PLEXUS

Plexus Sys3 UNIX Programmer's Manual -- vol 2B

98-05037.3 June 20, 1983

Plexus Sys3 UNIX Programmer's Manual -- vol 2B

98-05037.3 June 20, 1983

PLEXUS COMPUTERS INC

2230 Martin Ave

Santa Clara, CA 95050

408/988-1755

Plexus Sys3 UNIX Programmer's Manual -- vol 2B

PREFACE

This manual contains a collection of documents that describe specific aspects of the UNIX* operating system. These include descriptions of programming, language, administrative and maintenance tools.

Additional documents describing the operating system, document preparation tools and programming and language tools are collected in the _Plexus Sys3 UNIX Programmer's Manual -- vol 2A_.

Both these volumes (2A and 2B) should be used as supplementary documents for the _Plexus Sys3 UNIX Programmer's Manual -- vol 1A_ and _Plexus Sys3 UNIX Programmer's Manual -- vol 1B_, the basic reference manual for the operating system.

Comments

Please address all comments concerning this manual to:

Plexus Computers Inc
Technical Publications Dept
2230 Martin Ave
Santa Clara, CA 95050
408/988-1755

Revision History

The second edition (#98-05037.2) contains new front matter.

This edition (#98-05037.3) contains a new VPM document.

---

* UNIX is a trademark of Bell Laboratories. Plexus Computers Inc is licensed to distribute UNIX under the authority of AT&T.

Calculators

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

†UNIX is a Trademark of Bell Laboratories.

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

†UNIX is a Trademark of Bell Laboratories.

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions

```
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 (**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 (**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
while (relation) {
        statement
        last-expression
}
```

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.

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

†UNIX is a Trademark of Bell Laboratories.

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.

†UNIX is a Trademark of Bell Laboratories.

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.

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

$$x_{n+1} = \frac{1}{2}(x_n + \frac{y}{x_n})$$

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,

```
[lipl+  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).

Fortran

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. Assem-

†UNIX is a Trademark of Bell Laboratories.

	bler output for a source file **x.f**, **x.e**, **x.r**, or **x.c** is put on file **x.s**.
−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**. **includes** 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 **opened** 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 saved 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 $m \leqslant n$. 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 **parameters** 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 **elseifs** 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 $1 \leqslant k \leqslant n$, 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

This paper is a revised and expanded version of oe 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.

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 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, multilevel **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.

Graphics

PWB/Graphics Overview

A. R. Feuer

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

PWB/Graphics, or just *graphics*, is the name given to a growing collection of numerical and graphical commands available as part of the Programmer's Workbench [1]. In its initial release, *graphics* includes commands to construct and edit numerical data plots and hierarchy charts. This memorandum will help you get started using *graphics* and show you where to find more information. The examples below assume that you are familiar with the UNIX™ Shell [1].

2. BASIC CONCEPTS

The basic approach taken in *graphics* is to generate a drawing by describing it rather than by drafting it. Any drawing is seen as having two fundamental attributes: its underlying logic and its visual layout. The layout encompasses one representation of the logic. For example, consider the attributes of a drawing that consists of a plot of the function $y=x^2$ for x between 0 and 10. The logic of the plot is the description as just given, viz. $y=x^2, 0 \leqslant x \leqslant 10$. The layout consists of an x-y grid, axes labeled perhaps 0 to 10 and 0 to 100, and lines drawn connecting the x-y pairs 0,0 to 1,1 to 2,4 and so on.

The way to generate a picture in *graphics* is

gather data | transform the data | generate a layout | display the layout.

To generate the specific plot of $y=x^2, 0 \leqslant x \leqslant 10$ and display it on a Tektronix* display terminal would be:

```
gas −s0,t10 | af "x^2" | plot | td
```

gas	generates sequences of numbers, in this case starting at 0 and terminating at 10.
af	performs general arithmetic transformations.
plot	builds x-y plots.
td	displays drawings on Tektronix terminals.

The resulting drawing is shown in Figure 1.

The layout generated by a *graphics* program may not always be precisely what is wanted. There are two ways to influence the layout. Each drawing program accepts options to direct certain layout features. For instance, in the previous example we may have wanted the x-axis labels to indicate each of the numbers plotted and we might not have wanted any y-axis labels at all. To achieve this the *plot* command would be changed to:

```
plot −xi1,ya
```

producing the drawing of Figure 2.

The output from any drawing command can also be affected by editing it directly at a display terminal using the graphical editor, *ged*. To edit a drawing really means to edit the computer representation of the drawing. In the case of *graphics* the representation is called a graphical primitive string, or GPS. All of the drawing commands (e.g., *plot*) write GPS and all of the device filters (e.g., *td*) read GPS. *Ged* allows you to manipulate GPS at a display terminal by interacting with the drawing the GPS describes.

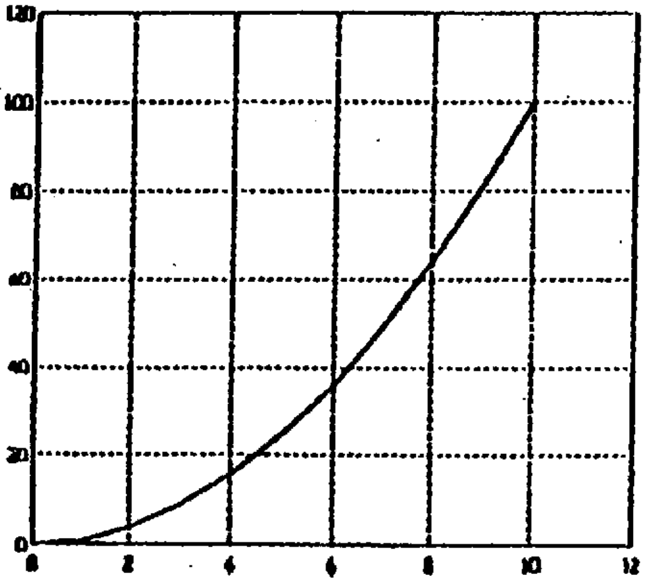


Figure 1. gas -s0,t10 | af "x^2" | plot | td

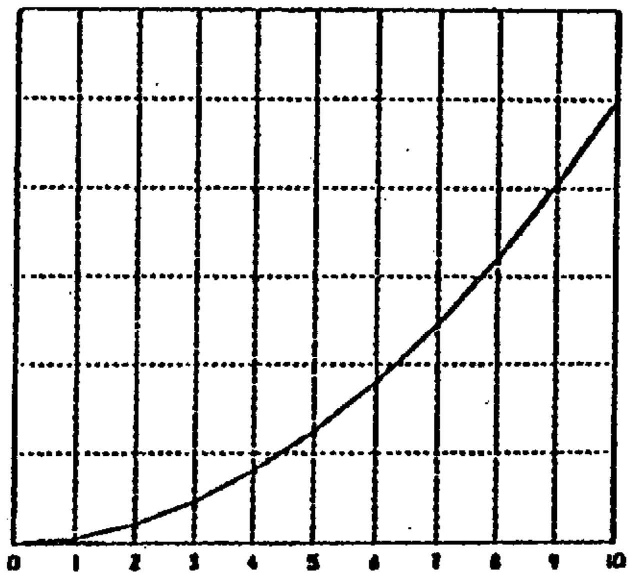


Figure 2. gas -s0,t10 | af "x^2" | plot -xi1,ya | td

GPS describes graphical objects drawn within a Cartesian plane 65,534 units on each axis. The plane, known as the *universe*, is partitioned into 25 equal sized square regions. Multi-drawing displays can be produced by placing drawings into adjacent regions and then displaying each region.

3. GETTING STARTED

To access the *graphics* commands when logged in on a PWB/UNIX system type **graphics**. Your *Shell* variable PATH will be altered to include the *graphics* commands and the *Shell* primary prompt will be changed to ^. Any command accessible before typing **graphics** will still be accessible; *graphics* only adds commands, it doesn't take any away. Once in *graphics*, you can find out about any of the *graphics* commands using *whatis*. Typing **whatis** by itself on a command line will generate a list of all the commands in *graphics* along with instructions on how to find out more about any of them.

All of the *graphics* commands accept the same command line format:

A command is:	a *command-name* followed by *argument*(s).
A *command-name* is:	the name of any of the *graphics* commands.
An *argument* is:	a *file-name* or an *option-string*.
A *file-name* is:	any file name not beginning with −, or a − by itself to reference the standard input.
An *option-string* is:	a − followed by *option*(s).
An *option* is:	letter(s) followed by an optional value. Options may be separated by commas.

You will get the best results with *graphics* commands if you use a display terminal. *Plot*(1) filters can be used in conjunction with *gtop* (see *gutil*(1)) to get somewhat degraded drawings on Versatec printers and Dasi-type terminals. And since GPS can be stored in a file, it can be created from any terminal for later displaying on a graphical device.

To remove the *graphics* commands from your PATH *Shell* variable type EOT (control-d on most terminals). To logoff UNIX from *graphics* type **quit**.

4. EXAMPLES OF WHAT YOU CAN DO

4.1 Numerical Manipulation and Plotting

Stat(1) describes a collection of numerical commands. All of these commands operate on vectors. A vector is a text file that contains numbers separated by delimiters, where a delimiter is anything that is not a number. For example,

1 2 3 4 5, and
arf tty47 Mar 5 09:52

are both vectors. (The latter being the vector: 47 5 9 52.)

Here is an easy way to generate a Celsius-Fahrenheit conversion table using *gas* to generate the vector of Celsius values:

```
gas −s0,t100,i10 | af "C,9/5*C+32"
```

The output is:

```
0.0        32
10         50
20         68
30         86
40         104
50         122
60         140
70         158
80         176
90         194
100        212
```

This is what is going on:

gas −s0,t100,i10	We have seen *gas* in an earlier example. In this case the sequence starts at 0, terminates at 100, and the increment between successive elements is 10.
af "C,9/5*C+32"	We have also seen *af*. Arguments to *af* are expressions. Operands in an expression are either constants or filenames. If a filename is given that does not exist in the current directory it is taken as the name for the standard input. In this example C references the standard input. The output is a vector with odd elements coming from the standard input and even elements being a function of the preceding odd element.

Here is an example that illustrates the use of vector titles and multiline plots:

```
gas | title −v"first ten integers" >N
root N >RN
root −r3 N >R3N
root −r1.5 N >R1.5N
plot −FN,g N R1.5N RN R3N | td
```

The resulting plot is shown in Figure 3.

title −v"*name*"	*Title* associates a *name* with a vector. In this case, first ten integers is associated with the vector output by *gas*. The vector is stored in file N.
root −r*n*	*Root* outputs the *n*th root of each element on the input. If −r*n* is not given then the square root is output. Also, if the input is a titled vector the title will be transformed to reflect the root function.
plot −F*X*,g *Y(s)*	This command generates a multiline plot with *Y(s)* plotted versus *X*. The g option causes tick marks to appear instead of grid lines.

The next example generates a histogram of random numbers.

```
rand −n100 | title −v"100 random numbers" | qsort | bucket | hist | td
```

The output is shown in Figure 4.

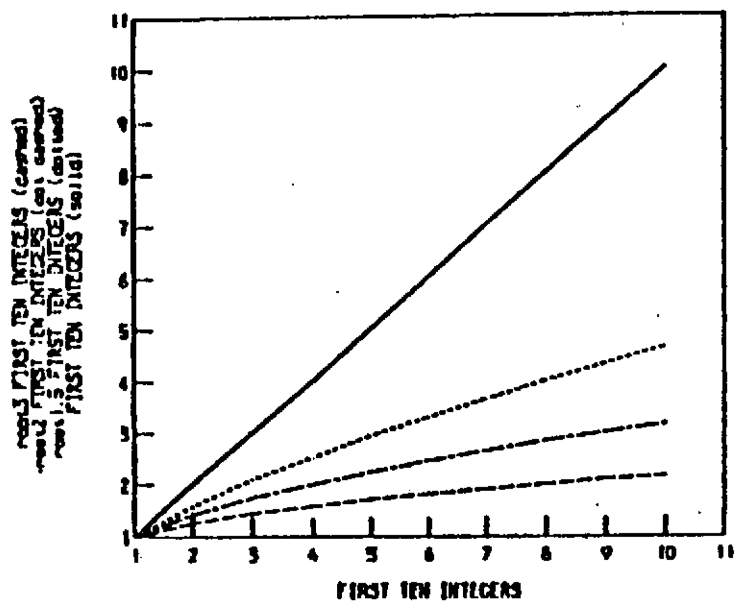


Figure 3. Some roots of the first ten integers

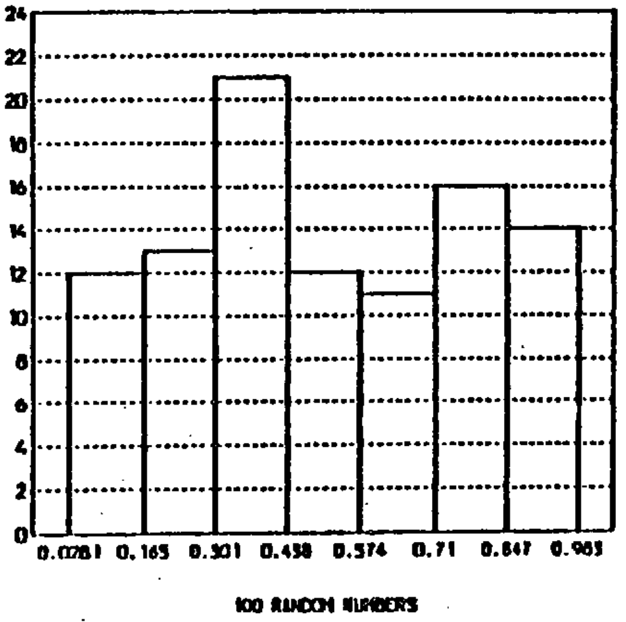


Figure 4. Histogram of 100 random numbers

rand −n100	*Rand* outputs random numbers using *rand*(3C). In this case 100 numbers are output in the range 0 to 1.
qsort	*Qsort* sorts the elements of a vector in ascending order.
bucket	*Bucket* breaks the range of a vector into intervals and counts how many elements from the vector fall into each interval. The output is a vector with odd elements being the interval boundaries and even elements being the counts.
hist	*Hist* builds a histogram based on interval boundaries and counts.

4.2 Drawings Built from Boxes

There is a large class of drawings composed from boxes and text. Examples are structure charts, configuration drawings, and flow diagrams. In *graphics* the general procedure to construct such box drawings is the same as that for numerical plotting. Namely gather and transform the data, build and display the layout.

As an example, consider hierarchy charts. The command line

```
dtoc | vtoc | td
```

outputs the drawing shown in Figure 5.

Dtoc outputs a table of contents that describes a directory structure (Figure 5a). The fields from left to right are level number, directory name, and the number of ordinary readable files contained in the directory. *Vtoc* reads a (textual) table of contents and outputs a visual table of contents, or hierarchy chart. Input to *vtoc* consists of a sequence of entries, each describing a box to be drawn. An entry consists of a level number, an optional style field, a text string to be placed in the box, and a mark field to appear above the top right hand corner of the box.

5. WHERE TO GO FROM HERE

The best way to learn about *graphics* is to log onto a PWB/UNIX system and use it. Tutorials exist for *stat*(1) and *ged*(1). [2] contains administrative information for *graphics*. Reference information can be found in the PWB/UNIX User's Manual under the following manual pages:

ged(1), the graphical editor;
gps(5), a description of a graphical primitive string;
graphics(1), the entry point for *graphics*;
gutil(1), a collection of utility commands;
stat(1), numerical manipulation and plotting commands;
tek4000(1), a collection of commands to manipulate Tektronix 4000 series terminals; and
toc(1), routines to build tables of contents.

6. REFERENCES

[1] *PWB/UNIX User's Manual* – Release 2.0., Bell Laboratories, 1979.

[2] R. L. Chen and D. E. Pinkston, *Administrative Information for PWB/Graphics*, Bell Laboratories Memorandum, 1979.

Figure 5. Directory Structure for Graphics

