically given to one of a number of machine-dependent routines that may in turn recursively call *order* to achieve a subgoal of the computation (for example, one of the arguments may be computed into a temporary register). After this subgoal has been achieved, the process begins again with the modified tree. If the machine-dependent heuristics are unable to reduce the tree further, a number of default rewriting rules may be considered appropriate. For example, if the left operand of a + is a scratch register, the + can be replaced by a += operator; the tree may then match a template.

To close this introduction, we will discuss the steps in compiling code for the expression

$a$ += $b$

where $a$ and $b$ are static variables.

To begin with, the whole expression tree is examined with cookie FOREFF, and no match is found. Search with other cookies is equally fruitless, so an attempt at rewriting is made. Suppose we are dealing with the Interdata 8/32 for the moment. It is recognized that the left hand and right hand sides of the += operator are addressable, and in particular the left hand side has no side effects, so it is permissible to rewrite this as

$a = a + b$

and this is done. No match is found on this tree either, so a machine dependent rewrite is done; it is recognized that the left hand side of the assignment is addressable, but the right hand side is not in a register, so *order* is called recursively, being asked to put the right hand side of the assignment into a register. This invocation of *order* searches the tree for a match, and fails. The machine dependent rule for + notices that the right hand operand is addressable; it decides to put the left operand into a scratch register. Another recursive call to *order* is made, with the tree consisting solely of the leaf $a$, and the cookie asking that the value be placed into a scratch register. This now matches a template, and a load instruction is emitted. The node consisting of $a$ is rewritten in place to represent the register into which $a$ is loaded, and this third call to *order* returns. The second call to *order* now finds that it has the tree

**reg** + $b$

to consider. Once again, there is no match, but the default rewriting rule rewrites the + as a += operator, since the left operand is a scratch register. When this is done, there is a match: in fact,

**reg** += $b$

simply describes the effect of the add instruction on a typical machine. After the add is emitted, the tree is rewritten to consist merely of the register node, since the result of the add is now in the register. This agrees with the cookie passed to the second invocation of *order*, so this invocation terminates, returning to the first level. The original tree has now become

$a$ = **reg**

which matches a template for the store instruction. The store is output, and the tree rewritten to become just a single register node. At this point, since the top level call to *order* was interested only in side effects, the call to *order* returns, and the code generation is completed; we have generated a load, add, and store, as might have been expected.

The effect of machine architecture on this is considerable. For example, on the Honeywell 6000, the machine dependent heuristics recognize that there is an ‘‘add to storage’’ instruction, so the strategy is quite different; $b$ is loaded in to a register, and then an add to storage instruction generated to add this register in to $a$. The transformations, involving as they do the semantics of C, are largely machine independent. The decisions as to when to use them, however, are almost totally machine dependent.

Having given a broad outline of the code generation process, we shall next consider the heart of it: the templates. This leads naturally into discussions of template matching and register allocation, and finally a discussion of the machine dependent interfaces and strategies.

**The Templates**

The templates describe the effect of the target machine instructions on the model of computation around which the compiler is organized. In effect, each template has five logical sections, and represents an assertion of the form:

> **If** we have a subtree of a given shape (1), and we have a goal (cookie) or goals to achieve (2), and we have sufficient free resources (3), **then** we may emit an instruction or instructions (4), and rewrite the subtree in a particular manner (5), and the rewritten tree will achieve the desired goals.

These five sections will be discussed in more detail later. First, we give an example of a template:

```
ASG PLUS,   INAREG,
            SAREG,      TINT,
            SNAME,      TINT,
                        0,          RLEFT,
                        "           add         AL,AR\n",
```

The top line specifies the operator (+=) and the cookie (compute the value of the subtree into an AREG). The second and third lines specify the left and right descendants, respectively, of the += operator. The left descendant must be a REG node, representing an A register, and have integer type, while the right side must be a NAME node, and also have integer type. The fourth line contains the resource requirements (no scratch registers or temporaries needed), and the rewriting rule (replace the subtree by the left descendant). Finally, the quoted string on the last line represents the output to the assembler: lower case letters, tabs, spaces, etc. are copied *verbatim*. to the output; upper case letters trigger various macro-like expansions. Thus, **AL** would expand into the **A**ddress form of the **L**eft operand — presumably the register number. Similarly, **AR** would expand into the name of the right operand. The *add* instruction of the last section might well be emitted by this template.

In principle, it would be possible to make separate templates for all legal combinations of operators, cookies, types, and shapes. In practice, the number of combinations is very large. Thus, a considerable amount of mechanism is present to permit a large number of subtrees to be matched by a single template. Most of the shape and type specifiers are individual bits, and can be logically or'ed together. There are a number of special descriptors for matching classes of operators. The cookies can also be combined. As an example of the kind of template that really arises in practice, the actual template for the Interdata 8/32 that subsumes the above example is:

```
ASG OPSIMP, INAREG|FORCC,
            SAREG,      TINT|TUNSIGNED|TPOINT,
            SAREG|SNAME|SOREG|SCON, TINT|TUNSIGNED|TPOINT,
                        0,          RLEFT|RESCC,
                        "           OI          AL,AR\n",
```

Here, OPSIMP represents the operators +, −, |, &, and ˆ. The **OI** macro in the output string expands into the appropriate **I**nteger **O**pcode for the operator. The left and right sides can be integers, unsigned, or pointer types. The right side can be, in addition to a name, a register, a memory location whose address is given by a register and displacement (OREG), or a constant. Finally, these instructions set the condition codes, and so can be used in condition contexts: the cookie and rewriting rules reflect this.

**The Template Matching Algorithm.**

The heart of the second pass is the template matching algorithm, in the routine *match*. *Match* is called with a tree and a cookie; it attempts to match the given tree against some template that will transform it according to one of the goals given in the cookie. If a match is successful, the transformation is applied; *expand* is called to generate the assembly code, and then *reclaim* rewrites the tree, and reclaims the resources, such as registers, that might have become free as a result of the generated code.

This part of the compiler is among the most time critical. There is a spectrum of implementation techniques available for doing this matching. The most naive algorithm simply looks at the templates one by one. This can be considerably improved upon by restricting the search for an acceptable template. It would be possible to do better than this if the templates were given to a separate program that ate them and generated a template matching subroutine. This would make maintenance of the compiler much more complicated, however, so this has not been done.

The matching algorithm is actually carried out by restricting the range in the table that must be searched for each opcode. This introduces a number of complications, however, and needs a bit of sympathetic help by the person constructing the compiler in order to obtain best results. The exact tuning of this algorithm continues; it is best to consult the code and comments in *match* for the latest version.

In order to match a template to a tree, it is necessary to match not only the cookie and the op of the root, but also the types and shapes of the left and right descendants (if any) of the tree. A convention is established here that is carried out throughout the second pass of the compiler. If a node represents a unary operator, the single descendant is always the ''left'' descendant. If a node represents a unary operator or a leaf node (no descendants) the ''right'' descendant is taken by convention to be the node itself. This enables templates to easily match leaves and conversion operators, for example, without any additional mechanism in the matching program.

The type matching is straightforward; it is possible to specify any combination of basic types, general pointers, and pointers to one or more of the basic types. The shape matching is somewhat more complicated, but still pretty simple. Templates have a collection of possible operand shapes on which the opcode might match. In the simplest case, an *add* operation might be able to add to either a register variable or a scratch register, and might be able (with appropriate help from the assembler) to add an integer constant (ICON), a static memory cell (NAME), or a stack location (OREG).

It is usually attractive to specify a number of such shapes, and distinguish between them when the assembler output is produced. It is possible to describe the union of many elementary shapes such as ICON, NAME, OREG, AREG or BREG (both scratch and register forms), etc. To handle at least the simple forms of indirection, one can also match some more complicated forms of trees; STARNM and STARREG can match more complicated trees headed by an indirection operator, and SFLD can match certain trees headed by a FLD operator: these patterns call machine dependent routines that match the patterns of interest on a given machine. The shape SWADD may be used to recognize NAME or OREG nodes that lie on word boundaries: this may be of some importance on word–addressed machines. Finally, there are some special shapes: these may not be used in conjunction with the other shapes, but may be defined and extended in machine dependent ways. The special shapes SZERO, SONE, and SMONE are predefined and match constants 0, 1, and −1, respectively; others are easy to add and match by using the machine dependent routine *special*.

When a template has been found that matches the root of the tree, the cookie, and the shapes and types of the descendants, there is still one bar to a total match: the template may call for some resources (for example, a scratch register). The routine *allo* is called, and it attempts to allocate the resources. If it cannot, the match fails; no resources are allocated. If successful, the allocated resources are given numbers 1, 2, etc. for later reference when the assembly code is generated. The routines *expand* and *reclaim* are then called. The *match* routine then returns a special value, MDONE. If no match was found, the value MNOPE is returned; this is a signal to the caller to try more cookie values, or attempt a rewriting rule. *Match* is also used to select rewriting rules, although the way of doing this is pretty straightforward. A special cookie, FORREW, is used to ask *match* to search for a rewriting rule. The rewriting rules are keyed to various opcodes; most are carried out in *order*. Since the question of when to rewrite is one of the key issues in code generation, it will be taken up again later.

**Register Allocation.**

The register allocation routines, and the allocation strategy, play a central role in the correctness of the code generation algorithm. If there are bugs in the Sethi-Ullman computation that cause the number of needed registers to be underestimated, the compiler may run out of scratch registers; it is essential that the allocator keep track of those registers that are free and busy, in order to detect such conditions.

Allocation of registers takes place as the result of a template match; the routine *allo* is called with a word describing the number of A registers, B registers, and temporary locations needed. The allocation of temporary locations on the stack is relatively straightforward, and will not be further covered; the bookkeeping is a bit tricky, but conceptually trivial, and requests for temporary space on the stack will never fail.

Register allocation is less straightforward. The two major complications are *pairing* and *sharing*. In many machines, some operations (such as multiplication and division), and/or some types (such as longs or double precision) require even/odd pairs of registers. Operations of the first type are exceptionally difficult to deal with in the compiler; in fact, their theoretical properties are rather bad as well.[9] The second issue is dealt with rather more successfully; a machine dependent function called *szty(t)* is called that returns 1 or 2, depending on the number of A registers required to hold an object of type *t*. If *szty* returns 2, an even/odd pair of A registers is allocated for each request.

The other issue, sharing, is more subtle, but important for good code quality. When registers are allocated, it is possible to reuse registers that hold address information, and use them to contain the values computed or accessed. For example, on the IBM 360, if register 2 has a pointer to an integer in it, we may load the integer into register 2 itself by saying:

```
L       2,0(2)
```

If register 2 had a byte pointer, however, the sequence for loading a character involves clearing the target register first, and then inserting the desired character:

```
SR      3,3
IC      3,0(2)
```

In the first case, if register 3 were used as the target, it would lead to a larger number of registers used for the expression than were required; the compiler would generate inefficient code. On the other hand, if register 2 were used as the target in the second case, the code would simply be wrong. In the first case, register 2 can be *shared* while in the second, it cannot.

In the specification of the register needs in the templates, it is possible to indicate whether required scratch registers may be shared with possible registers on the left or the right of the input tree. In order that a register be shared, it must be scratch, and it must be used only once, on the appropriate side of the tree being compiled.

The *allo* routine thus has a bit more to do than meets the eye; it calls *freereg* to obtain a free register for each A and B register request. *Freereg* makes multiple calls on the routine *usable* to decide if a given register can be used to satisfy a given need. *Usable* calls *shareit* if the register is busy, but might be shared. Finally, *shareit* calls *ushare* to decide if the desired register is actually in the appropriate subtree, and can be shared.

Just to add additional complexity, on some machines (such as the IBM 370) it is possible to have ‘‘double indexing’’ forms of addressing; these are represented by OREGS’s with the base and index registers encoded into the register field. While the register allocation and deallocation *per se* is not made more difficult by this phenomenon, the code itself is somewhat more complex.

Having allocated the registers and expanded the assembly language, it is time to reclaim the resources; the routine *reclaim* does this. Many operations produce more than one result. For example, many arithmetic operations may produce a value in a register, and also set the condition codes. Assignment operations may leave results both in a register and in memory. *Reclaim* is passed three parameters; the tree and cookie that were matched, and the rewriting field of the template. The rewriting field allows the specification of possible results; the tree is rewritten to reflect the results of the operation. If the tree was computed for side effects only (FOREFF), the tree is freed, and all resources in it reclaimed. If the tree was computed for condition codes, the resources are also freed, and the tree replaced by a special node type, FORCC. Otherwise, the value may be found in the left argument of the root, the right argument of the root, or one of the temporary resources allocated. In these cases, first the resources of the tree, and the newly allocated resources, are freed; then the resources needed by the result are made busy again. The final result must always match the shape of the input cookie; otherwise, the compiler error ‘‘cannot reclaim’’ is generated. There are some machine dependent ways of preferring results in registers or memory when there are multiple results matching multiple goals in the cookie.

**The Machine Dependent Interface**

The files *order.c*, *local2.c*, and *table.c*, as well as the header file *mac2defs*, represent the machine dependent portion of the second pass. The machine dependent portion can be roughly divided into two: the easy portion and the hard portion. The easy portion tells the compiler the names of the registers, and arranges that the compiler generate the proper assembler formats, opcode names, location counters, etc. The hard portion involves the Sethi–Ullman computation, the rewriting rules, and, to some extent, the templates. It is hard because there are no real algorithms that apply; most of this portion is based on heuristics. This section discusses the easy portion; the next several sections will discuss the hard portion.

If the compiler is adapted from a compiler for a machine of similar architecture, the easy part is indeed easy. In *mac2defs*, the register numbers are defined, as well as various parameters for the stack frame, and various macros that describe the machine architecture. If double indexing is to be permitted, for example, the symbol R2REGS is defined. Also, a number of macros that are involved in function call processing, especially for unusual function call mechanisms, are defined here.

In *local2.c*, a large number of simple functions are defined. These do things such as write out opcodes, register names, and address forms for the assembler. Part of the function call code is defined here; that is nontrivial to design, but typically rather straightforward to implement. Among the easy routines in *order.c* are routines for generating a created label, defining a label, and generating the arguments of a function call.

These routines tend to have a local effect, and depend on a fairly straightforward way on the target assembler and the design decisions already made about the compiler. Thus they will not be further treated here.