```
0.      "source"       2
1.      "glib.d"       1
1.1.    "gpl.d"        12
1.2.    "gsl.d"        14
2.      "gutil.d"      6
2.1.    "cvrtopt.d"    7
2.2.    "gtop.d"       8
2.3.    "ptog.d"       5
3.      "stat.d"       54
4.      "tek4000.d"    5
4.1.    "ged.d"        37
4.4.    "td.d"         8
5.      "toc.d"        3
5.1.    "ttoc.d"       3
5.2.    "vtoc.d"       22
6.      "whatis.d"     108
```

Figure 5a. Dtoc output

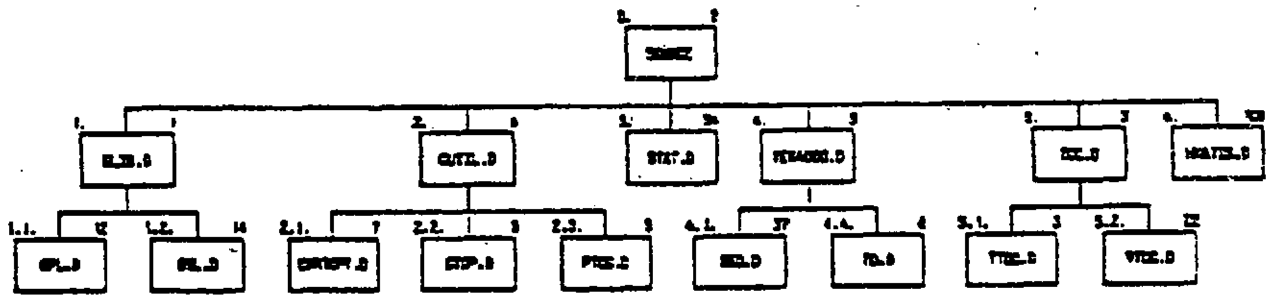


Figure 5b. Vtoc output

Administrative Information For PWB/Graphics

Ruth L. Chen
Diane E. Pinkston

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

This document is a reference guide for system administrators who are using or establishing a PWB/Graphics facility [1] on UNIX™. It contains information about directory structure, installation, makefiles, hardware requirements, and miscellaneous facilities of PWB/Graphics.

2. PWB/Graphics STRUCTURE

Figure 1 contains a graphical representation of the directory structure of PWB/Graphics. In this paper, the *Shell* variable $SRC will represent the parent node for graphics source. On PWB/UNIX $SRC is /usr/src/cmd. If PWB/Graphics is copied onto other systems, $SRC could have other values but should, in general, be the same as on PWB/UNIX.

The *graphics* command (see *graphics*(1)) resides in /usr/bin. All other PWB/Graphics executables are located in /usr/bin/graf. /usr/lib/graf contains text for whatis documentation (see *gutil*(1)) and editor scripts for *ttoc* (see *toc*(1)).

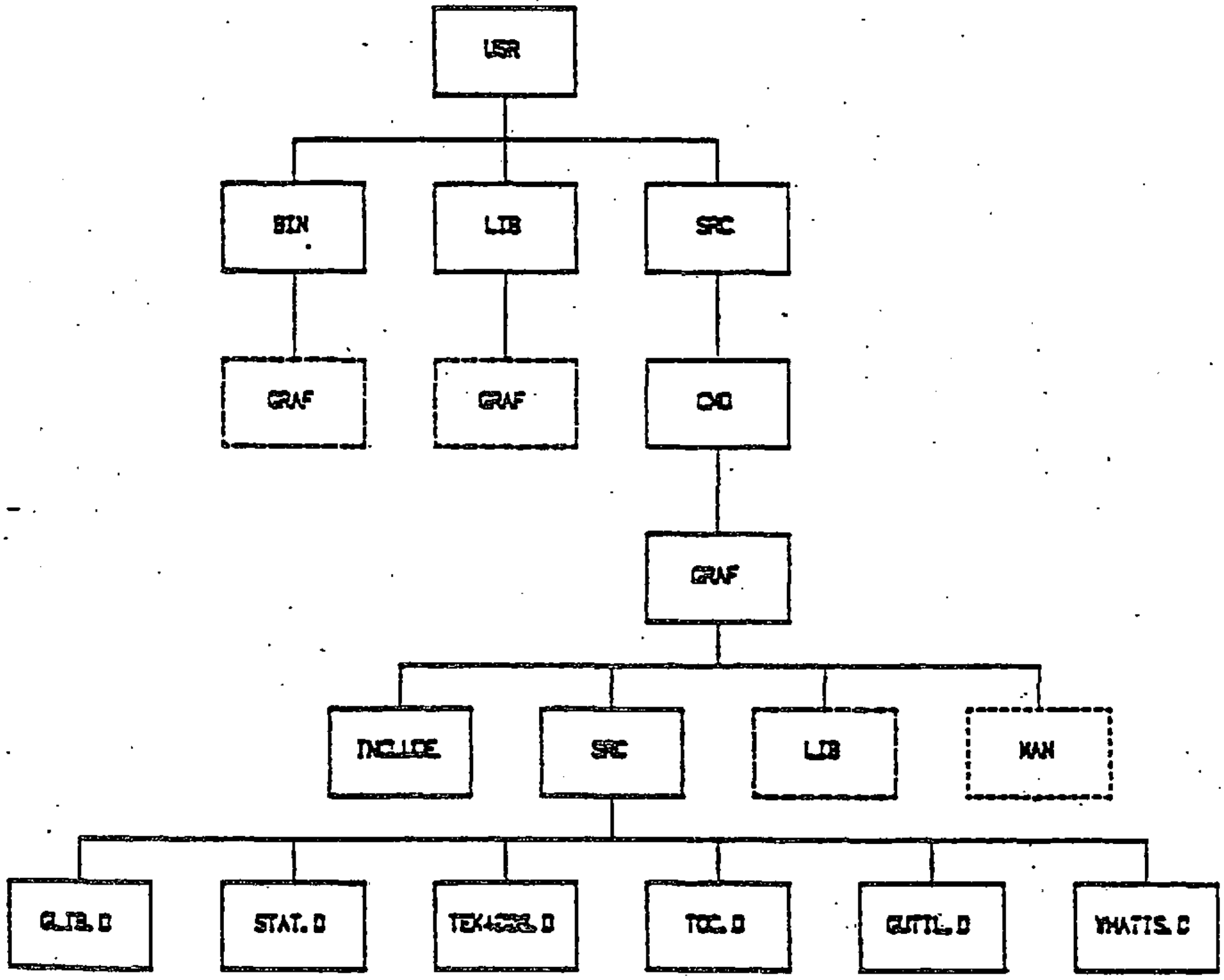
PWB/Graphics source resides below the directory $SRC/graf. $SRC/graf is broken into the following subdiretories:

- include - contains the following header files: debug.h, errpr.h, gsl.h, gpl.h, setopt.h, and util.h.
- src - contains source code partitioned into subdirectories by subsystem. Each subdirectory contains its own Makefile (or Install file for whatis.d).
  - glib.d - contains source used to build the graphical subroutine library in $SRC/graf/lib/glib.a.
  - stat.d - contains source for numerical analysis and plotting routines.
  - tek4000.d - contains source for *ged* (the graphical editor), *td* (a Tektronix display function), and other Tektronix dependent routines.
  - gutil.d - contains source for utility programs.
  - toc.d - contains source for table of contents drawing routines.
  - whatis.d - contains mm files and the install routine for quick-reference documentation.
- lib - contains glib.a which contains commonly used graphical subroutines.
- man - Figure 1 shows $SRC/graf/man as a dotted box because this directory does not exist on PWB/UNIX systems where all manual pages reside in /usr/man. $SRC/graf/man is created if PWB/Graphics is copied onto another system (see section 3.), and will contain the following manual page files: graphics.1, gutil.1, stat.1, tek4000.1, toc.1, ged.1 and gps.5.

3. INSTALLING PWB/Graphics

Procedures for installing PWB/Graphics:

Fig. 1 PWB/Graphics Structure



1. PWB/UNIX systems,
   - To build the entire Graphics system (i.e. all boxes except man in Figure 1), execute (as superuser)

     ```
     ./:mkcmd graf
     ```

     ./:mkcmd resides in /usr/src, and all manual pages exist in /usr/man.
   - To build a particular subsystem, execute

     ```
     ./:mkcmd graf subsystem
     ```
   - To build a particular *command* within a subsystem, execute

     ```
     ./:mkcmd graf subsystem command-name
     ```
2. UNIX/TS systems not running PWB,
   - See appendix for tape copying procedures.
   - Build $SRC/graf/lib and PWB/Graphics executables (dashed boxes in Figure 1) by typing:

     ```
     make -f $SRC/graf/graf.mk
     ```
   - To make a particular graphics subsystem use the Makefile in $SRC/graf/src, e.g.

     ```
     cd $SRC/graf/src
     make subsystem
     ```
   - Note, there is a name conflict between PWB/Graphics *plot* and UNIX/TS *plot*(1). The recommended fix is to remove /usr/bin/plot and move the *plot*(1) filters from /usr/lib to /usr/bin.

A *subsystem* is either *glib*, *stat*, *tek4000*, *toc*, *gutil* or *whatis*. *Glib* must exist before other subsystems can be built. Write permission in /usr/bin and /usr/lib is needed, and the following libraries are assumed to exist:

/lib/libc.a	Standard C library, used by all subsystems.
/lib/libm.a	Math library, used by all subsystems.
/usr/lib/macros/[nt]pwbmm.m*	Programmer's Workbench memorandum macros for [*nt*]*roff*, used by the whatis subsystem.

The build process takes approximately one hour of system time. If the make must be stopped, it is a good idea to rebuild from the top. Upon completion, the following things will be created and owned by bin.

/usr/lib/graf	A directory for data and editor scripts.
/usr/bin/graf	A directory for executables.
/usr/bin/graphics	Command entry point for PWB/Graphics.

Makefiles use executable *Shell* procedures *cco* and *cca*. *Cco* is used to compile C source and install load modules in /usr/bin/graf. The *cca* command compiles C programs and loads object code into archive files.

Whatis.d contains source files for *whatis* and the executable command *Install*.

```
Install command-name
```

calls *nroff* to produce whatis documentation for *command-name* in /usr/lib/graf. To install the entire whatis subsystem, use the Makefile in $SRC/graf/src.

3.1 Makefile Parameters

Makefiles use various macro parameters, some of which can be specified on the command line to redirect outputs or inputs. Parameters specified in higher level Makefiles are passed to lower levels. Below is a list of specifiable parameters for Makefiles followed by their default values in parenthesis and an explanation of their usage.

1. $SRC/graf/graf.mk

BIN1 (/usr/bin)	installation directory for the *graphics* command.
BIN2 (/usr/bin/graf)	installation directory for other graphic commands.
SRC (/usr/src/cmd)	parent directory for source code.

2. $SRC/graf/src/Makefile

BIN1 (/usr/bin)	installation directory for the *graphics* command.
BIN2 (/usr/bin/graf)	installation directory for other graphic commands.
LIB (/usr/lib/graf)	installation directory for whatis documentation.

3. $SRC/graf/src/stat.d/Makefile

BIN (../../bin)	installation directory for executable commands.

4. $SRC/graf/src/toc.d/Makefile

BIN (../../bin)	installation directory for executable commands.

5. $SRC/graf/src/tek4000.d/Makefile

BIN (../../bin)	installation directory for executable commands.

6. $SRC/graf/src/gutil.d/Makefile

BIN (../../bin)	installation directory for executable commands.

The following example will make a new version of the graphical editor, *ged*, in /a1/pmt/dp/bin:

```
cd $SRC/graf/src/tek4000.d
make BIN=/a1/pmt/dp/bin ged
```

4. TEKTRONIX TERMINAL

The PWB/Graphics display function *td* and the graphical editor *ged* both use Tektronix Series 4010 storage tubes. Below is a list of device considerations necessary for PWB/Graphics operation.

4.1 Getty Table Entry

When a Tektronix 4010 series terminal is connected via a dedicated line to UNIX, an entry in the system table (in /usr/src/cmd/getty.c) is suggested, to store terminal status information. This table entry appears as follows on PWB/UNIX:

```
/* table '6' -- 4800/9600 -- tektronix 4014 */
        '6', 7,
        ANYP+ RAW+ FF1, ANYP+ ECHO+ CRMOD+ FF1,
        B4800, B4800,
        "\033\014\000login: ",

        7, '6',
        ANYP+ RAW+ FF1, ANYP+ ECHO+ CRMOD+ FF1,
        B9600, B9600,
        "\033\014\000login: ",
```

but on other systems it may have to be created and then referenced in /etc/inittab. Standard parity and a form-feed delay are necessary. The form-feed delay gives the screen time to clear without losing information. Below is an example of the terminal status as printed by *stty*:

```
speed 4800 baud
erase = '#'; kill = '@'
even odd -- nl echo -- tabs ff1
```

4.2 Strap Options

The standard strap options as listed below should be used (see the Reference Manual for the Tektronix 4014 [3]):

1. LF effect - LF causes line-feed only.
2. CR effect - CR caused carriage return only.
3. Del implies loy - Del key is interpreted as low-order y value.
4. Graphics Input terminators - None.

4.3 Enhanced Graphics Module

The Enhanced Graphics Module of Tektronix terminals is required for PWB/Graphics. The EGM provides different line styles (solid, dotted, dot-dashed, dashed, and long-dashed), right and left margin cursor location, and 12-bit cursor addressing (4096 by 4096 screen points).

5. MISCELLANEOUS INFORMATION

5.1 Announcements

The *graphics* command provides a means of printing out announcements to users. To set up an anouncement facility, create a readable text file containing the announcements named **announce**. Also in /usr/bin/graphics redefine the *Shell* variable **$GRAF** to be the directory pathname of the announce file.

5.2 Uselog

The *graphics* command also provides a means of monitoring its use by listing users in a file. To set up a uselog facility create a writeable file named **.uselog** (in the same directory as announce if announcements are being used) and redefine the *Shell* variable **$GRAF** within /usr/bin/graphics to specify the directory location. Each time a user executes *graphics*, an entry of the login name, terminal number, and system date are recorded in **.uselog**.

5.3 Restricted Environments

Restricted environments can be used to limit user access to the system (see *rsh*(1) [4]). In a restricted environment, commands in /rbin and /usr/rbin are executed before those in /bin and /usr/bin. The commands *ed*, *mv*, *rm*, and *sh* require restricted interface programs which do not allow users to move or remove files that begin with dot (.)[2].

Creating restricted environments for graphics:

1. Create a restricted *ged* command in /usr/rbin as follows:

   ```
   exec /usr/bin/graf/ged −R
   ```

2. Create restricted logins for users or create a community login with a working directory (reached through .profile) set up for each user. A restricted login specifies /bin/rsh as the terminal interface program and is created by adding /bin/rsh to the end of the /etc/passwd file entry for that login.
3. Call graphics −r from .profile.

The execution of **graphics −r** changes $PATH to look for commands in /rbin and /usr/rbin and executes a restricted *Shell*. The −R option is appended to the *ged* command so that the escape from *ged* to UNIX (*!command*) will also use a restricted *Shell*.

ACKNOWLEDGEMENTS

We wish to thank Alan R. Feuer for his valuable contributions, suggestions, and careful reading of this document. We also thank M. J. Petrella for his help in supplying information concerning the PWB/UNIX environment.

REFERENCES

[1] Feuer, A. R. *PWB/Graphics Overview*. Bell Laboratories, 1979.

[2] Petrella, M. J. *Restricted Access to PWB/UNIX* − DRAFT. Bell laboratories, May 1979.

[3] Tektronix. *User's Manual for 4014 and 4014-1 Display Terminal*. July, 1974.

[4] *PWB/UNIX User's Manual* − Release 2.0.

APPENDIX

Procedures for tape copying (as superuser)

– Locate graphics source by changing directory to $SRC, the parent directory.

– Then copy source and manual pages from the tape by typing