**The Rewriting Rules**

When a tree fails to match any template, it becomes a candidate for rewriting. Before the tree is rewritten, the machine dependent routine *nextcook* is called with the tree and the cookie; it suggests another cookie that might be a better candidate for the matching of the tree. If all else fails, the templates are searched with the cookie FORREW, to look for a rewriting rule. The rewriting rules are of two kinds; for most of the common operators, there are machine dependent rewriting rules that may be applied; these are handled by machine dependent functions that are called and given the tree to be computed. These routines may recursively call *order* or *codgen* to cause certain subgoals to be achieved; if they actually call for some alteration of the tree, they return 1, and the code generation algorithm recanonicalizes and tries again. If these routines choose not to deal with the tree, the default rewriting rules are applied.

The assignment ops, when rewritten, call the routine *setasg*. This is assumed to rewrite the tree at least to the point where there are no side effects in the left hand side. If there is still no template match, a default rewriting is done that causes an expression such as

$$a \mathrel{+}= b$$

to be rewritten as

$$a = a + b$$

This is a useful default for certain mixtures of strange types (for example, when $a$ is a bit field and $b$ an character) that otherwise might need separate table entries.

Simple assignment, structure assignment, and all forms of calls are handled completely by the machine dependent routines. For historical reasons, the routines generating the calls return 1 on failure, 0 on success, unlike the other routines.

The machine dependent routine *setbin* handles binary operators; it too must do most of the job. In particular, when it returns 0, it must do so with the left hand side in a temporary register. The default rewriting rule in this case is to convert the binary operator into the associated assignment operator; since the left hand side is assumed to be a temporary register, this preserves the semantics and often allows a considerable saving in the template table.

The increment and decrement operators may be dealt with with the machine dependent routine *setincr*. If this routine chooses not to deal with the tree, the rewriting rule replaces

$$x{+}{+}$$

by

*( (x += 1) − 1)*

which preserves the semantics. Once again, this is not too attractive for the most common cases, but can generate close to optimal code when the type of x is unusual.

Finally, the indirection (UNARY MUL) operator is also handled in a special way. The machine dependent routine *offstar* is extremely important for the efficient generation of code. *Offstar* is called with a tree that is the direct descendant of a UNARY MUL node; its job is to transform this tree so that the combination of UNARY MUL with the transformed tree becomes addressable. On most machines, *offstar* can simply compute the tree into an A or B register, depending on the architecture, and then *canon* will make the resulting tree into an OREG. On many machines, *offstar* can profitably choose to do less work than computing its entire argument into a register. For example, if the target machine supports OREGS with a constant offset from a register, and *offstar* is called with a tree of the form

*expr + const*

where *const* is a constant, then *offstar* need only compute *expr* into the appropriate form of register. On machines that support double indexing, *offstar* may have even more choice as to how to proceed. The proper tuning of *offstar*, which is not typically too difficult, should be one of the first tries at optimization attempted by the compiler writer.

**The Sethi-Ullman Computation**

The heart of the heuristics is the computation of the Sethi-Ullman numbers. This computation is closely linked with the rewriting rules and the templates. As mentioned before, the Sethi-Ullman numbers are expected to estimate the number of scratch registers needed to compute the subtrees without using any stores. However, the original theory does not apply to real machines. For one thing, the theory assumes that all registers are interchangeable. Real machines have general purpose, floating point, and index registers, register pairs, etc. The theory also does not account for side effects; this rules out various forms of pathology that arise from assignment and assignment ops. Condition codes are also undreamed of. Finally, the influence of types, conversions, and the various addressability restrictions and extensions of real machines are also ignored.

Nevertheless, for a ''useless'' theory, the basic insight of Sethi and Ullman is amazingly useful in a real compiler. The notion that one should attempt to estimate the resource needs of trees before starting the code generation provides a natural means of splitting the code generation problem, and provides a bit of redundancy and self checking in the compiler. Moreover, if writing the Sethi-Ullman routines is hard, describing, writing, and debugging the alternative (routines that attempt to free up registers by stores into temporaries ''on the fly'') is even worse. Nevertheless, it should be clearly understood that these routines exist in a realm where there is no ''right'' way to write them; it is an art, the realm of heuristics, and, consequently, a major source of bugs in the compiler. Often, the early, crude versions of these routines give little trouble; only after the compiler is actually working and the code quality is being improved do serious problem have to be faced. Having a simple, regular machine architecture is worth quite a lot at this time.

The major problems arise from asymmetries in the registers: register pairs, having different kinds of registers, and the related problem of needing more than one register (frequently a pair) to store certain data types (such as longs or doubles). There appears to be no general way of treating this problem; solutions have to be fudged for each machine where the problem arises. On the Honeywell 66, for example, there are only two general purpose registers, so a need for a pair is the same as the need for two registers. On the IBM 370, the register pair (0,1) is used to do multiplications and divisions; registers 0 and 1 are not generally considered part of the scratch registers, and so do not require allocation explicitly. On the Interdata 8/32, after much consideration, the decision was made not to try to deal with the register pair issue; operations such as multiplication and division that required pairs were simply assumed to take all of the scratch registers. Several weeks of effort had failed to produce an algorithm that seemed to have much chance of running successfully without inordinate debugging effort. The difficulty of this issue should not be minimized; it represents one of the main intellectual efforts in porting the compiler. Nevertheless, this problem has been fudged with a degree of success on nearly a dozen machines, so the compiler writer should not abandon hope.

The Sethi-Ullman computations interact with the rest of the compiler in a number of rather subtle ways. As already discussed, the *store* routine uses the Sethi-Ullman numbers to decide which subtrees are too difficult to compute in registers, and must be stored. There are also subtle interactions between the rewriting routines and the Sethi-Ullman numbers. Suppose we have a tree such as

$$A - B$$

where $A$ and $B$ are expressions; suppose further that $B$ takes two registers, and $A$ one. It is possible to compute the full expression in two registers by first computing $B$, and then, using the scratch register used by $B$, but not containing the answer, compute $A$. The subtraction can then be done, computing the expression. (Note that this assumes a number of things, not the least of which are register-to-register subtraction operators and symmetric registers.) If the machine dependent routine *setbin*, however, is not prepared to recognize this case and compute the more difficult side of the expression first, the Sethi-Ullman number must be set to three. Thus, the Sethi-Ullman number for a tree should represent the code that the machine dependent routines are actually willing to generate.

The interaction can go the other way. If we take an expression such as

$$*\,(\,p + i\,)$$

where $p$ is a pointer and $i$ an integer, this can probably be done in one register on most machines. Thus, its Sethi-Ullman number would probably be set to one. If double indexing is possible in the machine, a possible way of computing the expression is to load both $p$ and $i$ into registers, and then use double indexing. This would use two scratch registers; in such a case, it is possible that the scratch registers might be unobtainable, or might make some other part of the computation run out of registers. The usual solution is to cause *offstar* to ignore opportunities for double indexing that would tie up more scratch registers than the Sethi-Ullman number had reserved.

In summary, the Sethi-Ullman computation represents much of the craftsmanship and artistry in any application of the portable compiler. It is also a frequent source of bugs. Algorithms are available that will produce nearly optimal code for specialized machines, but unfortunately most existing machines are far removed from these ideals. The best way of proceeding in practice is to start with a compiler for a similar machine to the target, and proceed very carefully.

**Register Allocation**

After the Sethi-Ullman numbers are computed, *order* calls a routine, *rallo*, that does register allocation, if appropriate. This routine does relatively little, in general; this is especially true if the target machine is fairly regular. There are a few cases where it is assumed that the result of a computation takes place in a particular register; switch and function return are the two major places. The expression tree has a field, *rall*, that may be filled with a register number; this is taken to be a preferred register, and the first temporary register allocated by a template match will be this preferred one, if it is free. If not, no particular action is taken; this is just a heuristic. If no register preference is present, the field contains NOPREF. In some cases, the result must be placed in a given register, no matter what. The register number is placed in *rall*, and the mask MUSTDO is logically or'ed in with it. In this case, if the subtree is requested in a register, and comes back in a register other than the demanded one, it is moved by calling the routine *rmove*. If the target register for this move is busy, it is a compiler error.

Note that this mechanism is the only one that will ever cause a register-to-register move between scratch registers (unless such a move is buried in the depths of some template). This simplifies debugging. In some cases, there is a rather strange interaction between the register allocation and the Sethi-Ullman number; if there is an operator or situation requiring a particular register, the allocator and the Sethi-Ullman computation must conspire to ensure that the target register is not being used by some intermediate result of some far-removed computation. This is most easily done by making the special operation take all of the free registers, preventing any other partially-computed results from cluttering up the works.

**Compiler Bugs**

The portable compiler has an excellent record of generating correct code. The requirement for reasonable cooperation between the register allocation, Sethi-Ullman computation, rewriting rules, and templates builds quite a bit of redundancy into the compiling process. The effect of this is that, in a surprisingly short time, the compiler will start generating correct code for those programs that it can compile. The hard part of the job then becomes finding and eliminating those situations where the compiler refuses to compile a program because it knows it cannot do it right. For example, a template may simply be missing; this may either give a compiler error of the form ''no match for op ...'' , or cause the compiler to go into an infinite loop applying various rewriting rules. The compiler has a variable, *nrecur*, that is set to 0 at the beginning of an expressions, and incremented at key spots in the compilation process; if this parameter gets too large, the compiler decides that it is in a loop, and aborts. Loops are also characteristic of botches in the machine-dependent rewriting rules. Bad Sethi-Ullman computations usually cause the scratch registers to run out; this often means that the Sethi-Ullman number was underestimated, so *store* did not store something it should have; alternatively, it can mean that the rewriting rules were not smart enough to find the sequence that *sucomp* assumed would be used.

The best approach when a compiler error is detected involves several stages. First, try to get a small example program that steps on the bug. Second, turn on various debugging flags in the code generator, and follow the tree through the process of being matched and rewritten. Some flags of interest are −e, which prints the expression tree, −r, which gives information about the allocation of registers, −a, which gives information about the performance of *rallo*, and −o, which gives information about the behavior of *order*. This technique should allow most bugs to be found relatively quickly.

Unfortunately, finding the bug is usually not enough; it must also be fixed! The difficulty arises because a fix to the particular bug of interest tends to break other code that already works. Regression tests, tests that compare the performance of a new compiler against the performance of an older one, are very valuable in preventing major catastrophes.

**Summary and Conclusion**

The portable compiler has been a useful tool for providing C capability on a large number of diverse machines, and for testing a number of theoretical constructs in a practical setting. It has many blemishes, both in style and functionality. It has been applied to many more machines than first anticipated, of a much wider range than originally dreamed of. Its use has also spread much faster than expected, leaving parts of the compiler still somewhat raw in shape.

On the theoretical side, there is some hope that the skeleton of the *sucomp* routine could be generated for many machines directly from the templates; this would give a considerable boost to the portability and correctness of the compiler, but might affect tunability and code quality. There is also room for more optimization, both within *optim* and in the form of a portable ''peephole'' optimizer.

On the practical, development side, the compiler could probably be sped up and made smaller without doing too much violence to its basic structure. Parts of the compiler deserve to be rewritten; the initialization code, register allocation, and parser are prime candidates. It might be that doing some or all of the parsing with a recursive descent parser might save enough space and time to be worthwhile; it would certainly ease the problem of moving the compiler to an environment where *Yacc* is not already present.

Finally, I would like to thank the many people who have sympathetically, and even enthusiastically, helped me grapple with what has been a frustrating program to write, test, and install. D. M. Ritchie and E. N. Pinson provided needed early encouragement and philosophical guidance; M. E. Lesk, R. Muha, T. G. Peterson, G. Riddle, L. Rosler, R. W. Mitze, B. R. Rowland, S. I. Feldman, and T. B. London have all contributed ideas, gripes, and all, at one time or another, climbed ''into the pits'' with me to help debug. Without their help this effort would have not been possible; with it, it was often kind of fun.

**References**

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).

2. S. C. Johnson, ''Lint, a C Program Checker,'' Comp. Sci. Tech. Rep. No. 65 (1978). updated version TM 78-1273-3

3. A. Snyder, *A Portable Compiler for the Language C,* Master's Thesis, M.I.T., Cambridge, Mass. (1974).

4. S. C. Johnson, ''A Portable Compiler: Theory and Practice,'' *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp.97-104 (January 1978).

5. M. E. Lesk, S. C. Johnson, and D. M. Ritchie, *The C Language Calling Sequence*, 1977.

6. S. C. Johnson, ''Yacc — Yet Another Compiler-Compiler,'' Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

7. A. V. Aho and S. C. Johnson, ''Optimal Code Generation for Expression Trees,'' *J. Assoc. Comp. Mach.* **23**(3), pp.488-501 (1975). Also in *Proc. ACM Symp. on Theory of Computing,* pp. 207-217, 1975.

8. R. Sethi and J. D. Ullman, ''The Generation of Optimal Code for Arithmetic Expressions,'' *J. Assoc. Comp. Mach.* **17**(4), pp.715-728 (October 1970). Reprinted as pp. 229-247 in *Compiler Techniques*, ed. B. W. Pollack, Auerbach, Princeton NJ (1972).

9. A. V. Aho, S. C. Johnson, and J. D. Ullman, ''Code Generation for Machines with Multiregister Operations,'' *Proc. 4th ACM Symp. on Principles of Programming Languages*, pp.21-28 (January 1977).

# A Dial-Up Network of UNIX™ Systems

*D. A. Nowitz*

*M. E. Lesk*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

A network of over eighty UNIX† computer systems has been established using the telephone system as its primary communication medium. The network was designed to meet the growing demands for software distribution and exchange. Some advantages of our design are:

- The startup cost is low. A system needs only a dial-up port, but systems with automatic calling units have much more flexibility.
- No operating system changes are required to install or use the system.
- The communication is basically over dial-up lines, however, hardwired communication lines can be used to increase speed.
- The command for sending/receiving files is simple to use.

Keywords: networks, communications, software distribution, software maintenance

August 18, 1978

†UNIX is a Trademark of Bell Laboratories.

# A Dial-Up Network of UNIX™ Systems

*D. A. Nowitz*

*M. E. Lesk*

Bell Laboratories
Murray Hill, New Jersey 07974

## 1. Purpose

The widespread use of the UNIX† system[1] within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back an forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only.[23] Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

## 2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes; in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish ''active'' and ''passive'' systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there; this makes a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud;

†UNIX is a Trademark of Bell Laboratories.

hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.[45] Thus, systems typically join our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustratives the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

## 3. Processing

The user has two commands which set up communications, *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

### File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.
- Place a call to a remote system.

- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

*Uucico* may be started in several ways;

a) by a system daemon,

b) by one of the *uucp* or *uux* programs,

c) by a remote system.

**Scan For Work**

The file names in the spool directory are constructed to allow the daemon programs *(uucico, uuxqt)* to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

**Call Remote System**

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a ‘‘systems’’ file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number,

[6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. ‘‘nyc’’, ‘‘boston’’) which get translated into dial sequences using a ‘‘dial-codes’’ file. This permits the same ‘‘phone number’’ to be stored at every site, despite local variations in telephone services and dialing conventions.