```
cpio −idm < /dev/mt4        (creates graf)
ed graf
cpio −idm < /dev/mt0        (creates man)
```

This will result in the directory structure indicated by the solid boxes plus $SRC/graf/man in Figure 1. Necessary sub-directories will be created (see *cpio*(1)).

January 1980

A Tutorial Introduction to the Graphical Editor

Alan R. Feuer

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

Ged is an interactive graphical editor used to display, edit, and construct drawings on Tektronix® 4010 series display terminals. The drawings are represented as a sequence of objects in a token language known as GPS (for graphical primitive string). GPS is produced by the drawing commands in PWB/Graphics [1] such as *vtoc* and *plot* as well as by *ged* itself.

The examples in this tutorial illustrate how to construct and edit simple drawings. Try them to become familiar with how the editor works, but keep in mind that *ged* is intended primarily to edit the output of other programs rather than to construct drawings from scratch. A summary of editor commands and options is given in Section 3.

As for notation, literal keystrokes are printed in **boldface**. Meta-characters are also in boldface and are surrounded by angled brackets. For example, **<return>** means return and **<sp>** means space. In the examples, output from the terminal is printed in normalface type. Inline comments are in normalface and are surrounded by parentheses.

2. COMMANDS

To start we will assume that you have successfully entered the graphics environment (as described in *graphics*(1) of [2]) while logged in at a display terminal. To enter *ged* type

ged <return>

After a moment the screen should be clear save for the *ged* prompt, •, in the upper left corner. The • tells you that *ged* is ready to accept a command.

Each command passes through a sequence of stages during which you describe what the command is to do. All commands pass through a subset of these stages:

1. *command line*
2. *text*
3. *points*
4. *pivot*
5. *destination*

As a rule, each stage is terminated by typing **<return>** . The **<return>** for the last stage of a command triggers execution.

2.1 The Command Line

The simplest commands consist only of a *command line*. The *command line* is modeled after a conventional command line in the *Shell*. That is

command-name [−*option*(s)] [*filename*] **<return>**

? is an example of a simple command. It lists the commands and options understood by *ged*. Type

•**? <return>** (you type a question mark followed by a return)

to generate the list.

A command is executed by typing the first character of its name. *Ged* will echo the full name and wait for the rest of the *command line.* For example, e references the *erase* command. As *erase* consists only of stage 1, typing <return> causes the erase action to occur. Typing <rubout> after a command name and before the final <return> for the command aborts the command. Thus while

    •erase <return>

erases the display screen,

    •erase <rubout>

brings the editor back to •.

Following the command-name, *options* may be entered. Options control such things as the width and style of lines to be drawn or the size and orientation of text. Most options have a default value that applies if a value for the option is not specified on the command line. The *set* command allows you to examine and modify the default values. Type

    •set <return>

to see the current default values.

The value of an option is either of type integer, character, or Boolean. Boolean values are represented by + for true and − for false. A default value is modified by providing it as an option to the *set* command. For example, to change the default text height to 300 units type:

    •set −h300 <return>

Arguments on the command line, but not the command-name, may be edited using the erase and kill characters from the *Shell.* (Actually, this applies whenever text is being entered.)

2.2 Constructing Graphical Objects

Drawings are stored as GPS in a *display buffer* internal to the editor. Typically, a drawing in *ged* is composed of instances of three graphical primitives: *arc, lines,* and *text.*

2.2.1 Generating text. To put a line of text on the display screen use the *Text* command. First enter the *command line* (stage 1):

    •Text <return>

Next enter the *text* (stage 2):

    a line of text <return>.

And then enter the starting *point* for the text (stage 3):

    <position cursor> <return>

Positioning of the graphic cursor is done either with the thumbwheel knobs on the terminal keyboard or with an auxiliary joystick. The <return> establishes the location of the cursor to be the starting point for the text string. The *Text* command ends at stage 3, so this <return> initiates the drawing of the text string.

Text accepts options to vary the angle, height, and line width of the characters, and to either center or right justify the text object. The text string may span more than one line by escaping the <return> (i.e., \<return>) to indicate continuation. To illustrate some of these capabilities, try the following:

•Text −r <return>	(right justify text)
top\<return>	
right <return>	
<position cursor> <return>	
•Text −a90 <return>	(rotate text 90 degrees)
lower\<return>	
left <return>	
<position cursor> <return>	(pick a point below and left of the previous point)

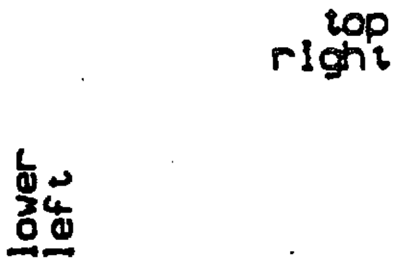


Figure 1. Generating text objects

2.2.2 Drawing lines. The *Lines* command is used to construct objects built from a sequence of straight lines. It consists of stages 1 and 3. Stage 1 is straightforward:

•Lines *possible options* **<return>**

Lines accepts options to specify line style and line width.

Stage 3, the entering of *points*, is more interesting. *Points* are referenced either with the graphic cursor or by name. We have already entered a point with the cursor for the *Text* command. For *Lines* it is more of the same. As an example, let us build a triangle:

•Lines <return>	
<position cursor> <sp>	(locate the first point)
<position cursor> <sp>	(the second point)
<position cursor> <sp>	(the third point)
<position cursor> <sp>	(back to the first point)
<return>	(terminate *points*, draw triangle)

Typing **<sp>** enters the location of the crosshairs as a point. *Ged* identifies the point with an integer and adds the location to the current *point set*. The last point entered can be erased by typing **#** . The current point set can be cleared by typing **@** . On receiving the final **<return>** the points are connected in numerical order.

2.2.2.1 Accessing points by name. The points in the current point set may be referenced by name using the **$** operator. **$***n* references the point numbered *n*. Using **$** we can redraw the triangle of Section 2.2.2 by entering:

•Lines <return>	
<position cursor> <sp>	
<position cursor> <sp>	
<position cursor> <sp>	
$0 <return>	(reference point 0)
<return>	

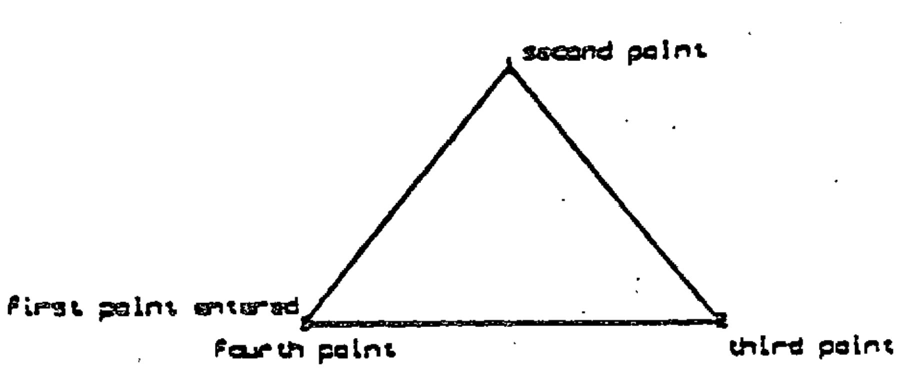


Figure 2. Building a triangle

At the start of each command that includes stage 3, *points*, the current point set is empty. The point set from the previous command is saved and is accessible using the . operator. . swaps the points in the previous point set with those in the current set. The = operator can be used to identify the current points. To illustrate, let us use the triangle just entered as the basis for drawing a quadrilateral:

•Lines <return>	
.	(access the previous point set)
=	(identify the current points)
#	(erase the last point)
<position cursor> <sp>	(add a new point)
$0 <return>	(close the figure)
<return>	

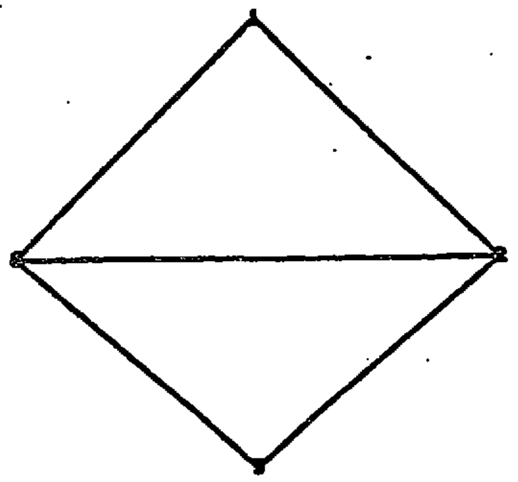


Figure 3. Accessing the previous point set

Individual points from the previous point set can be referenced by using the . operator with $.
We will build a triangle that shares an edge with the quadrilateral:

```
•Lines <return>
$.1 <return>              (reference point 1 from the previous point set)
$.2 <return>              (reference point 2)
<sp>                      (enter a new point)
$0 <return>               (or $.1, to close the figure)
<return>
```

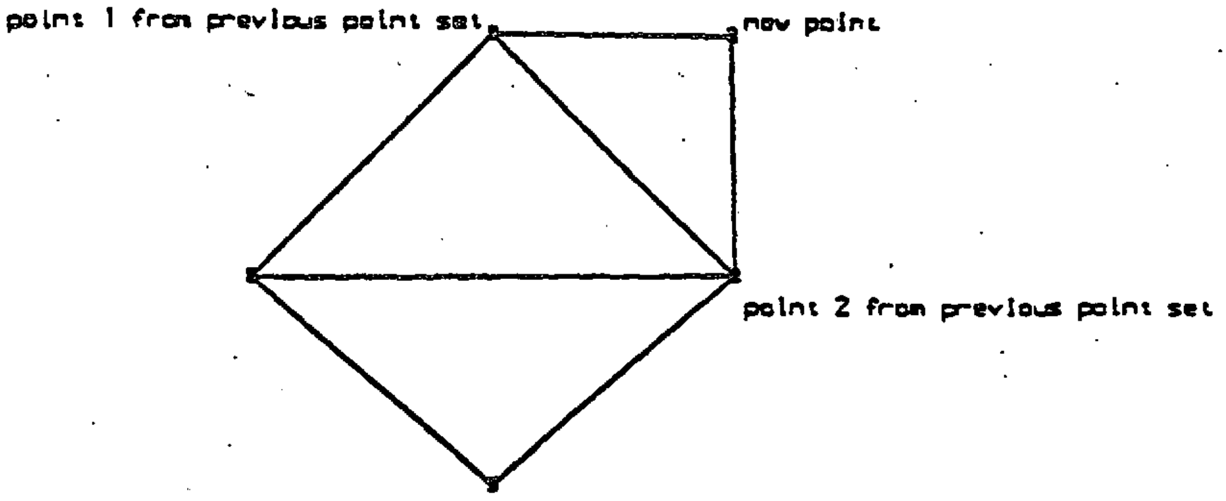

point 1 from previous point set
new point
point 2 from previous point set

Figure 4. Referencing points from the previous point set

A point can also be given a name. The > operator allows you to associate an upper case letter with a point just entered. A simple example is:

```
*Lines <return>
<position cursor> <sp>          (enter a point)
>A                              (name the point A)
<position cursor> <sp>
<return>
```

In commands that follow you can now reference point A using the $ operator, as in:

```
*Lines <return>
$A
<position cursor> <sp>
<return>
```

2.2.3 Drawing curves. Curves are interpolated from a sequence of three or more points. The *Arc* command generates a circular arc given three points on a circle. The arc is drawn starting at the first point, through the second point, and ending at the third point. A circle is an arc with the first and third points coincident. One way to draw a circle is thus:

```
*Arc <return>
<position cursor> <sp>
<position cursor> <sp>
$0 <return>
<return>
```

2.3 Editing Objects

2.3.1 Addressing objects. An object is addressed by pointing to one of its *handles.* All objects have an *object-handle.* Usually the object-handle is the first point entered when the object was created. The *objects* command marks the location of each object-handle with an O. Type

```
*objects -v <return>
```

to see the handles of all the objects on the screen.

Some objects, *Lines* for example, also have *point-handles.* Typically each of the points entered when an object is constructed becomes a point-handle. (Yes, an object-handle is also a point-handle.) The *points* command marks each of the point-handles.

A handle is pointed to by including it within a *defined-area.* A defined-area is generated either with a command line option or interactively using the graphic cursor. As an example, try deleting one of the objects you have created on the screen.

```
*Delete <return>
<position cursor> <sp>          (above and to the left of some object-handle)
<position cursor> <sp>          (below and to the right of the object-handle)
<return>                        (the defined-area should include the object-handle)
<return>                        (if all is well, delete the object)
```

The defined-area is outlined with dotted lines. The reason for the seemingly extra <return> at the end of the *Delete* command is to give you an opportunity to stop the command (using <rubout>) if the defined-area is not quite right. Every command that accepts a defined-area will wait for a confirming <return> . Use the *new* command to get a fresh copy of the remaining objects.

Notice that defined-areas are entered as *points* in the same way that objects are created. Actually, a defined-area may be generated by giving anywhere from zero to 30 points. Inputting zero points is particularly useful to point to a single handle. It creates a small defined-area about the location of the terminating <return> . Using a zero point defined-area, the *Delete* command would be:

```
•Delete <return>
<position cursor>                   (center the crosshairs on the object-handle)
<return>                            (terminate the defined-area)
<return>                            (delete the object)
```

A defined-area can also be given as a command line option. For example, to delete everything in the display buffer give the universe option to the *Delete* command. Note the difference between the commands Delete −u and erase.

2.3.2 Changing the location of an object. Objects are moved using the *Move* command. Create a circle using *Arc*, then move it as follows:

```
•Move <return>
<position cursor> <return>          (centered on the object-handle)
<return>                            (this establishes a pivot, marked with an asterisk)
<position cursor> <return>          (this establishes a destination)
```

The basic move operation relocates every point in each object addressed by the distance from the *pivot* to the *destination*. In this case we chose the pivot to be the object-handle, so effectively we moved the object-handle to the destination point.

2.3.3 Changing the shape of an object. The *Box* command is a special case of generating lines. Given two points it creates a rectangle such that the two points are at opposite corners. The sides of the rectangle lie parallel to the edges of the screen. Draw a box:

```
•Box <return>
<position cursor> <sp>
<position cursor> <return>
```

Box generates point-handles at each vertex of the rectangle. Use the *points* command to mark the point-handles. The shape of an object can be altered by moving point-handles. The next example illustrates one way to double the height of a box.

```
•Move −p+ <return>
<position cursor> <sp>              (left of the box, between the top and bottom edges)
<position cursor> <return>          (right of the box, below the bottom edge)
<position cursor> <return>          (on the top edge)
<position cursor> <return>          (directly below on the bottom edge)
```

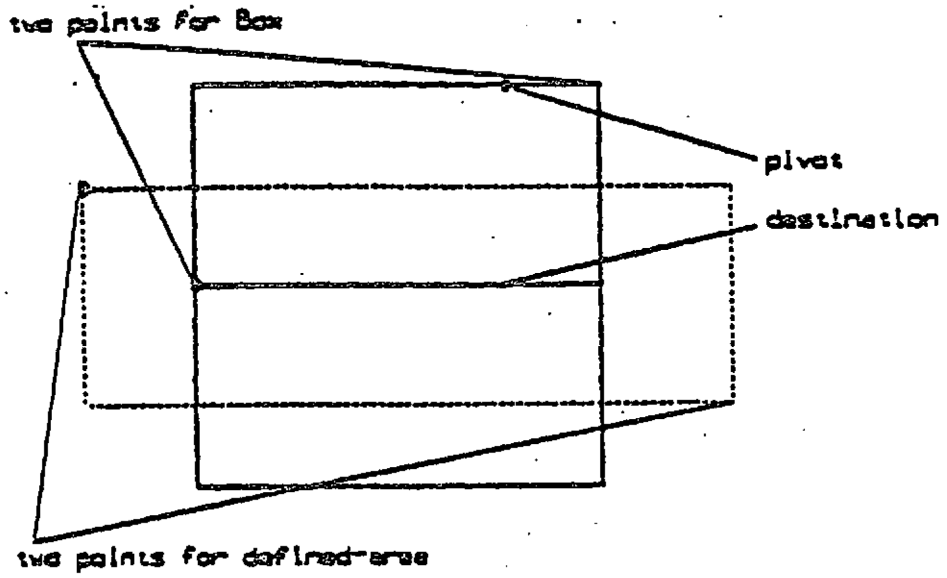


Figure 5. Growing a box

Since the points flag is true, the operation is applied to each point-handle addressed. In this case each point-handle within the defined-area is moved the distance from the pivot to the destination. If p were false only the object-handle would have been addressed.

2.3.4 Changing the size of an object. The size of an object can be changed using the *Scale* command. *Scale* scales objects by changing the distance from each handle of the object to a pivot by a factor. Put a line of text on the screen and try the following *Scale* commands:

```
•Scale -f200 <return>                  (factor is in percent)
<position cursor> <return>             (point to object-handle)
<position cursor> <return>             (set pivot to rightmost character)
<return>

•Scale -f50 <return>
. <return>                             (reference the previous defined-area)
<position cursor> <return>             (set pivot above a character near the middle)
<return>
```

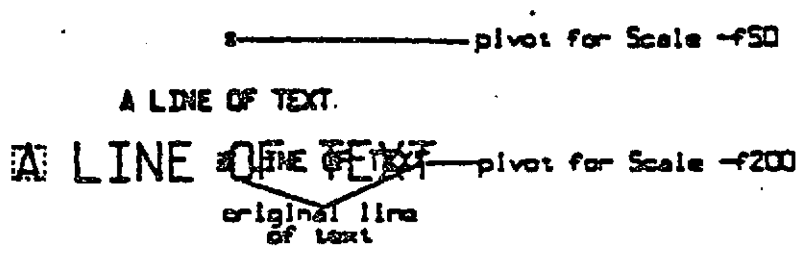


Figure 6. Scaling text

A useful insight into the behavior of scaling is to note that the position of the pivot does not change. Also observe that the defined-area is scaled to preserve its relationship to the graphical objects.

The size of objects can also be changed by moving point-handles. Generate a circle, this time using the *Circle* command:

•Circle <return>	
<position cursor> <sp>	(specify the center)
<position cursor> <return>	(specify a point on the circle)

Circle generates an arc with the first and third point at the point specified on the circle. The second point of the arc is located 180° around the circle. One way to change the size of the circle is to move one of the point-handles (using Move −p).

The size of text characters can be changed via a third mechanism. Character height is a property of a line of text. The *Edit* command allows you to change character height as follows:

•Edit −h*height* <return>	(*height* is in universe units, see Section 2.4)
<position cursor> <return>	(point to the object-handle)
<return>	

2.3.5 Changing the orientation of an object. The orientation of an object can be altered using *Rotate. Rotate* rotates each point of an object about a pivot by an angle. Try the following rotations on a line of text:

•Rotate −a90 <return>	(angle is in degrees)
<position cursor> <return>	(point to object-handle)
<position cursor> <return>	(set pivot to rightmost character)
<return>	

•Rotate −a−90 <return>	
. <return>	(reference previous defined-area)
<position cursor> <return>	(set pivot to a character near the middle)
<return>	

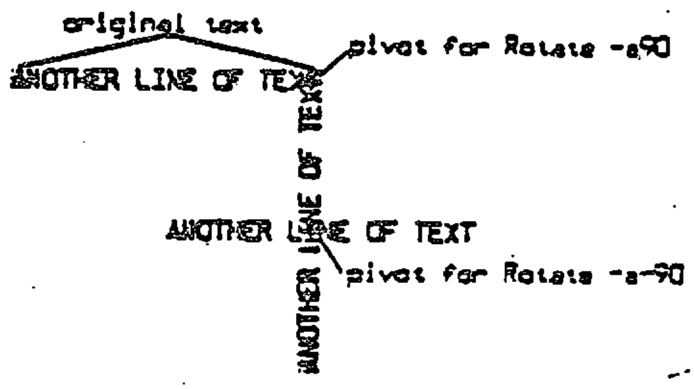


Figure 7. Rotating text

2.3.6 Changing the style or width of lines. In the current editor objects can be drawn from lines in any of five styles (solid, dashed, dot-dashed, dotted, long-dashed) and three widths (narrow, medium, bold). Style is controlled by the s option, width by w.

```
•Lines -wn,sdo <return>
<position cursor> <sp>
<position cursor> <sp>
<return>
```

creates a narrow width dotted line.

```
•Edit -wb,sdd <return>
<position cursor> <return>          (point to object-handle of the line)
<return>
```

changes the line to bold dot-dashed.

2.4 View Commands

All of the objects we have drawn lie within a Cartesian plane, 65,534 units on each axis, known as the *universe.* Thus far we have displayed only a small portion of the universe on the display screen. The command

```
•view -u <return>
```

displays the entire universe.

2.4.1 Windowing. A mapping of a portion of the universe onto the display screen is called a *window.* The extent or magnification of a window is altered using the *zoom* command. To build a window that includes all of the objects you have drawn type

```
•zoom <return>
<position cursor> <sp>              (above and to the left of any object)
<position cursor> <return>          (below and to the right, also end points)
<return>                            (verify)
```

Zooming can be either *in* or *out.* Zooming in, as with a camera lens, increases the magnification of the window. The area outlined by *points* is expanded to fill the screen. Zooming out decreases magnification. The current window is shrunk so that it fits within the defined-area. The direction of the zoom is controlled by the sense of the out flag; o true means zoom out.

The location of a window is altered using *view*. *View* moves the window so that a given point in the universe lies at a given location on the screen.

```
•view <return>
<position cursor> <return>        (locate a point in the universe)
<position cursor> <return>        (locate a point on the screen)
```

View also provides access to several predefined windows. We have already seen view −u. view −h displays the *home-window* . The home-window is the window that circumscribes all of the objects in the universe. The result is similar to that of the example using *zoom* given earlier.

Lastly, using *view* you may select to window on a particular *region*. The universe is partitioned into 25 equal sized regions. Regions are numbered from 1 to 25 beginning at the lower left and proceeding toward the upper right. Region 13, the center of the universe, is used as the default region by drawing commands such as *plot* and *vtoc* (see [1]).

2.5 Other Commands

2.5.1 Interacting with files. To save the contents of the display buffer copy it to a file using the *write* command:

```
•write filename <return>
```

The contents of *filename* will be a GPS, thus it can be displayed using any of the device filters (e.g., *td* [1]) or read back into *ged*.

A GPS is read into the editor using the *read* command:

```
•read filename <return>
```

The GPS from *filename* is appended to the display buffer and then displayed. Because *read* does not change the current window only some or none of the objects read may be visible. A useful command sequence to view everything read is

```
•read −e− filename <return>
•view −h <return>
```

The display function of *read* is inhibited by setting the echo flag to false. view −h windows on and displays the full display buffer.

The *read* command may also be used to input text files. The form is:

```
read [−option(s)] filename <return>
```

followed by a single point to locate the first line of text. A text object is created for each line of text from *filename*. Options to *read* are the same as those for the *Text* command.

2.5.2 Leaving the editor. Use the *quit* command to terminate an editing session. As with the text editor *ed*, *quit* responds with ? if the internal buffer has been modified since the last *write*. A second *quit* forces exit.

2.6 Other Useful Things to Know.

2.6.1 One line UNIX escape. As in *ed*, ! provides a temporary escape to the *Shell*.

2.6.2 Typing ahead. Most programs under UNIX allow you to type input before the program is ready to receive it. In general this is not the case with *ged*; characters typed before the appropriate prompt are lost.

2.6.3 Speeding things up. Displaying the contents of the display buffer can be time consuming, particularly if much text is involved. The wise use of two flags to control what gets displayed can make life more pleasant: the echo flag controls echoing of new additions to the display buffer; the text flag controls whether text will be outlined or drawn.

3. COMMAND SUMMARY

In the summary, characters actually typed are printed in boldface. Command stages are printed in italics. Arguments surrounded by brackets are optional. Parentheses surrounding arguments separated by "or" means that exactly one of the arguments must be given. For example, the *Delete* command (Section 3.2) accepts the arguments **−universe**, **−view**, and *points*.

3.1 Construct commands:

Arc	[**−echo,style,width**] *points*
Box	[**−echo,style,width**] *points*
Circle	[**−echo,style,width**] *points*
Hardware	[**−echo**] *text points*
Lines	[**−echo,style,width**] *points*
Text	[**−angle,echo,height,midpoint,rightpoint,text,width**] *text points*

3.2 Edit commands:

Delete	(**−** (**universe** or **view**) or *points*)
Edit	[**−angle,echo,height,style,width**] (**−** (**universe** or **view**) or *points*)
Kopy	[**−echo,points,x**] *points pivot destination*
Move	[**−echo,points,x**] *points pivot destination*
Rotate	[**−angle,echo,kopy,x**] *points pivot destination*
Scale	[**−echo,factor,kopy,x**] *points pivot destination*

3.3 View commands:

coordinates	*points*
erase	
new	
objects	(**−** (**universe** or **view**) or *points*)
points	(**−** (**labelled-points** or **universe** or **view**) or *points*)
view	(**−** (**home** or **universe** or **region**) or [**−x**] *pivot destination*)
x	[**−view**] *points*
zoom	[**−out**] *points*

3.4 Other commands:

quit	
read	[**−angle,echo,height,midpoint,rightpoint,text,width**] *filename* [*destination*]
set	[**−angle,echo,factor,height,kopy,midpoint,points,rightpoint,style,text,width,x**]
write	*filename*

!*command*

?

3.5 Options:

Options specify parameters used to construct, edit, and view graphical objects. If a parameter used by a command is not specifed as an *option*, the default value for the parameter will be used. The format of command *options* is

−*option* [,*option*]

where *option* is *keyletter*[*value*]. Flags take on the *values* of true or false indicated by + and − respectively. If no *value* is given with a flag, true is assumed.

Object options:

angle*n*	Specify an angle of *n* degrees.
echo	When true, changes to the display buffer will be echoed on the screen.
factor*n*	Specify a scale factor of *n* percent.
height*n*	Specify height of *text* to be *n* universe-units ($0 \leq n < 1280$).
kopy	The commands *Scale* and *Rotate* can be used to either create new objects or to alter old ones. When the kopy flag is true, new objects are created.
midpoint	When true, use the midpoint of a text string to locate the string.
out	When true, reduce magnification during *zoom*.
points	When true, operate on points otherwise operate on objects.
rightpoint	When true, use the rightmost point of a text string to locate the string.
style*type*	Specify line style to be one of following *types*:

so	solid
da	dashed
dd	dot-dashed
do	dotted
ld	long-dashed

text	Most text is drawn as a sequence of lines. This can sometimes be painfully slow. When the text flag is false, *text* strings are outlined rather than drawn.
width*type*	Specify line width to be one of following *types*:

n	narrow
m	medium
b	bold

x	One way to find the center of a rectangular area is to draw the diagonals of the rectangle. When the x flag is true, defined-areas are drawn with their diagonals.

Area options:

home	Reference the home-window.
region*n*	Reference region *n*.
universe	Reference the universe-window.
view	Reference those objects currently in view.

4. ACKNOWLEDGEMENTS

Ged borrows freely from the ideas and code of the *gex* program by D. J. Jackowski. The first version of *ged* was written by D. E. Pinkston.

5. REFERENCES

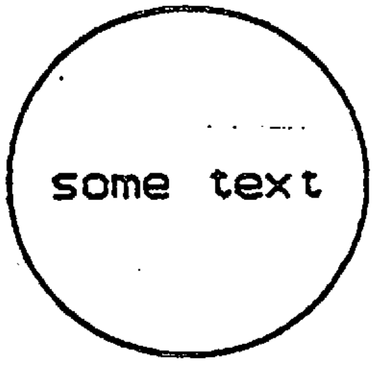
[1] Feuer, A. R.; "PWB/Graphics Overview"; TM 79-3782-1, June 11, 1979.

[2] *PWB/UNIX User's Manual* Release 2.0, Bell Laboratories, 1979.

APPENDIX: SOME EXAMPLES OF WHAT CAN BE DONE

1. Text Centered Within a Circle

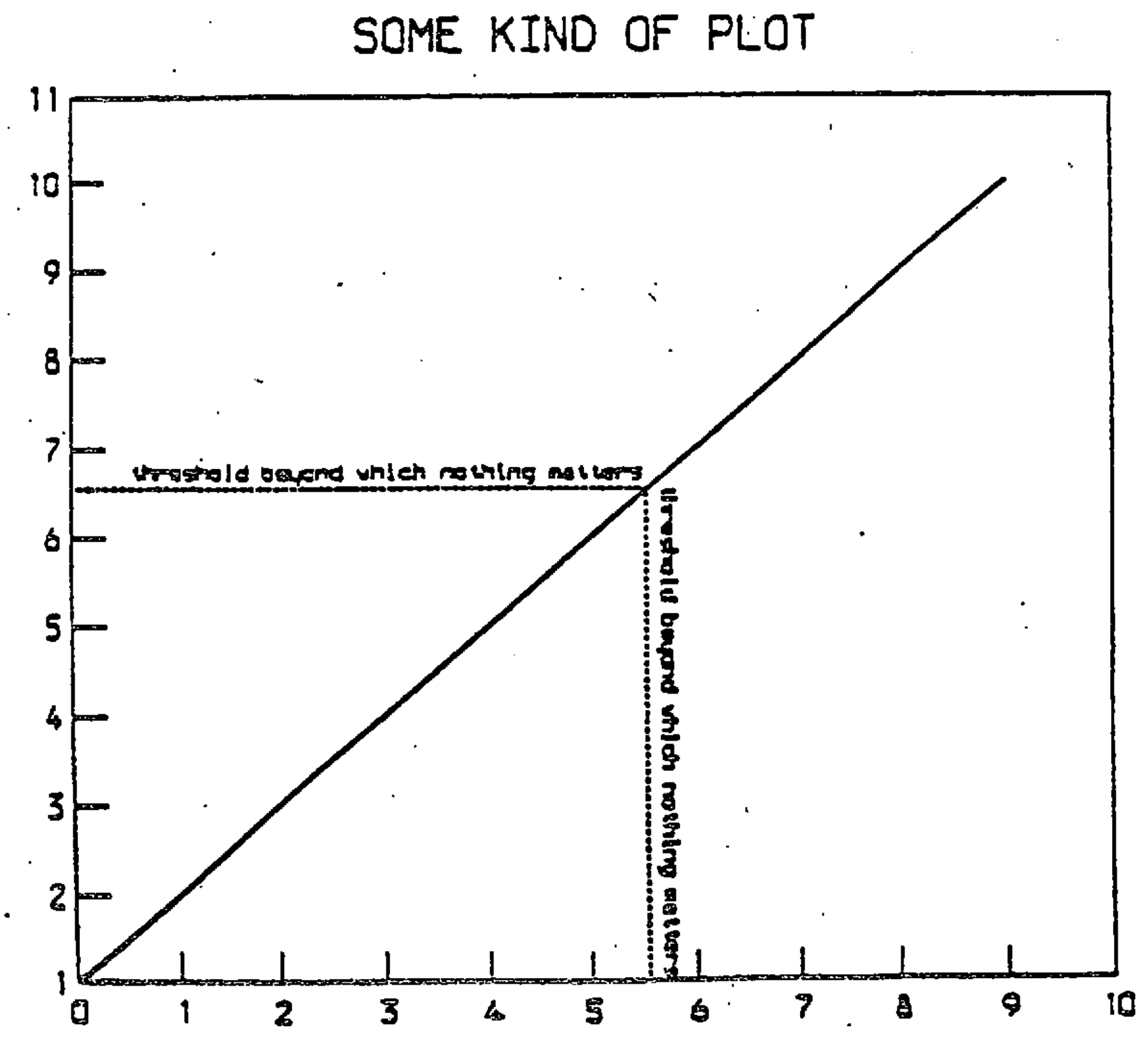
•Circle <cr>	
<position cursor> <sp>	(establish center)
<position cursor> <cr>	(establish radius)
•Text −m <cr>	(text is to be centered)
some text <cr>	
$.0 <cr>	(first point from previous set, i.e., circle center)
<cr>	



2. Making Notes on a Plot