A ‘‘devices’’ file is scanned using fields [3] and [4] from the ‘‘systems’’ file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

**Line Protocol Selection**

The remote system sends a message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the uucp transmission program. Other protocols may be added by individual installations.

**Work Processing**

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER.* These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

center; c l. S send a file, R receive a file, C copy complete, H hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, a *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

**Conversation Termination**

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

**4. Present Uses**

One application of this software is remote mail. Normally, a UNIX system user writes ‘‘mail dan’’ to send mail to user ‘‘dan’’. By writing ‘‘mail usg!dan’’ the mail is sent to user ‘‘dan’’ on system ‘‘usg’’.

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a ‘‘stockroom’’ which allows remote users to call in and request software. He keeps a ‘‘stock list’’ of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's ‘‘diff’’ program which compares two text files and indicates the differences, line by line, between them.[6] Only lines which are not identical are printed. Similarly, the program ''uudiff'' compares files (or directories) on two machines. One of these directories may be on a passive system. The ''uudiff'' program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The ''uux'' command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

### 5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

center; c c n n. Nominal speed Characters/sec. 300 baud 27 1200 baud 100-110 9600 baud 200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.

The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

### 6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages.[7] There are distribution errors and delays, resulting in old software running at remote sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The ''stockroom'' serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the ''latest version'' that has just arrived; all too often it features the ''latest bug.'' What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system ‘‘mail’’ and ‘‘diff,’’ plus the many implied commands represented by ‘‘uux.’’ However, we still need inter-system ‘‘write’’ (real-time inter-user communication) and ‘‘who’’ (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

## 7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.
2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.
3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.
4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

## Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver and protocol, and A. S. Cohen, J. Lions, and P. F. Long for their suggestions and assistance.

## References

1. D. M. Ritchie and K. Thompson, ‘‘The UNIX Time-Sharing System,’’ *Bell Sys. Tech. J.* **57**(6), pp.1905-1929 (1978).
2. T. A. Dolotta, R. C. Haight, and J. R. Mashey, ‘‘UNIX Time-Sharing System: The Programmer’s Workbench,’’ *Bell Sys. Tech. J.* **57**(6), pp.2177-2200 (1978).
3. G. L. Chesson, ‘‘The Network UNIX System,’’ *Operating Systems Review* **9**(5), pp.60-66, Also in *Proc. 5th Symp. on Operating Systems Principles.* (1975).
4. A. G. Fraser, ‘‘Spider — An Experimental Data Communications System,’’ *Proc. IEEE Conf. on Communications*, p.21F (June 1974). IEEE Cat. No. 74CH0859-9-CSCB.
5. A. G. Fraser, ‘‘A Virtual Channel Network,’’ *Datamation*, pp.51-56 (February 1975).
6. J. W. Hunt and M. D. McIlroy, ‘‘An Algorithm for Differential File Comparison,’’ Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey (June 1976).
7. F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, Mass. (1975).

# Uucp Implementation Description

*D. A. Nowitz*

*ABSTRACT*

Uucp is a series of programs designed to permit communication between UNIX systems using either dial-up or hardwired communication lines. This document gives a detailed implementation description of the current (second) implementation of uucp.

This document is for use by an administrator/installer of the system. It is not meant as a user's guide.

October 31, 1978

# Uucp Implementation Description

*D. A. Nowitz*

## Introduction

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution. The first version of the system was designed and implemented by M. E. Lesk.[1] This paper describes the current (second) implementation of the system.

Uucp is a batch type operation. Files are created in a spool directory for processing by the uucp demons. There are three types of files used for the execution of work. *Data files* contain data for transfer to remote systems. *Work files* contain directions for file transfers between systems. *Execution files* are directions for UNIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

- uucp — This program creates work and gathers data files in the spool directory for the transmission of files.
- uux — This program creates work files, execute files and gathers data files for the remote execution of UNIX commands.
- uucico — This program executes the work files for data transmission.
- uuxqt — This program executes the execution files for UNIX command execution.

The secondary programs are:

- uulog — This program updates the log file with new entries and reports on the status of uucp requests.
- uuclean — This program removes old files from the spool directory.

The remainder of this paper will describe the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

## 1. Uucp - UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The *uucp* command was designed to look like *cp* to the user. The syntax is

*uucp* **[** option **]** ... source ... destination

where the source and destination may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

- –d — Make directories when necessary for copying the file.
- –c — Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.

---

†UNIX is a Trademark of Bell Laboratories.

1 M. E. Lesk and A. S. Cohen, UNIX Software Distribution by Communication Link, private communication.

−g*letter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)

−m Send mail on completion of the work.

The following options are used primarily for debugging:

−r Queue the job but do not start *uucico* program.

−s*dir* Use directory *dir* for the spool directory.

−x*num* *Num* is the level of debugging output desired.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as ''*?*[]*''. If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

uucp *.c usg!/usr/dan

will set up the transfer of all files whose names end with ''.c'' to the ''/usr/dan'' directory on the''usg'' machine.

The source and/or destination names may also contain a *~user* prefix. This translates to the login directory on the specified system. For names with partial path-names, the current directory is prepended to the file name. File names with *../* are not permitted.

The command

uucp usg!~dan/*.h ~dan

will set up the transfer of files whose names end with ''.h'' in dan's login directory on system ''usg'' to dan's local login directory.

For each source file, the program will check the source and destination file-names and the system-part of each to classify the work into one of five types:

[1] Copy source to destination on local system.

[2] Receive files from other systems.

[3] Send files to a remote systems.

[4] Send files from remote systems to another remote system.

[5] Receive files from remote systems when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the −r option was specified).

**Type 1**

A *cp* command is used to do the work. The −*d* and the −*m* options are not honored in this case.

**Type 2**

A one line *work file* is created for each file requested and put in the spool directory with the following fields, each separated by a blank. (All *work files* and *execute files* use a blank as the field separator.)

[1] R

[2] The full path-name of the source or a ~user/path-name. The *~user* part will be expanded on the remote system.

[3] The full path-name of the destination file. If the *~user* notation is used, it will be immediately expanded to be the login directory for the user.

[4] The user's login name.

[5] A ''–'' followed by an option list. (Only the –m and –d options will appear in this list.)

**Type 3**

For each source file, a *work file* is created and the source file is copied into a *data file* in the spool directory. (A ''–c'' option on the *uucp* command will prevent the *data file* from being made.) In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

[1] S

[2] The full-path name of the source file.

[3] The full-path name of the destination or ~user/file-name.

[4] The user's login name.

[5] A ''–'' followed by an option list.

[6] The name of the *data file* in the spool directory.

[7] The file mode bits of the source file in octal print format (e.g. 0666).

**Type 4 and Type 5**

*Uucp* generates a *uucp* command and sends it to the remote machine; the remote *uucico* executes the *uucp* command.

**2. Uux - UNIX To UNIX Execution**

The *uux* command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of the uux command is

*uux* **[** – **] [** option **]** ... command-string

where the command-string is made up of one or more arguments. All special shell characters such as ''<>|^'' must be quoted either by quoting the entire command-string or quoting the character as a separate argument. Within the command-string, the command and file names may contain a *system-name!* prefix. All arguments which do not contain a ''!'' will not be treated as files. (They will not be copied to the execution machine.) The ''–'' is used to indicate that the standard input for *command-string* should be inherited from the standard input of the *uux* command. The options, essentially for debugging, are:

–r Don't start *uucico* or *uuxqt* after queuing the job;

–x*num* Num is the level of debugging output desired.