```
•! gas | plot −g >A <cr>           (generate a plot, put it in file A)

•read −e− A <cr>                    (input the plot, but do not display it)
•view −h <cr>                       (window on the plot)
•Lines −sdo <cr>                    (draw dotted lines)
<position cursor> <sp>
<position cursor> <sp>
<position cursor> <sp>
<cr>                                (end of Lines)
•set −h150,wn <cr>                  (set text height to 150, line width to narrow)
•Text −r <cr>                       (right justify text)
threshold beyond which nothing matters <cr>
<position cursor> <cr>              (set right point of text)
•Text −a−90 <cr>                    (rotate text negative 90 degrees)
threshold beyond which nothing matters <cr>
<position cursor> <cr>              (set top end of text)
•x <cr>                             (find center of plot)
<position cursor> <sp>              (top left of plot)
<position cursor> <cr>              (bottom right)
•Text −h300,wm,m <cr>               (build title: height 300, weight medium, centered)
SOME KIND OF PLOT <cr>
<position cursor> <cr>              (set title centered above plot)
```

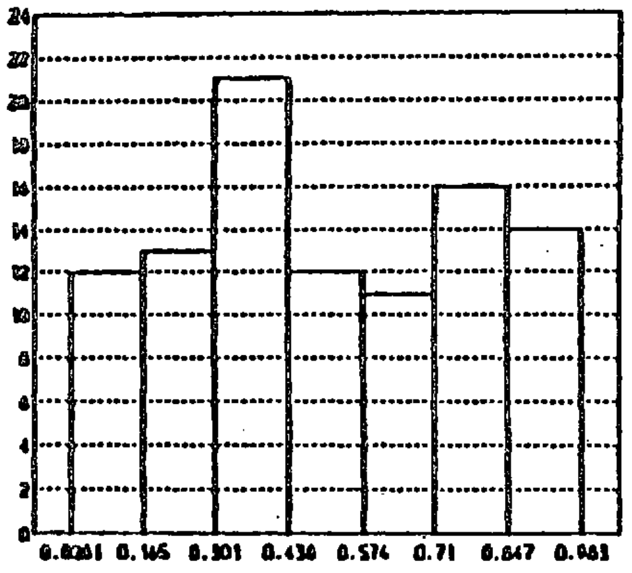


3. A Page Layout with Drawings and Text

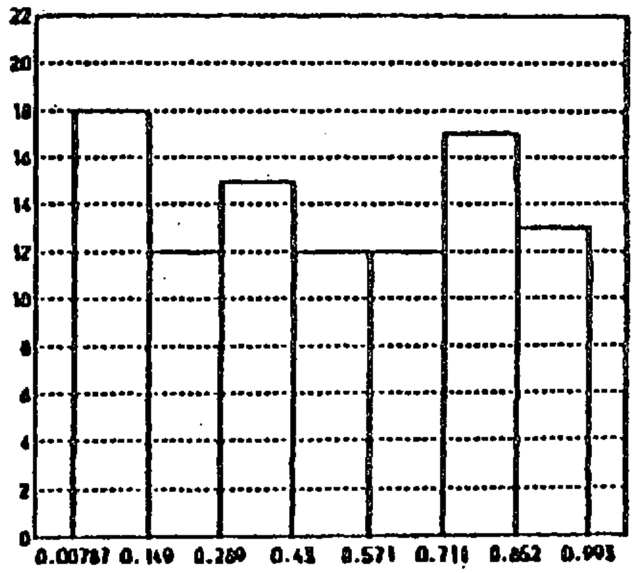
```
*! rand -s1,n100 | title -v"seed 1" | qsort | bucket | hist -r12 >A <cr>
        (put a histogram, region 12, of 100 random numbers into file A)
*! rand -s2,n100 | title -v"seed 2" | qsort | bucket | hist -r13 >B <cr>
        (put another histogram, region 13, into file B)
*! ed <cr>                       (create a file of text using the text editor)
a <cr>
On this page are two histograms <cr>
from a series of 40 <cr>
designed to illustrate the weakness <cr>
of multiplicative congruential random number generators.  <cr>
.pl \n(nlu <cr>                  (mark end of page)
. <cr>
w C <cr>                         (put the text into file C)
156
q <cr>
*! nroff C | yoo C <cr>          (format C, leave the output in C)

*view -u <cr>                    (window on the universe)
*read A <cr>
*read B <cr>
*read -h300,wn,m C <cr>          (text height 300, line weight narrow, text centered)
<position cursor> <cr>           (center text over two plots)
*view -h <cr>                    (window on the resultant drawing)
```

On this page are two histograms from a series of 40 designed to illustrate the weakness of multiplicative congruential random number generators.



SEED 1



SEED 2

LEX

Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt
Bell Laboratories
Murray Hill, New Jersey 07974

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 be used to 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. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

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

$$\begin{tikzcd}
\text{Source} \arrow[r] & \boxed{\text{Lex}} \arrow[r] & \text{yylex} \\
\text{Input} \arrow[r] & \boxed{\text{yylex}} \arrow[r] & \text{Output}
\end{tikzcd}$$

An overview of Lex

Figure 1

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 there are only two host languages, C[1] and Fortran (in the form of the Ratfor language[2]). Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

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

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

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

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

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

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

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

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time

$$\begin{tikzcd}
 & \text{lexical rules} \arrow[d] & \text{grammar rules} \arrow[d] & \\
 & \boxed{\text{Lex}} \arrow[d] & \boxed{\text{Yacc}} \arrow[d] & \\
\text{Input} \arrow[r] & \boxed{\text{yylex}} \arrow[r] & \boxed{\text{yyparse}} \arrow[r] & \text{Parsed input}
\end{tikzcd}$$

Lex with Yacc

Figure 2

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 (in C) or branches of a computed GOTO (in Ratfor). 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]
```

in C or

```
yytext(yyleng)
```

in Ratfor.

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. There is another important routine in Ratfor, named *lexshf*, which is described below under "Character Set". 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.

In Ratfor all of the standard I/O library routines, *input*, *output, unput, yywrap,* and *lexshf,* are defined as integer functions. This requires *input* and *yywrap* to be called with arguments. One dummy argument is supplied and ignored.

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                     [TEde][-+]?{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* for a C host language source and *lex.yy.r* for a Ratfor host environment. There are two I/O libraries, one for C defined in terms of the C standard library [6], and the other defined in terms of Ratfor. To indicate that a Lex source file is intended to be used with the Ratfor host language, make the first line of the file *%R*.

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. The C host language is default, but may be explicitly requested by making the first line of the source file *%C*.

The Ratfor generated by Lex is the same on all systems, but can not be compiled directly on TSO. See below for instructions. The Ratfor I/O library, however, varies slightly because the different Fortrans disagree on the method of indicating end-of-input and the name of the library routine for logical AND. The Ratfor I/O library, dependent on Fortran character I/O, is quite slow. In particular it reads all input lines as 80A1 format; this will truncate any longer line, discarding your data, and pads any shorter line with blanks. The library version of *input* removes the padding (including any trailing blanks from the original input) before processing. Each source file using a Ratfor host should begin with the "%R" command.

UNIX. The libraries are accessed by the loader flags *-llc* for C and *-llr* for Ratfor; the C name may be abbreviated to *-ll.* So an appropriate set of commands is

C Host	Ratfor Host
lex source	lex source
cc lex.yy.c -ll -lS	rc -2 lex.yy.r -llr

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. Note the "-2" option in the Ratfor compile command; this requests the larger version of the compiler, a useful precaution.

GCOS. The Lex commands on GCOS are stored in the "." library. The appropriate command sequences are:

C Host	Ratfor Host
./lex source	./lex source
./cc lex.yy.c ./lexclib h=	./rc a= lex.yy.r ./lexrlib h=

The resulting program is placed on the usual file *.program* for later execution (as indicated by the "h=" option); it may be copied to a permanent file if desired. Note the "a=" option in the Ratfor compile command; this indicates that the Fortran compiler is to run in ASCII mode.

TSO. Lex is just barely available on TSO. Restrictions imposed by the compilers which must be used with its output make it rather inconvenient. To use the C version, type

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(cload)' 'libraryname membername'
```

The first command analyzes the source file and writes a C program on file *lex.yy.text.* The second command runs this file through the C compiler and links it with the Lex C library (stored on 'hr289.lcl.load') placing the object program in your file *libraryname.LOAD(membername)* as a completely linked load module. The compiling command uses a special version of the C compiler command on TSO which provides an unusually large intermediate assembler file to compensate for the unusual bulk of C-compiled Lex programs on the OS system. Even so, almost any Lex source program is too big to compile, and must be split.

The same Lex command will compile Ratfor Lex programs, leaving a file *lex.yy.rat* instead of *lex.yy.text* in your directory. The Ratfor program must be edited, however, to compensate for peculiarities of IBM Ratfor. A command sequence to do this, and then compile and load, is available. The full commands are:

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(rload)' 'libraryname membername'
```

with the same overall effect as the C language commands. However, the Ratfor commands will run in a 150K byte partition, while the C commands require 250K bytes to operate.

The steps involved in processing the generated Ratfor program are:

a. Edit the Ratfor program.
   1. Remove all tabs.
   2. Change all lower case letters to upper case letters.
   3. Convert the file to an 80-column card image file.

b. Process the Ratfor through the Ratfor preprocessor to get Fortran code.

c. Compile the Fortran.

d. Load with the libraries 'hr289.lrl.load' and 'sys1.fortlib'.

The final load module will only read input in 80-character fixed length records. ***Warning:*** Work is in progress on the IBM C compiler, and Lex and its availability on the IBM 370 are subject to change without notice.

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 -lS
```

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]+          {
                scanf(-1, yytext, "%d", &k);
                if (k%7 == 0)
                    printf("%d", k+3);
                else
                    printf("%d",k);
                }
```

to do just that. The rule [0-9]+ recognizes strings of digits; ***scanf*** 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]+                {
                        scanf(-1, yytext, "%d", &k);
                        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. In C, the I/O routines are assumed to deal directly in this representation. In Ratfor, it is anticipated that many users will prefer left-adjusted rather than right-adjusted characters; thus the routine *lexshf* is called to change the representation delivered by *input* into a right-adjusted integer. If the user changes the I/O library, the routine *lexshf* should also be changed to a compatible version. The Ratfor library I/O system is arranged to represent the letter *a* as in the Fortran value *1Ha* while in C the letter *a* is represented 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 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.

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

Sample character table.

It is not likely that C users will wish to use the character table feature; but for Fortran portability it may be essential.

Although the contents of the Lex Ratfor library routines for input and output run almost unmodified on UNIX, GCOS, and OS/370, they are not really machine independent, and would not work with CDC or Burroughs Fortran compilers. The user is of course welcome to replace *input, output, unput* and *lexshf* but to replace them by completely portable Fortran routines is likely to cause a substantial decrease in the speed of Lex Ratfor programs. A simple way to produce portable routines would be to leave *input* and *output* as routines that read with 80A1 format, but replace *lexshf* by a table lookup routine.

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) A language specifier, which must also precede any rules or included code, in the form "%C" for C or "%R" for Ratfor.

7) 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:

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

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.

TSO Lex is an older version. Among the non-supported features are REJECT, start conditions, or variable length trailing context, And any significant Lex source is too big for the IBM C compiler when translated.

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.

M4

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 nth 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.

RJE

UNIX Remote Job Entry User's Guide

A. L. Sabsevitz
K. A. Kelleman

Bell Laboratories
Piscataway, New Jersey 08854

1. PREFACE

A set of background processes running under UNIX† support remote job entry to IBM System/360 and /370 host computers. RJE is the communal name for this subsystem.[1] UNIX communicates with IBM's Job Entry Subsystem by mimicking an IBM 360 remote multileaving work station. The *UNIX User's Manual* entry *rje*(8) summarizes its design and operation. The manual also contains a description of the *send*(1C) command, which is the user's primary method of submitting jobs to RJE, and *rjestat*(1C), which allows the user to monitor the status of RJE and to send operator commands to the host system. This guide is a tutorial overview of RJE and is addressed to the user who needs to know how to use the system, but does *not* need to know details of its implementation. The two following sections constitute an introduction to RJE.

2. PRELIMINARIES

To become a UNIX user, you must receive a login name that identifies you to the UNIX system. You should also get a copy of the *UNIX User's Manual;* it contains a fairly complete description of the system and includes the section *How to Get Started*, which introduces you to UNIX; you should read that section before proceeding with this guide.

In order to begin using RJE, you need only become familiar with a subset of basic commands. You must understand the directory structure of the file system, and you should know something about the attributes of files: see *cd*(1), *chmod*(1), *chown*(1), *cp*(1), *ln*(1), *ls*(1), *mkdir*(1), *mv*(1), *rm*(1). You must know how to enter, edit, and examine text files: see *cat*(1), *ed*(1), *pr*(1). You should know how to communicate with other users and with the system: see *mail*(1), *mesg*(1), *who*(1), *write*(1). And, finally, you might have to know how to describe your terminal to the system: see *ascii*(5), *stty*(1), *tabs*(1).