The command

pr abc | uux – usg!lpr

will set up the output of ''pr abc'' as standard input to an lpr command to be executed on system ''usg''.

*Uux* generates an *execute file* which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed by the *uucico* program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*. )

The *execute file* will be processed by the *uuxqt* program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

**User Line**

U user system

where the *user* and *system* are the requester's login name and system.

**Required File Line**

F file-name real-name

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the *execute file*. The *uuxqt* program will check for the existence of all required files before the command is executed.

**Standard Input Line**

I file-name

The standard input is either specified by a ‘‘<’’ in the command-string or inherited from the standard input of the *uux* command if the ‘‘–’’ option is used. If a standard input is not specified, ‘‘/dev/null’’ is used.

**Standard Output Line**

O file-name system-name

The standard output is specified by a ‘‘>’’ within the command-string. If a standard output is not specified, ‘‘/dev/null’’ is used. (Note – the use of ‘‘>>’’ is not implemented.)

**Command Line**

C command **[** arguments **]** ...

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All *required files* will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command is executed using the Shell specified in the *uucp.h* header file. In addition, a shell ‘‘PATH’’ statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

## 3. Uucico - Copy In, Copy Out

The *uucico* program will perform the following major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

*Uucico* may be started in several ways;

a) by a system daemon,

b) by one of the *uucp, uux, uuxqt* or *uucico* programs,

c) directly by the user (this is usually for testing),

d) by a remote system. (The uucico program should be specified as the ‘‘shell’’ field in the ‘‘/etc/passwd’’ file for the ‘‘uucp’’ logins.)

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If no system name is specified (−s option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

−r1 Start the program in *MASTER* mode. This is used when *uucico* is started by a program or ''cron'' shell.

−s*sys* Do work only for system *sys.* If −*s* is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

−d*dir* Use directory *dir* for the spool directory.

−x*num* *Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the *uucico* program.

**Scan For Work**

The names of the work related files in the spool directory have format

type . system-name grade number

where:

*Type* is an upper case letter, ( *C* - copy command file, *D* - data file, *X* - execute file);

*System-name* is the remote system;

*Grade* is a character;

*Number* is a four digit, padded sequence number.

The file

C.res45n0031

would be a *work file* for a file transfer between the local machine and the ''res45'' machine.

The scan for work is done by looking through the spool directory for *work files* (files with prefix ''C.''). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

**Call Remote System**

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* file. The information contained for each system is;

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number if field [3] is *ACU* or the device name (same as field [3]) if not *ACU,*

[6] login information (multiple fields),

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins. The *SLAVE* can also reply with a ''call-back required'' message in which case, the current conversation is terminated.

**Line Protocol Selection**

The remote system sends a message

P*proto-list*

where proto-list is a string of characters, each representing a line protocol.

The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or *N* which means there is no common protocol.

**Work Processing**

The initial roles ( *MASTER* or *SLAVE* ) for the work processing are the mode in which each program starts. (The *MASTER* has been specified by the ''−r1'' uucico option.) The *MASTER* program does a work search similar to the one used in the ''Scan For Work'' section.

There are five messages used during the work processing, each specified by the first character of the message. They are;

S send a file,
R receive a file,
C copy complete,
X execute a *uucp* command,
H hangup.

The *MASTER* will send *R*, *S* or *X* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, *XY*, *XN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a *CN* message is sent. (In the case of *CN*, the transferred file will be in the spool directory with a name beginning with ''TM'.) The requests and results are logged on both systems.

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, an *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

**Conversation Termination**

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final ''OO'' message to the other. The original *SLAVE* program will clean up and terminate. The *MASTER* will proceed to call other systems and process work as long as possible or terminate if a *−s* option was specified.

## 4. Uuxqt - Uucp Command Execution

The *uuxqt* program is used to execute *execute files* generated by *uux.* The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the spool directory for *execute files* (prefix ''X.''). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The *execute file* is described in the ''Uux'' section above.

### Command Execution

The execution is accomplished by executing a *sh −c* of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

## 5. Uulog - Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to prepend these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

−s*sys* Print entries where *sys* is the remote system name;

−u*user* Print entries for user *user.*

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

## 6. Uuclean - Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

−d*dir* The directory to be scanned is *dir*.

−m Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the uucp programs.)

−n*hours* Change the aging time from 72 hours to *hours* hours.

−p*pre* Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)

−x*num* This is the level of debugging output desired.

## 7. Security

**The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.**

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

- The login for uucp does not get a standard shell. Instead, the *uucico* program is started. Therefore, the only work that can be done is through *uucico*.
- A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call-back for certain login-ids. (See the ‘‘Files required for execution’’ section for the file description.)
- A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.
- The *uuxqt* program comes with a list of commands that it will execute. A ‘‘PATH’’ shell statement is prepended to the command line as specifed in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (Programs uucp, uucico, uux, uuxqt should be also owned by uucp and have the setuid bit set.)

## 8. Uucp Installation

There are several source modifications that may be required before the system programs are compiled. These relate to the directories used during compilation, the directories used during execution, and the local *uucp system-name.*

The four directories are:

| | |
|---|---|
| lib | (/usr/src/cmd/uucp) This directory contains the source files for generating the *uucp* system. |
| program | (/usr/lib/uucp) This is the directory used for the executable system programs and the system files. |
| spool | (/usr/spool/uucp) This is the spool directory used during *uucp* execution. |
| xqtdir | (/usr/spool/uucp/.XQTDIR) This directory is used during execution of *execute files*. |

The names given in parentheses above are the default values for the directories. The italicized named *lib, program, xqtdir,* and *spool* will be used in the following text to represent the appropriate directory names.

There are two files which may require modification, the *makefile* file and the *uucp.h* file. The following paragraphs describe the modifications. The modes of *spool* and *xqtdir* should be made ‘‘0777’’.

### Uucp.h modification

Change the *program* and the *spool* names from the default values to the directory names to be used on the local system using global edit commands.

Change the *define* value for *MYNAME* to be the local *uucp* system-name.

### makefile modification

There are several *make* variable definitions which may need modification.

| | |
|---|---|
| INSDIR | This is the *program* directory (e.g. INSDIR=/usr/lib/uucp). This parameter is used if ‘‘make cp’’ is used after the programs are compiled. |
| IOCTL | This is required to be set if an appropriate *ioctl* interface subroutine does not exist in the standard ‘‘C’’ library; the statement ‘‘IOCTL=ioctl.o’’ is required in this case. |
| PKON | The statement ‘‘PKON=pkon.o’’ is required if the packet driver is not in the kernel. |

### Compile the system The command

```
make
```

will compile the entire system. The command make cp

will copy the commands to the to the appropriate directories.

The programs *uucp*, *uux*, and *uulog* should be put in ‘‘/usr/bin’’. The programs *uuxqt*, *uucico*, and *uuclean* should be put in the *program* directory.

**Files required for execution**

There are four files which are required for execution, all of which should reside in the *program* directory. The field separator for all files is a space unless otherwise specified.

**L-devices**

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp.* The special device files are assumed to be in the */dev* directory. The format for each entry is

line call-unit speed

where;

| | |
|---|---|
| line | is the device for the line (e.g. cul0), |
| call-unit | is the automatic call unit associated with *line* (e.g. cua0), (Hardwired lines have a number ‘‘0’’ in this field.), |
| speed | is the line speed. |

The line

cul0 cua0 300

would be set up for a system which had device cul0 wired to a call-unit cua0 for use at 300 baud.