3. BASIC RJE

Let's suppose that you have used the editor, *ed*(1), to create the file, **jobfile**, that contains your job control statements (JCL) and input data. This file should look exactly like a card deck, except that for convenience alphabetic characters may be in either upper or lower case. Here is an example:

† UNIX is a trademark of Bell Laboratories.

1. In this paper, RJE refers to the facilities provided by UNIX, and *not* to the Remote Job Entry feature of IBM's HASP and JES subsystems.

```
$ cat jobfile
//gener job (9999,r740),pgmrname,class=x usr=(mylogin,myplace)
//step exec pgm=iebgener
//sysprint dd sysout=a
//sysin dd dummy
//sysut2 dd sysout=a
//sysut1 dd *
   first card of data
        .
        .
        .
   last card of data
/*
```

To submit this job for execution, you must invoke the *send*(1C) command:

```
$ send jobfile
```

The system will reply:

```
10 cards
Queued as /usr/rje/rd3125
```

Note that *send* tells you the number of cards it submitted and reports the file name that contains your job in the queue of all jobs waiting to be transmitted to the host system. Until the transmission of the job actually begins, you can prevent the job from being transmitted by doing a **chmod 0** on the queued file to make it unreadable. For our example, you could say:

```
chmod 0 /usr/rje/rd3125
```

When your job is accepted by the host system, a job number will be assigned to it, and an acknowledgement message will be generated. This indicates that your job has been scheduled on the host system. Later, after the job has executed, its output will be returned to the UNIX system. You will be notified automatically of both of these events: if you are logged in when RJE detects these events, and if you are permitting messages to be sent to your terminal (see *mesg*(1)). The following two messages will be sent to you (still using the example above) when the job is scheduled and when the output is returned, respectively:

Two bells
```
12:18:42 gener job 384 —— rd3125 acknowledged
```
Two bells
```
12:21:54 gener job 384 —— /a1/user/rje/prnt0 ready
```

Two bells, with an interval of one second between them, precede each message. They should be interpreted as a warning to stop typing on your terminal, so that the imminent message is not interspersed with your typing.

If you are not logged in when one of these events occurs, or if you do not allow messages to be sent to your terminal, then the notification will be posted to you via the *mail*(1) command. You can prevent messages directly by executing the *mesg*(1) command, or indirectly by executing another command, such as *pr*(1), which prohibits messages for as long as it is active. You may inspect (by invoking the *mail* command) your mail file (/usr/mail/*logname*) at any time for messages that have been diverted. Setting your **MAIL** variable to the name of your mail file will cause the shell to notify you when mail arrives. For this example, the mail might look as follows:

```
$ mail
From rje Mon Aug  1 12:20:36 1977
12:18:42 gener job 384 -- rd3125 acknowledged
? d
From rje Mon Aug  1 12:21:55 1977
12:21:54 gener job 384 -- /a1/user/rje/prnt0 ready
? d
```

The job acknowledgement message performs two functions. First, it confirms the fact that your job has been scheduled for eventual execution. Second, it assigns a number to the job in such a way that the number and the name together will uniquely identify the job for some period of time.

The output ready message provides the name of a UNIX file into which output has been written and identifies the job to which the output belongs (see *ls*(1)):

```
$ ls -l prnt0
-r--r--xr--  1 rje        1184  Aug   1  12:21  prnt0
```

Note that rje retains ownership of the output and allows you only read access to it. It is intended that you will inspect the file, perhaps extract some information from it, and then promptly delete it (see *rm*(1)):

```
$ rm -f prnt0
```

The retention of machine-generated files, such as RJE output, is discouraged. It is your responsibility to remove files from your RJE directory. RJE output files may be truncated if the output exceeds a set limit. This limit is tunable by the system administrator. Output beyond the current limit will be discarded, with no provision for retrieval. If the output were truncated in the previous example, the second notification message would have been:

```
        Two bells
12:21:54 gener job 384 -- /a1/user/rje/prnt0 ready (truncated)
```

The user should also be aware that RJE attempts to keep a set number of blocks free on any file system it uses. This number is also tunable by the system administrator. Warning messages or suspension of certain functions will occur as this limit is approached.

The most elementary way to examine your output is to *cat* it to your terminal. The Appendix of this document shows the result of listing the output of our sample job in this way. Because UNIX has no high volume printing capability, you should route to the host's printer any large listings of which you desire a hard copy.

The structure of an output listing will generally conform to the following sequence:

```
HASP log
jcl information
data sets
HASP end
```

Normally burst pages will not be present. Single, double, and triple spacing is reflected in the output file, but other forms controls, such as the skip to the top of a new page, are suppressed. Page boundaries are indicated by the presence of a blank (space character) at the end of the last line of each page.

The big file scanner *bfs*(1) or the context editor *ed*(1) provide a more flexible method than *cat*(1) for examining printed output; *bfs* can handle files of any size and is more efficient than *ed* for scanning files.

RJE is also capable of receiving punched output as formatted files (see *pnch*(5)); this format allows an exact representation of an arbitrary card deck to be stored on the UNIX machine. However, there are few commands that can be used to manipulate these files. You will probably want to route your punched output to one of the host's output devices.

4. SEND COMMAND

The *send*(1C) command is capable of more general processing than has been indicated in the previous section. In the first place, it will concatenate a sequence of files to create a single job stream. This allows files of JCL and files of data to be maintained separately on the UNIX machine. In addition, it recognizes any line of an input file that begins with the character ~ as being a *control* line that can call for the inclusion, inside the current file, of some other file. This allows you to *send* a top level skeleton that "pulls" in subordinate files as needed. Some of these may be "virtual" files that actually consist of the output of UNIX commands or Shell procedures. Furthermore, the *send* command is able to collect input directly from a terminal, and can be instructed to prompt for required information.

Each source of input can contain a format specification that determines such things as how to expand tabs and how long can an input line be. The manual entry for *fspec*(5) explains how to define such formats. When properly instructed, *send* will also replace arbitrarily defined keywords by other text strings or by EBCDIC character codes. (These two substitution facilities are useful in other applications besides RJE; for that reason, *send* may be invoked under the name *gath* to produce standard output *without* submitting an RJE job.)

Two options of *send* that everyone should be acquainted with are: the ability to specify to which host computer the job is to be submitted, and a flag that guarantees that a job will be transmitted to the host computer in order of submission (relative to other jobs submitted with the same flag). To run our sample job on a host machine known to RJE as A, we would issue the command:

```
$ send A jobfile
```

When no host is explicitly cited, *send* makes a reasonable choice.

To insure that a job will be transmitted in order of submission, set the −x flag:

```
$ send −x jobfile
```

This flag should be used sparingly. The complete list of arguments and flags that control the execution of *send* can be found in *send*(1C).

5. JOB STREAM

It is assumed that the job stream submitted as the result of a single execution of *send* consists of a single *job*, i.e., the file that is queued for transmission should contain one JOB card near the beginning and no others. A priority control card may legitimately precede the JOB card. The JOB card must conform to the local installation's standard. At BISP, it has the following structure:

```
//name job (acct[,...]),pgmrname[,keywds=?] [usr=,...]
```

6. USER SPECIFICATION

A "usr" specification is required on print or punch output that is to be delivered to a UNIX user.

```
usr=(login,place,[level])
```

where *login* is the UNIX login name of the user, *level* is the desired level of notification (see end of this section for an explanation), and *place* is as follows:

A. If *place* is the name of a directory (writable by others), then the output file is placed there as a unique **prnt** or **pnch** file. The mode of the file will be 454.

B. If *place* is the name of an existing, writable (by others), non-executable (by others) file, then the output file replaces it. The mode of the file will be 454.

C. If *place* is the name of a non-existent file in a writable (by others) directory, then the output file is placed there. The mode of the file will be 454.

D. If *place* is the name of an executable (by others) file, then the RJE output is set up as standard input to *place*, and *place* is executed. Five string arguments are passed to *place*. For example, if *place* is a shell procedure, the following arguments are passed as $1 ... $5:

   1. Flag indicating whether file space is scarce in the file system where *place* resides. A 0 indicates that space is *not* scarce, while 1 indicates that it is.
   2. Job name.
   3. Programmer's name.
   4. Job number.
   5. Login name from the "usr=..." specification.

   A ":" is passed if a value is not present. The current directory for the execution of *place* will be set to the directory containing *place*. The environment (see *environ*(7)) will contain values for **LOGNAME** and **HOME** based on the login name from the "usr=..." specification, and a value for **TZ**. Since the login name supplied on the "usr=..." specification cannot be believed for security purposes, the UID will be set to a reserved value.

E. In all other cases, the output will be thrown away.

The *place* value must not be a full path name, unless it refers to an executable file (see D above). For cases A, B, and C above (and case D, if a full path name is not supplied), the name of the user's login directory will be used to form a full path name.

The "usr=..." field may occur anywhere within the first 100 card images sent and within the first 200 output images received by the UNIX system. The only restriction is that it be contained completely on a single line or card image. Therefore, the "usr=..." field may be placed on a JOB card or comment card. It may also be passed as data.

For redirection of output by the host, a "usr=..." card, if not already present, must be supplied by the user. This can be done by placing a job step that creates this card before your output steps.

Messages generated by RJE or passed on from the host are assigned a level of importance ranging from 1 to 9. The levels currently in use are:

3	transmittal assurance
5	job acknowledgement
6	output ready message

The optional *level* field of the "usr=..." specification must be a one or two-digit code of the form *mw*. A message from the host with importance *x* (where *x* comes from the above list) is compared with each of the two decimal digits in *level*. If $x \geq w$ and if the user is logged in and is accepting messages, the message will be written to his or her terminal. Otherwise, if $x \geq m$, the message will be mailed to the user. In all other cases, the message will be discarded. The default *level* is 54. You should specify level 1 if you want to receive complete notification, and level 59 to divert the last three messages in the above list to your mailbox.

7. MONITORING RJE

RJE is designed to be an autonomous facility that does not require manual supervision. RJE is initiated automatically by the UNIX reboot procedures and continues in execution until the system is shut down. Experience has shown RJE to be reasonably robust, although it is vulnerable to system crashes and reconfigurations.

Users have a right to assume that when the UNIX system is up for production use, RJE will also be up. This implies more than an ability to execute the *send*(1C) command, which should be available at all times; it means that queued jobs should be submitted to the host for execution and their output returned to the UNIX system. If a user cannot obtain any throughput from RJE, he or she should so advise the UNIX operators.

The *rjestat*(1C) command, invoked with no arguments will report the status of all RJE links for which a given UNIX system is configured. It may sometimes also print a message of the day from RJE.

```
$ rjestat
RJE to B operating normally.
RJE to A down, reason: IBM not responding.
```

A host machine may be reported to be not responding to RJE because it is down, or because of its operator's failure to initialize the associated line, or because of a communications hardware failure.

Rjestat also has the ability to send operator commands to the host machine and retrieve the responses generated by the commands. Refer to the *rjestat*(1C) manual entry for a complete description of this command.

APPENDIX

Sample JES2 Output Listing

```
$ cat rje/prnt0
14.40.31 JOB 384  $HASP373 GENER  STARTED - INIT 26 - CLASS X - SYS RRMA
14.40.32 JOB 384  $HASP395 GENER  ENDED
------ JES2 JOB STATISTICS ------
 1 AUG 77 JOB EXECUTION DATE

          54 CARDS READ

          76 SYSOUT PRINT RECORDS

           0 SYSOUT PUNCH RECORDS

     0.01 MINUTES EXECUTION TIME
    1       //GENER JOB (9999,R740),PGMRNAME,CLASS=X                    JOB 384
            ***      USR=(MYLOGIN,MYPLACE)
    2       //IEBGENER EXEC PGM=IEBGENER
    3       //SYSPRINT DD DUMMY
    4       //SYSIN DD DUMMY
    5       //SYSUT2 DD SYSOUT=A
    6       //SYSUT1 DD *
            //
IEF236I ALLOC. FOR GENER IEBGENER
IEF237I DMY ALLOCATED TO SYSPRINT
IEF237I DMY ALLOCATED TO SYSIN
IEF237I JES ALLOCATED TO SYSUT2
IEF237I JES ALLOCATED TO SYSUT1
IEF142I GENER IEBGENER - STEP WAS EXECUTED - COND CODE 0000
IEF285I   JES2.JOB0384.SO0102                      SYSOUT
IEF285I   JES2.JOB0384.SI0101                      SYSIN
IEF373I STEP /IEBGENER/ START 77242.1440
IEF374I STEP /IEBGENER/ STOP 77242.1440 CPU 0MIN 00.13SEC SRB 0MIN 00.01SEC VIRT 36K SYS 188K
******* SERVICE UNITS=0000174          SERVICE RATE=0000268 SERVICE UNITS/SECOND
******* PERFORMANCE GROUP=005
******* EXCP COUNT BY UNIT ADDRESS
IEF375I  JOB /GENER / START 77242.1440
IEF376I  JOB /GENER / STOP  77242.1440 CPU    0MIN 00.13SEC SRB    0MIN 00.01SEC
******* SERVICE UNITS=0000174          SERVICE RATE=0000268 SERVICE UNITS/SECOND
******* APPROXIMATE PROCESSING TIME=            .01 MINUTES
******* EXCPS=000000000
******* PROJECTED CHARGES=             .01

          first line of data
               .
               .
               .
          last line of data

*OS/VS2 REL 3.7 JES2*  END  JOBNAME=GENER   BIN=R740      JOB #=384  PGMRNAME
*OS/VS2 REL 3.7 JES2*  END  JOBNAME=GENER   BIN=R740      JOB #=384  PGMRNAME
*OS/VS2 REL 3.7 JES2*  END  JOBNAME=GENER   BIN=R740      JOB #=384  PGMRNAME

$ rm -f rje/prnt0
```

UNIX Remote Job Entry Administrator's Guide

M. J. Fitton

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

1.1 Purpose

This document is intended to augment the existing body of documentation on the design and operation of UNIX† IBM RJE[1]. The reader should be familiar with *rje*(8), and the *UNIX Remote Job Entry User's Guide*, April 1, 1980. There will be assumptions made concerning allocation of responsibilities between UNIX and IBM operations, hardware configuration, etc. Although these assumptions may not fully apply to your location, they should not interfere with the intent of this document.

The major topics discussed in this paper are as follows:

- SETTING UP — hardware requirements and RJE generation on the IBM and UNIX systems.
- DIRECTORY STRUCTURES — the controlling RJE directory structure and a typical RJE subsystem directory structure.
- RJE PROGRAMS — programs that make up an RJE subsystem.
- UTILITY PROGRAMS — utility programs that are available for debugging or tracing.
- RJE ACCOUNTING — the accounting of jobs done by RJE, and some methods for using this accounting data.
- TROUBLE SHOOTING — error recovery and procedures for identifying and fixing RJE problems.

1.2 Facilities

Discussions will focus on a hypothetical RJE connection between a UNIX system, *pwba*, and an IBM 370/168, referred to as *B*. We also assume that *pwba* is connected to an IBM 370/158, referred to as *C*. The UNIX machine emulates an IBM System/360 remote multi-leaving work station. For more information on the multi-leaving protocol, see Appendix B of *OS/VS MVS JES2 Logic* (SY24-6000-1).

2. SETTING UP

2.1 Hardware

To use RJE on a UNIX system the following hardware is needed (one per remote line):

- KMC11-B Microprocessor — used to drive the RJE line
- DMC11-DA or DMC11-FA line unit — the DMC11-DA interfaces with Bell 208 and 209 synchronous modems or equivalent. Speeds of up to 19,200 bits per second can be used. The DMC11-FA interfaces with Bell 500 A LI/5 synchronous modems or equivalent. Speeds of up to 250,000 bits per second can be used.

† UNIX is a trademark of Bell Laboratories.

1. In this paper, RJE refers to the facilities provided by UNIX and *not* to the Remote Job Entry feature of IBM's HASP and JES2 subsystems.

On the DMC11 line unit, the Cyclic Redundancy Check (CRC) switch should be set to inhibit automatic transmission of CRC bytes. The line unit should hold the line at logical zero when inactive. For a more detailed description of the above hardware, see *Terminals and Communications Handbook*, Digital Equipment Corporation, 1979.

2.2 IBM Generation

The following applies to the host IBM system. The remote line to the UNIX machine should be described as a System/360 remote work station. The following parameters must be initialized and *must* agree with their counterparts on the UNIX machine:

- Number of printers (NUMPR) — the number of logical printers (up to 7)
- Number of punches (NUMPU) — the number of logical punches (up to 7)
- Number of readers (NUMRD) — the number of logical readers (up to 7)

The JES2 parameters for our hypothetical connection to IBM system *B* are as follows:

```
RMT5 S/360,LINE=5,CONSOLE,MULTI,TRANSP,NUMPR=5,
        NUMPU=1,NUMRD=5,ROUTECDE=5
R5.PR1 PRWIDTH=132
R5.PR2 PRWIDTH=132
R5.PR3 PRWIDTH=132
R5.PR4 PRWIDTH=132
R5.PR5 PRWIDTH=132
R5.PU1 NOSUSPND
R5.RD1 PRIOINC=0,PRIOLIM=14
R5.RD2 PRIOINC=0,PRIOLIM=14
R5.RD3 PRIOINC=0,PRIOLIM=14
R5.RD4 PRIOINC=0,PRIOLIM=14
R5.RD5 PRIOINC=0,PRIOLIM=14
```

System *pwba* is referenced by line 5 (LINE=5), remote 5 (RMT5). It is defined as having a console, for the *rjestat*(1C) command, five printers, one punch, and five readers. Although you may have up to seven printers or punches, the total number of printers and punches may not exceed eight. The line is described as a transparent (TRANSP), multi-leaving (MULTI) line. The remaining information describes attributes of the printers, punches, and readers.

Normally, separator pages are transmitted with IBM print files. UNIX RJE does not remove separator pages. To prevent transmission of separator pages on printer 1 of the previous example, its attributes would be:

```
R5.PR1 PRWIDTH=132,NOSEP
```

NOSEP should be included for all printers when separator pages are not desired. Most IBM systems can also be told via a console command to cancel transmission of separator pages on printers. This can be done from the IBM system console, or from the remote UNIX machine via *rjestat*. For example, the following JES2 command would cancel separator page transmission on printer 1:

```
$TR5.PR1,S=N
```

2.3 UNIX Generation

If the RJE remote dialing facility is to be used, the administrator must make sure that the definition for RJECU in the file /usr/include/rje.h is the device to be used for remote dialing. RJECU is defined to be /dev/dn2 when distributed. To compile and install RJE, the normal *make*(1) procedures are used (see *Setting up UNIX*). Once an RJE subsystem has been installed, the remote line must be described in the configuration file /usr/rje/lines. This file as it exists on our hypothetical system *pwba* is as follows:

```
B pwba /usr/rje1 rje1 vpm0 5:5:1 1200:512:y
C pwba /usr/rje2 rje2 vpm1 1:1:1 1200:512
```

/usr/rje/lines is accessed by all components of RJE. Each line of the table (maximum of 8) defines an RJE connection. Its seven columns may be labeled **host**, **system**, **directory**, **prefix**, **device**, **peripherals**, and **parameters**. These columns are described as follows:

- **host** — The IBM System name, e.g., A, B, C. This string can be up to 5 characters long.
- **system** — The UNIX System name (see *uname*(1)).
- **directory** — the directory name of the servicing RJE subsystem (e.g., **/usr/rje2**).
- **prefix** — the string prepended to most files and programs in the **directory** (i.e., **rje2**).
- **device** — the name of the controlling Virtual Protocol Machine (VPM) device, with **/dev/** excised. In order to specify a VPM device, all VPM software must be installed, and the proper special files must be made (see *vpm*(4) and *mknod*(1M)).
- **peripherals** — information on the logical devices (readers, printers, punches) used by RJE. There are three subfields. Each subfield is separated by ":" and is described as follows:

  1. Number of logical readers.
  2. Number of logical printers.
  3. Number of logical punches.

  Note: the number of peripherals specified for an RJE subsystem *must* agree with the number of peripherals that have been described on the remote machine for that line.
- **parameters** — this field contains information on the type of connection to make. Each subfield is separated by ":". Any or all fields may be omitted; however, the fields are positional. All but trailing delimiters must be present. For example, in:

  ```
  1200:512:::9-555-1212
  ```

  subfields 3 and 4 are missing. Each subfield is defined as follows:

  1. **space** — this subfield specifies the amount of space (S) in blocks that RJE tries to maintain on file systems it touches. The default is 0 blocks. *Send*(1C) will not submit jobs and *rjeinit* issues a warning when less than $1.5S$ blocks are available; *rjerecv* stops accepting output from the host when the capacity falls to S blocks; RJE becomes dormant, until conditions improve. If the space on the file system specified by the user on the "usr=" card would be depleted to a point below S, the file will be put in the job subdirectory of the connection's home directory rather than in the place that the user requested.
  2. **size** — this subfield specifies the size in blocks of the largest file that can be accepted from the host without truncation taking place. The default is no truncation. Note that UNIX has a default one Mega-byte file size limit.
  3. **badjobs** — this subfield specifies what to do with undeliverable returning jobs. If an output file is undeliverable for any reason other than file system space limitations (e.g., missing or invalid "usr=" card) and this subfield contains the letter **y**, the output will be retained in the **job** subdirectory of the home directory, and login **rje** is notified via *mail*(1). If this subfield has any other value, undeliverable output will be discarded. The default is **n**.
  4. **console** — this subfield specifies the status of the interactive status terminal for this line. If the subfield contains an **i**, the status console facilities of *rjestat* will be inhibited. In all cases, the normal non-interactive uses of *rjestat* will continue to function. The default is **y**.

5. **dial-up** — this subfield contains a telephone number to be used to call a host machine. The telephone number may contain the digits 0 through 9, and the character "−", which denotes a pause. If the telephone number is not present, no dialing is attempted, and a leased line is assumed.

When multiple readers have been specified, jobs that are submitted for transmission to IBM are assigned to the reader with the fewest cards on it. Each reader gets an equal amount of service. This prevents smaller jobs from having to wait for a previously submitted large job to be transmitted. When multiple printers or punches have been specified, returning jobs get assigned to free printers (or punches) allowing smaller output files to bypass large output files.

Deciding how many peripherals to specify depends on the use of that RJE subsystem. If an RJE subsystem is heavily used for off-line printing (i.e., output does not return to the UNIX machine), the administrator would want to specify multiple readers, but would not have a need for multiple printers or punches.

3. DIRECTORY STRUCTURES

3.1 Controlling Directory

The controlling directory used by RJE is **/usr/rje**. This directory contains RJE programs for use by separate RJE subsystems (e.g., **rje1**, **rje2**, **rje3**), and the shell queuer's directory. Most RJE programs existing here have been compiled such that each RJE subsystem shares the text of these programs. A snapshot of this directory on our hypothetical machine is as follows:

```
-rwxr-xr-x   2 rje    rje     4068 Mar  4 10:42 cvt
-rw-r--r--   1 rje    rje       42 Apr 10 09:52 lines
-rwxr-xr-x   2 rje    rje    15096 Apr 10 13:01 rjedisp
-rwxr-xr-x   2 rje    rje     2328 Mar  4 10:21 rjehalt
-rwxr-xr-x   2 rje    rje    10396 Apr 15 10:07 rjeinit
-r-x------   2 rje    rje      785 Apr  8 09:00 rjeload
-rwsr-xr-x   2 rje    rje     5040 Mar 27 09:28 rjeqer
-rwxr-xr-x   2 rje    rje     4072 Apr  1 15:40 rjerecv
-rwxr-xr-x   2 rje    rje     3888 Mar 27 09:35 rjexmit
-rwsr-xr-x   1 root   rje     2696 Mar 27 14:42 shqer
-rwxr-xr-x   2 rje    rje     5920 Apr  2 15:47 snoop
drwxr-xr-x   2 rje    rje       80 Mar 25 13:26 sque
```

RJE subsystems are generated in their own directory by linking the program names in this directory to the appropriate names in the subsystem directory. The programs are described in Section 4. The file **lines** is the configuration file used by all RJE subsystems. The directory **sque** is used by the Shell queuer (*shqer*). This directory contains:

```
-rw-r--r--   1 rje    rje        0 Feb 14 14:04 errors
-rw-r--r--   1 rje    rje        0 Feb 14 14:04 log
```

When *shqer* has work to do, the files **log** and **errors** will be of non-zero length, and temporary files (**tmp***) will also appear here. For a complete description of *shqer* and these files, see Section 4.8.

3.2 Subsystem Directory

The RJE subsystem described in this section maintains the connection between *pwba* and IBM *B*, and will be referred to as *rje1*. The first line of **/usr/rje/lines** (see Section 2.3) describes *rje1*. As noted in this file, *rje1* runs in the directory **/usr/rje1**. A snapshot of this directory is as follows:

```
-rw-r--r--   1 rje    rje     4990 Apr 15 08:30 acctlog
-rwxr-xr-x   2 rje    rje     4068 Mar  4 10:42 cvt
-rw-r--r--   1 rje    rje        0 Apr 15 04:02 errlog
drwxrwxrwx   2 rje    rje      192 Apr 10 09:51 job
-rw-r--r--   1 rje    rje      194 Apr 15 08:11 joblog
-rw-r--r--   1 rje    rje        0 Apr 15 08:11 resp
-rwxr-xr-x   2 rje    rje    15096 Apr 10 13:01 rjeldisp
-rwxr-xr-x   2 rje    rje     2328 Mar  4 10:21 rjelhalt
-rwxr-xr-x   2 rje    rje    10396 Apr 15 10:07 rjelinit
-r-x------   2 rje    rje      785 Apr  8 09:00 rjelload
-rwsr-xr-x   2 rje    rje     5040 Mar 27 09:28 rjelqer
-rwxr-xr-x   2 rje    rje     4072 Apr  1 15:40 rjelrecv
-rwxr-xr-x   2 rje    rje     3888 Mar 27 09:35 rjelxmit
drwxr-xr-x   2 rje    rje      144 Apr 15 08:30 rpool
-rwxr-xr-x   2 rje    rje     5920 Apr  2 15:47 snoop0
drwxrwxrwx   2 rje    rje      176 Apr 10 13:03 spool
drwxr-xr-x   2 rje    rje      224 Apr 10 13:56 squeue
-rw-r--r--   1 rje    rje        0 Apr 15 10:30 stop
-rw-r--r--   1 rje    rje      274 Mar  7 20:25 testjob
```

The programs *rjel**, *cvt*, and *snoop0* are linked to the corresponding programs in /usr/rje, and are described in detail in Section 4. The remaining files and their uses are as follows:

- **acctlog** — accounting data is stored in this file, if it exists. This file is the responsibility of the RJE administrator. For a discussion of its uses, see Section 5.
- **errlog** — used by *rjel* to log errors. It can be useful for debugging *rjel* problems.
- **joblog** — used by *rjelqer* and *rjestat* to notify *rjelxmit* that a job (or console request) has been submitted. It also contains the process-group number of the *rjel* processes. The program *cvt* can be used to convert this file to a readable form.
- **resp** — contains console messages received from IBM *B*. These messages can be responses for *rjestat*, or IBM responses to submitted jobs (i.e., on reader messages). This file is truncated if it grows to a size greater than 70,000 bytes.
- **stop** — indicates that *rjelhalt* has been executed. The existence of this file indicates to *rjestat* that *rjel* has been halted by the operator.
- **testjob** — a sample job that can be submitted to test the *rjel* subsystem. Originally, the job control statements may have to be changed to suit your IBM system.

When *rjel* terminates abnormally, the file dead should appear in this directory. This file contains a short message indicating why *rjel* is not operating, and is used by *rjestat* to report the problem. The remaining directories and their uses are as follows:

- **job** — used to save undeliverable jobs, if the proper parameter has been specified in /usr/rje/lines. The sample job described above is also delivered to this directory. This directory should be mode 777.
- **rpool** — contains temporary files used to gather output from the remote machine. These files are named **pr*** (for print output files), and **pu*** (for punch output files). Once a complete file has been received, the file is dispatched in the proper way by *rjeldisp*.
- **spool** — used by *send* to store temporary files to be submitted to the remote machine. This directory must be mode 777.
- **squeue** — used by *rjel* to store submitted files until they are transmitted. The program *rjelqer* is used by *send* to move the temporary files in the **spool** directory to this directory.

4. RJE PROGRAMS

All programs described below, with the exception of *rjestat*, exist in /usr/rje. These programs are "shared text" and are linked (except *shqer*) to the proper names in each subsystem directory. The names described below are generic; the programs in the *rje2* directory would be *rje2qer*, *rje2init*, etc.

Each available RJE subsystem occupies three process slots. The slots are used for *rje?xmit*, the transmitter; *rje?recv*, the receiver; and *rje?disp*, the dispatcher. One additional process slot is used for *shqer*, regardless of how many subsystems are available.

Each RJE subsystem tries to be self-sustaining, and logs any errors encountered during normal operation in its **errlog** file.

4.1 Rjeqer

This program is used by *send* to queue files for transmission. When invoked, it performs the following steps:

1. Moves the temporary *pnch*(5) format file in the **spool** directory to the **squeue** directory.
2. Writes an entry at the end of the file **joblog** containing:
   - the name of the file to be transmitted
   - the submitter's user ID
   - the number of card images in the file
   - the message level for this job

   The file **joblog** is used to notify *rjexmit* of work to be done.
3. Notifies user that file has been queued.

Send determines which host system is desired, and invokes the proper *rje?qer* by getting the prefix from the lines file (e.g., if sending to IBM *C* from our machine, *rje2qer* would be invoked).

4.2 Rjeload

This program is used to start an RJE subsystem. Its prefix determines which subsystem to start (e.g., *rje2load* starts *rje2*). To start the RJE subsystems on our machine, the following commands are executed in /etc/rc when changing to *init* state 2 (multi-user):

```
rm −f /usr/rje/sque/log
su rje −c "/usr/rje1/rje1load vpb0 kmc0"
su rje −c "/usr/rje2/rje2load vpb1 kmc1"
```

The file **/usr/rje/sque/log** is removed to ensure the correct operation of *shqer*. When invoked, *rjeload* performs the following steps:

1. Uses the VPM device from **/usr/rje/lines** to link the proper devices (see *vpmset*(1C)).
2. Uses *kasb*(1) to perform the following:
   - reset the KMC
   - load the VPM script (**/etc/rjeproto**)
   - start the KMC running
3. Executes *rje?init* to start the *rje?* processes (e.g., *rje2load* executes *rje2init*).