**L-dialcodes**

This file contains entries with location abbreviations used in the *L.sys* file (e.g. py, mh, boston). The entry format is

abb dial-seq

where;

| | |
|---|---|
| abb | is the abbreviation, |
| dial-seq | is the dial sequence to call that location. |

The line

py 165–

would be set up so that entry py7777 would send 165–7777 to the dial-unit.

**LOGIN/SYSTEM NAMES**

It is assumed that the *login name* used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique *system name* which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

**USERFILE**

This file contains user accessibility information. It specifies four types of constraint;

[1] which files can be accessed by a normal user of the local machine,

[2] which files can be accessed from a remote computer,

[3] which login name is used by a particular remote computer,

[4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format

login,sys **[** c **]** path-name **[** path-name **]** ...

where;

| | |
|---|---|
| login | is the login name for a user or the remote computer, |
| sys | is the system name for a remote computer, |
| c | is the optional *call-back required* flag, |
| path-name | is a path-name prefix that is acceptable for *user.* |

The constraints are implemented as follows.

[1] When the program is obeying a command stored on the local machine, *MASTER* mode, the path-names allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

[2] When the program is responding to a command from a remote machine, *SLAVE* mode, the path-names allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a *null* system name is used.

[3] When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name.

[4] If the line matched in ([3]) contains a ‘‘c’’, the remote machine is called back before any transactions take place.

The line

```
u,m  /usr/xyz
```

allows machine *m* to login with name *u* and request the transfer of files whose names start with ‘‘/usr/xyz’’.

The line

```
dan,  /usr/dan
```

allows the ordinary user *dan* to issue commands for files whose name starts with ‘‘/usr/dan’’.

The lines

```
u,m /usr/xyz  /usr/spool
u,  /usr/spool
```

allows any remote machine to login with name *u*, but if its system name is not *m*, it can only ask to transfer files whose names start with ‘‘/usr/spool’’.

The lines

```
root,  /
,  /usr
```

allows any user to transfer files beginning with ‘‘/usr’’ but the user with login *root* can transfer any file.

**L.sys**

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

**system name**

The name of the remote system.

**time**

This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800–1730).

The day portion may be a list containing some of

*Su Mo Tu We Th Fr Sa*

or it may be *Wk* for any week-day or *Any* for any day.

The time should be a range of times (e.g. 0800–1230). If no time portion is specified, any time of day is assumed to be ok for the call.

**device**

This is either *ACU* or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

**speed**

This is the line speed for the call (e.g. 300).

**phone**

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. mh5900, boston995–9980).

For the hardwired devices, this field contains the same string as used for the *device* field.

**login**

The login information is given as a series of fields and subfields in the format

expect send **[** expect send **]** ...

where; *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect**[**–send–expect**]**...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a BREAK character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the L.sys file would be

sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm looks at the last part of the string as illustrated in the password field.

## 9. Administration

This section indicates some events and files which must be administered for the *uucp* system. Some administration can be accomplished by *shell files* which can be initiated by *crontab* entries. Others will require manual intervention. Some sample *shell files* are given toward the end of this section.

**SQFILE – sequence check file**

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

**TM – temporary data files**

These files are created in the *spool* directory while files are being copied from a remote machine. Their names have the form

> **TM**.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

> uuclean –pTM

will remove all *TM* files older than three days.

**LOG – log entry files**

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE.* The command

> uulog

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new LOG entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

**STST – system status files**

These files are created in the spool directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a *TALKING* status when to machines are conversing. The form of the file name is

> **STST**.sys

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a *TALKING* status. In this case, the file must be removed before a conversation is attempted.

**LCK – lock files**

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

> **LCK..**str

where *str* is either a device or system name. The files may be left in the spool directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

**Shell Files**

The *uucp* program will spool work and attempt to start the *uucico* program, but the starting of *uucico* will sometimes fail. (No devices available, login failures etc.). Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a ''shell'' file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

> *program*/uulog
> *program*/uucico –r1

Note that the ''–r1'' option is required to start the *uucico* program in *MASTER* mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C.* or *D.* files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

> *program*/uuclean –pTM –pC. –pD.
> *program*/uuclean –pST –pLCK –n12

can be used. Note the ''–n12'' option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the ''–n'' option will use a three day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILE*s. A shell like

> cp *spool*/LOGFILE *spool*/o.LOGFILE
> rm *spool*/LOGFILE

can be used.

**Login Entry**

One or more logins should be set up for *uucp*. Each of the ''/etc/passwd'' entries should have the ''*program*/uucico'' as the shell to be executed. The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the *shell* argument limits the login to the use of uucp ( *uucico* ) only.

**File Modes**

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp*, *uux*, *uucico* and *uuxqt* should be owned by the *uucp* login with the ''setuid'' bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard *shell* for the *uucp* logins.

The *L.sys*, *SQFILE* and the *USERFILE* which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

# On the Security of UNIX

*Dennis M. Ritchie*

Bell Laboratories
Murray Hill, New Jersey 07974

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX† system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls— there may be bugs in this area, but none are known— but rather in lack of checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
while : ; do
        mkdir x
        cd x
done
```

Either a panic will occur because all the i-nodes on the device are used up, or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

In this version of the system, users are prevented from creating more than a set number of processes simultaneously, so unless users are in collusion it is unlikely that any one can stop the system altogether. However, creation of 20 or so CPU or disk-bound jobs leaves few resources available for others. Also, if many large jobs are run simultaneously, swap space may run out, causing a panic.

It should be evident that excessive consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group identification number (UID and GID). Nine of the protection bits are used to specify independently permission to read, to write, and to execute the file to the user himself, to members of the user's group, and to all other users. Each process generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared

†UNIX is a Trademark of Bell Laboratories.

against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file but ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a ''super user'' who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Traditionally, UNIX software has been exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. In the current version, this policy may be easily adjusted to suit the needs of the installation or the individual user. Associated with each process and its descendants is a mask, which is in effect *and*-ed with the mode of every file and directory created by that process. In this way, users can arrange that, by default, all their files are no more accessible than they wish. The standard mask, set by *login,* allows all permissions to the user himself and to his group, but disallows writing by others.

To maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's files inaccessible to others. The lack of sufficiency could follow from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below). For greater protection, an encryption scheme is available. Since the editor is able to create encrypted documents, and the *crypt* command can be used to pipe such documents into the other text-processing programs, the length of time during which cleartext versions need be available is strictly limited. The encryption scheme used is not one of the strongest known, but it is judged adequate, in the sense that cryptanalysis is likely to require considerably more effort than more direct methods of reading the encrypted files. For example, a user who stores data that he regards as truly secret should be aware that he is implicitly trusting the system administrator not to install a version of the crypt command that stores every typed password in a file.

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. In the current version, it is based on a slightly defective version of the Federal DES; it is purposely defective so that easily-available hardware is useless for attempts at exhaustive key-search. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is still feasible up to a point. We have observed that users choose passwords that are easy to guess: they are short, or from a limited alphabet, or in a dictionary. Passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out ''login:'' on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user *(su)* command is writable, anyone can copy the shell onto it and get a password-free version of *su.* A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. To take an obsolete example, the previous version of the *mail* command was set-UID and owned by the super-user. This version sent mail to the recipient's own directory. The notion was that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble was that *mail* was rather dumb: anyone could mail someone else's private file to himself. Much more serious is the following scenario: make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ''.mail'' to the password file in some writable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp/.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and tapes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su;* other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.

# Password Security: A Case History

*Robert Morris*

*Ken Thompson*

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

April 3, 1978

# Password Security: A Case History

*Robert Morris*

*Ken Thompson*

Bell Laboratories
Murray Hill, New Jersey 07974

## INTRODUCTION

Password security on the UNIX† time-sharing system [1] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e. prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so called ''super-user'' password, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual unauthorized access and accidental disclosure.

## PROLOGUE

The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons.

†UNIX is a Trademark of Bell Laboratories.

The technique is excessively vulnerable to lapses in security. Temporary loss of protection can occur when the password file is being edited or otherwise modified. There is no way to prevent the making of copies by privileged users. Experience with several earlier remote-access systems showed that such lapses occur with frightening frequency. Perhaps the most memorable such occasion occurred in the early 60's when a system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

Once such a lapse in security has been discovered, everyone's password must be changed, usually simultaneously, at a considerable administrative cost. This is not a great matter, but far more serious is the high probability of such lapses going unnoticed by the system administrators.

Security against unauthorized disclosure of the passwords was, in the last analysis, impossible with this system because, for example, if the contents of the file system are put on to magnetic tape for backup, as they must be, then anyone who has physical access to the tape can read anything on it with no restriction.

Many programs must get information of various kinds about the users of the system, and these programs in general should have no special permission to read the password file. The information which should have been in the password file actually was distributed (or replicated) into a number of files, all of which had to be updated whenever a user was added to or dropped from the system.

**THE FIRST SCHEME**

The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted. Such a scheme was first described in [3, p.91ff.]. It also seemed advisable to devise a system in which neither the password file nor the password program itself needed to be protected against being read by anyone.

All that was needed to implement these ideas was to find a means of encryption that was very difficult to invert, even when the encryption program is available. Most of the standard encryption methods used (in the past) for encryption of messages are rather easy to invert. A convenient and rather good encryption program happened to exist on the system at the time; it simulated the M-209 cipher machine [4] used by the U.S. Army during World War II. It turned out that the M-209 program was usable, but with a given key, the ciphers produced by this program are trivial to invert. It is a much more difficult matter to find out the key given the cleartext input and the enciphered output of the program. Therefore, the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key. The encrypted result was entered into the password file.

**ATTACKS ON THE FIRST APPROACH**

Suppose that the bad guy has available the text of the password encryption program and the complete password file. Suppose also that he has substantial computing capacity at his disposal.

One obvious approach to penetrating the password mechanism is to attempt to find a general method of inverting the encryption algorithm. Very possibly this can be done, but few successful results have come to light, despite substantial efforts extending over a period of more than five years. The results have not proved to be very useful in penetrating systems.

Another approach to penetration is simply to keep trying potential passwords until one succeeds; this is a general cryptanalytic approach called *key search.* Human beings being what they are, there is a strong tendency for people to choose relatively short and simple passwords that they can remember. Given free choice, most people will choose their passwords from a restricted character set (e.g. all lower-case letters), and will often choose words or names. This human habit makes the key search job a great deal easier.

The critical factor involved in key search is the amount of time needed to encrypt a potential password and to check the result against an entry in the password file. The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed. It is takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations.

If we want to check all passwords of length *n* that consist entirely of lower-case letters, the number of such passwords is $26^n$. If we suppose that the password consists of printable characters only, then the number of possible passwords is somewhat less than $95^n$. (The standard system ''character erase'' and ''line kill'' characters are, for example, not prime candidates.) We can immediately estimate the running time of a program that will test every password of a given length with all of its characters chosen from some set of characters. The following table gives estimates of the running time required on a PDP-11/70 to test all possible character strings of length *n* chosen from various sets of characters: namely, all lower-case letters, all lower-case letters plus digits, all alphanumeric characters, all 95 printable ASCII characters, and finally all 128 ASCII characters.

| n | 26 lower-case letters | 36 lower-case letters and digits | 62 alphanumeric characters | 95 printable characters | all 128 ASCII characters |
|---|---|---|---|---|---|
| 1 | 30 msec. | 40 msec. | 80 msec. | 120 msec. | 160 msec. |
| 2 | 800 msec. | 2 sec. | 5 sec. | 11 sec. | 20 sec. |
| 3 | 22 sec. | 58 sec. | 5 min. | 17 min. | 43 min. |
| 4 | 10 min. | 35 min. | 5 hrs. | 28 hrs. | 93 hrs. |
| 5 | 4 hrs. | 21 hrs. | 318 hrs. | | |
| 6 | 107 hrs. | | | | |

One has to conclude that it is no great matter for someone with access to a PDP-11 to test all lower-case alphabetic strings up to length five and, given access to the machine for, say, several weekends, to test all such strings up to six characters in length. By using such a program against a collection of actual encrypted passwords, a substantial fraction of all the passwords will be found.

Another profitable approach for the bad guy is to use the word list from a dictionary or to use a list of names. For example, a large commercial dictionary contains typicallly about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some ''good'' things to try are:

- The dictionary with the words spelled backwards.
- A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.
- The above with initial upper-case letters.
- All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
- Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time;

15 were a single ASCII character;

72 were strings of two ASCII characters;

464 were strings of three ASCII characters;

477 were string of four alphamerics;

706 were five letters, all upper-case or all lower-case;

605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831, or 86% of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

**AN ANECDOTE**

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudo-random number generator with only $2^{15}$ starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only $2^{15}$ of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time.

**IMPROVEMENTS TO THE FIRST APPROACH**

**1. Slower Encryption**

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

**2. Less Predictable Passwords**

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters, then the program asks him to enter a longer password. This further reduces the efficacy of key search.

These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

**3. Salted Passwords**

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 ($2^{12}$). The reason for this is that there are 4096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

### 4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be unthinkable.

### 5. A Subtle Point

To login successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in a invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.

## CONCLUSIONS

On the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

It is also worth some effort to conceal even the encrypted passwords. Some UNIX systems have instituted what is called an ‘‘external security code’’ that must be typed when dialing into the system, but before logging in. If this code is changed periodically, then someone with an old password will likely be prevented from using it.

Whenever any security procedure is instituted that attempts to deny access to unauthorized persons, it is wise to keep a record of both successful and unsuccessful attempts to get at the secured resource. Just as an out-of-hours visitor to a computer center normally must not only identify himself, but a record is usually also kept of his entry. Just so, it is a wise precaution to make and keep a record of all attempts to log into a remote-access time-sharing system, and certainly all unsuccessful attempts.

Bad guys fall on a spectrum whose one end is someone with ordinary access to a system and whose goal is to find out a particular password (usually that of the super-user) and, at the other end, someone who wishes to collect as much password information as possible from as many systems as possible. Most of the work reported here serves to frustrate the latter type; our experience indicates that the former type of bad guy never was very successful.

We recognize that a time-sharing system must operate in a hostile environment. We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

**References**

[1] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* **17** (July 1974), pp. 365-375.

[2] *Proposed Federal Information Processing Data Encryption Standard.* Federal Register (40FR12134), March 17, 1975

[3] Wilkes, M. V. *Time-Sharing Computer Systems.* American Elsevier, New York, (1968).

[4] U. S. Patent Number 2,089,603.