4.3 Rjehalt

This program is used to halt an RJE subsystem. To halt *rje2* on our machine, **/usr/rje2/rje2halt** is executed. This should be done in the *shutdown* procedure for your machine to ensure graceful termination of RJE. *Rjehalt* will allow only those users with permission to halt an RJE subsystem. *Rjehalt* uses the header on the file **joblog** to get the process-group of the RJE subsystem processes. This group is signaled to terminate. When all processes have terminated, *rjehalt* sends a "signoff" record to the host machine. This signoff record is taken from the file **signoff** (ASCII text), if it exists, otherwise a "/*signoff" record is sent. On completion, *rjehalt* creates the file **stop** in the subsystem directory, that causes *rjestat* to report that RJE to the corresponding host has been stopped by the operator.

4.4 Rjeinit

This program initializes an RJE subsystem. It is used by *rjeload*, and can be used to restart a subsystem if the VPM script has previously been started. *Rjeinit* should only be executed by user **rje**. *Rjeinit* fails if there are less than 100 blocks or 10 inodes free in the file system. It issues a warning if there are less than 1.5X blocks, (where X is the first field in the parameters for that line), or 100 inodes free in the file system. If *rjeinit* fails, the reason for the failure is reported, and the file **dead** is created containing "Init failed". This will be reported by *rjestat* until a subsequent *rjeinit* succeeds. *Rjeinit* performs the following functions:

1. Dials a remote host if specified (see Section 2.3).
2. Truncates the console response file **resp**.
3. Sends a signon record to the host. The signon record is taken from the file **signon** (ASCII text), if it exists, otherwise *rjeinit* sends a blank record as a signon.
4. Sets up pipes for process communication.
5. Resets process-group for RJE subsystem and restarts error logging.
6. Rebuilds the **joblog** file from jobs queued for transmission.
7. Notifies *rjedisp* (via a pipe) of any returned files still remaining in the **rpool** directory.
8. Starts the appropriate background processes (*rje?xmit*, *rje?recv*, and *rje?disp*).
9. Reports started or not started.

If failure occurs in a background process, it is reported by that process (error logging). The failing process will normally attempt to reboot the subsystem by executing *rje?init* with a + as its argument (see Section 7). When *rjeinit* is executed with + as its argument, this indicates an attempted reboot, and *rjeinit* will behave differently (no re-dialing is done to remote hosts, errors are logged rather than printed, etc.).

4.5 Rjexmit

This program writes data to the VPM device. *Rjexmit* is started by *rjeinit* and runs in the background. When running, *rjexmit* performs the following processing:

1. Checks the **joblog** file for files to be transmitted. This is done every 5 seconds when not transmitting data. When transmitting data, the **joblog** is checked after transmitting 1 block from each active **reader**[2], and the **console**[3].

2. Reader refers to the logical readers used by RJE.

3. Console refers to the RJE logical console, which is separate from the logical readers.

2. Queues files from the **joblog** according to the first two characters of the file name:
   - **rd*** — these files are queued on the reader with the fewest cards. Normal use of the *send* command creates these files.
   - **sq*** — these files are queued on the last available reader to assure sequential transmission. Using the **−x** option to the *send* command creates these files.
   - **co*** — these files are queued on the console. The *rjestat* command creates these files.

   All files described above contain EBCDIC data.
3. Sends information to *rjedisp* (via a pipe) for use in user notification of job status (see Section 4.7).
4. Builds blocks for transmission from active readers and the console. These blocks are built according to the multi-leaving protocol.
5. Performs the following peripheral control:
   - Sends requests to open readers when jobs have been assigned to them. These readers are not active until a grant is received from *rjerecv* (via a pipe).
   - Halts and activates readers when waits or starts (respectively) are received from *rjerecv*.
   - Sends printer or punch grants when an open request is received from *rjerecv*.
6. Notifies *rjedisp* that a file has been transmitted, and unlinks the file.

If *rjexmit* encounters fatal errors, it creates the **dead** file with an appropriate message, and signals the other background processes to exit. If possible, *rjexmit* will attempt to reboot the RJE subsystem by executing *rjeinit*.

4.6 Rjerecv

This program reads data from the VPM device. *Rjerecv* is started by *rjeinit* and runs in the background. When running, *rjerecv* performs the following processing:

1. Reads blocks of data received from the host system.
2. Handles data received according to its type. The two types of data are:
   - **Control information** — *rjerecv* performs the following peripheral device control:
     a. Notifies *rjexmit* of grants to its requests to open readers.
     b. Passes wait and start reader information to *rjexmit*.
     c. Passes open requests (for printers and punches) from the host to *rjexmit*.
   - **User Information** — the three major types of user information received are:
     a. Console responses and job status messages. This data is appended to the **resp** file for use by *rjestat* and *rjedisp*.
     b. The printer output from user jobs. This data is collected in temporary files (**pr***) in the **rpool** directory. When a complete print job has been received, *rjerecv* notifies *rjedisp* (via a pipe) that the file is to be dispatched.
     c. The punch output from user jobs. This data is handled the same as printer output except that the **rpool** files are named **pu***.
3. If the console response file resp exceeds 70,000 characters, *rjerecv* truncates the file.
4. *Rjerecv* stops accepting output from the remote machine if the number of free blocks in the file system falls below space blocks (space is described in Section 2.3).

5. *Rjerecv* truncates files to size blocks if a received file exceeds this value (size is described in Section 2.3).

If *rjerecv* encounters fatal errors, it creates the dead file with an appropriate error message, signals the other background processes to exit, and reboots the RJE subsystem.

4.7 Rjedisp

This program dispatches user information. *Rjedisp* is started by *rjeinit* and runs in the background. When running, *rjedisp* performs the following processing:

1. Dispatches output; the two types of output are printer and punch output. After receiving notification of output ready from *rjerecv*, *rjedisp* searches for a "usr=" line in the received file. The format of a "usr=" line is as follows:

   usr=(user,place,level)

   Rjedisp dispatches the output according to the place field. See *UNIX Remote Job Entry User's Guide* for a detailed description of the user specification.

2. Dispatches messages. The three types of messages are as follows:

   - Job transmitted — this message is sent to the submitting user when *rjedisp* reads this event notice from the *rjexmit* pipe.
   - Job acknowledgement — *rjedisp* dispatches IBM acknowledgement messages to submitting users. If a job is not acknowledged properly or within a reasonable amount of time, a "Job not acknowledged" message is dispatched.
   - Output processing — *rjedisp* dispatches job output messages according to the options specified on the "usr=" card. A normal output message indicates the returned file name is ready.

   Messages can be masked by using the *level* on the "usr=" card.

3. Whenever output is to be handled by *shqer*, *rjedisp* checks that *shqer* is running. This is done by looking for the *shqer* log file. If this file does not exist, *rjedisp* starts *shqer*.

4.8 Shqer

This program executes user programs when they appear in the *place* field of the "usr=" line in a returned output file (print or punch). *Shqer* is started by *rjedisp* when the first output file using this feature is returned. Subsequent files using this feature are logged for execution by *rjedisp*. When started, *shqer* performs the following processing:

1. Builds the log file from file names in the /usr/rje/sque directory. Each log entry is the name of a file (tmp?) that contains the following information:

   - the name of the file to be executed
   - the name of the input file (file returned from IBM)
   - the name of the IBM job
   - the programmer name
   - the IBM job number
   - the user's name from the "usr=" line
   - the user's login directory
   - the minimum file system space

2. *Shqer* uses two parameters. The first is the delay time between log file reads. The second is a *nice*(2) factor which is applied to any programs spawned by *shqer*. These values are

defined in **/usr/include/rje.h** (QDELAY and QNICE).

3. When each log entry is read, the appropriate program is spawned with the following characteristics:
   - The returned RJE file is the standard input to the program.
   - The standard and diagnostic outputs are **/dev/null**.
   - The LOGNAME, HOME, and TZ variables are set to the appropriate values.
   - The arguments to the spawned program, in order, are:
     a. a numerical value indicating that the file system free space is equal or above (0) or below (1) space blocks (see Section 2.3).
     b. the IBM job name.
     c. the programmer name.
     d. the IBM job number.
     e. the user's login name.
4. After executing each program, the **tmp?** file and the returned RJE file are removed.

5. UTILITY PROGRAMS

5.1 Snoop

Snoop is the generic name of a program that can be used to trace the state of a VPM device and its associated communications line. *Snoop* depends on the *trace*(4) driver for its information. It reads trace entries from **/dev/trace** and converts them into a readable form that is printed on the standard output.

The usable name of *snoop* for a particular RJE subsystem is *snoopN*, where *N* is the low order three bits from the VPM minor device number. If VPM device names adhere to the **vpm0**, **vpm1**, **vpm***n* naming convention, each *snoop* name corresponds to its VPM device. In our hypothetical system, **vpm0** is used by the **rje1** subsystem, and **vpm1** is used by the **rje2** subsystem (see Section 2.3). Therefore, **/usr/rje1/snoop0** and **/usr/rje2/snoop1** are linked to **/usr/rje/snoop**.

Each *snoop* prints trace entries for its associated VPM device. Trace entries are printed in the following form:

sequence **type** **information**

where:

- **sequence** specifies the order of trace occurrences. It is a value between 0 and 99.
- **type** specifies the action being traced (e.g., transfers, driver activity).
- **information** describes data being transferred and driver activity.

The following table explains the meaning of trace **types** and their associated **information**.

type	**information**	**meaning**
CL	Closed	The VPM device has been closed.
CL	Clean	The VPM driver is cleaning up for this device.
OP	Opened	The VPM has been successfully opened.

OP	Failed(open)	The open failed because the device was already open.
OP	Failed(dev)	The open failed because the device number was out of range.
OP	Failed(set)	The open failed because the KMC could not be reset.
RR	Buf	The VPM script has returned a receive buffer to the VPM driver.
RX	Buf	The VPM script has returned a transmit buffer to the VPM driver.
RD	*num* bytes	*Num* bytes were read from the VPM device by *rjerecv*.
SC	Exit(*num*)	The VPM script has terminated. The VPM exit code is *num*. Exit codes are defined in *vpm*(4).
ST	Startup	The KMC has been started.
ST	Stopped	The VPM script has been stopped.
TR	Started	The script has started tracing.
TR	R-ACK	A two byte acknowledgement (ACK) string has been received from the remote system. This indicates that the previous transmission was properly received.
TR	S-ACK	A two byte acknowledgement (ACK) string has been transmitted to the remote system.
TR	R-NAK	A "not-acknowledged" (NAK) character has been received from the remote system. This indicates that the previous transmission was not properly received.
TR	S-NAK	A "not-acknowledged" (NAK) character has been transmitted to the remote system.
TR	R-ENQ	A enquiry (ENQ) character has been received from the remote system.
TR	S-ENQ	A enquiry (ENQ) character has been transmitted to the remote system.
TR	R-WAIT	The remote machine has requested that no data be transmitted to it.
TR	R-OKBLK	A valid data block was received from the remote machine.
TR	R-ERRBLK	An invalid Cyclic Redundancy Check (CRC) was received with a data block.
TR	R-SEQERR	The block sequence count on a received data block was invalid.
TR	R-JUNK	An invalid data block was received from the remote system.
TR	TIMEOUT	The remote machine did not respond within 3 seconds.
TR	S-BLK	A data block has been transmitted to the remote system.

WR	*num* bytes	*Num* bytes were written to the VPM device by *rjexmit*.

Trace entries of type **TR** are traces from the VPM script. Section 7.5 describes required responses to events and shows examples of typical *snoop* output.

5.2 Rjestat

This program is supplied as a user command. The program's two functions are to describe the status of the RJE subsystems and to provide a remote IBM status console. The remainder of this section describes these two functions.

5.2.1 RJE Status

When invoked, *rjestat* reports the status of the RJE subsystems. If remote system (**host**) names are specified, only those statuses are reported. *Rjestat* uses the following rules to report the status of a subsystem:

- *Rjestat* prints the contents of the file **status** if it exists in the subsystem directory. This file can contain any message the administrator wishes to have printed when users use *rjestat*.
- If the file **dead** exists in the subsystem's directory, the subsystem is not operating and the reason is contained in the file. *Rjestat* reports that RJE to **host** is down and prints the contents of the **dead** file as the reason.
- If the file **stop** exists in the subsystems directory, the *rjehalt* program has been used to inhibit that RJE subsystem. *Rjestat* reports that RJE to **host** has been stopped by the operator.
- If neither the **dead** nor the **stop** file exists, *rjestat* reports that RJE to **host** is operating normally.

Rjestat is supplied as the user's vehicle for checking the status of RJE. It is not meant to be an administrative tool; however, the reason for failure can be used to track the problem.

5.2.2 Status Console

To use *rjestat* as a status console, the **−***shost* argument is used. *Rjestat* prints the status of the subsystem, then prompts with **host:** if the subsystem is up. Each console request is submitted to the RJE processes for transmission, and output is handled as specified. *Rjestat* checks the status prior to submitting each request, and will tell the user to try later if the subsystem goes down. *Rjestat* allows the **rje** or super-user logins to submit other than display requests. For a complete description of how to use the status console features, see *rjestat*(1C).

5.3 Cvt

This program converts any subsystem's **joblog** file to readable form. The first line printed is the process group number of the subsystem processes. The remaining output consists of entries in the following form:

```
file    user-id    records    level
```

Where *file* is the name of the submitted file, *user-id* is the submitters user number, *records* is the number of "card" images, and *level* is the message level. The *records* and *level* fields are not used if the file name is **co*** (console request submitted by *rjestat*).

6. RJE ACCOUNTING

Each RJE subsystem will store accounting information in the **acctlog** file, if it exists. It is the responsibility of the RJE administrator to create and maintain this file in the subsystem's directory. Entries in this file describe RJE line use and are of the following form:

```
day    time    file    user    records
```

Each field is delimited by a tab character. The meanings of each field is as follows:

1. day — The day of occurrence in the form *mm/dd*.
2. time — The time of occurrence in the form *hh:mm:ss*.
3. file — The name of the UNIX file. The first two characters identify its type as follows:
   - **rd/sq** — the file was transmitted to the remote system
   - **pr** — the print output file was received from the remote system
   - **pu** — the punch output file was received from the remote system
4. user — The user ID of the user responsible for the transfer.
5. records — The number of records (card images) transferred for this file.

Since **acctlog** data is not used by RJE, it should not be allowed to grow too large. This can be accomplished by moving or processing the file during a system reboot (i.e., in **/etc/rc** *before* the RJE subsystems are started).

The following list describes some of the reports that could be generated from the **acctlog** data. Implementation of a program to produce accounting reports is the responsibility of the administrator.

- **Periodic Reports** — by using the **day** and **time** fields in the data, periodic usage reports can be produced.
- **By User Reports** — by using the **user** field in the data, usage-by-user reports can be produced.
- **By Subsystem Reports** — by using the **/usr/rje/lines** file information and each **acctlog** file, a usage-by-subsystem (or remote system) report can be produced.

Other reports can be produced using the type of file, size of jobs, etc.

7. TROUBLE SHOOTING

This section deals with RJE problems, and some methods for resolving them. The topics discussed in this section are as follows:

- Automatic Error Recovery
- Manual Error Recovery
- RJE Problems
- KMC/VPM Problems
- Trace Interpretation

7.1 Automatic Error Recovery

RJE attempts to be self-sustaining with respect to its availability. In general, if problems occur on the communications line or the remote machine (e.g., a crash) RJE will continually try to restart itself (this action will be referred to as a "reboot"). For example, if an RJE subsystem is started using *rjeload*, but the IBM system is not available, a fatal error will occur. The process that detects this error (usually *rjexmit* or *rjerecv*) will reboot the subsystem by executing *rjeinit* with a + as its argument. When *rjeinit* detects a + argument, it waits one minute before attempting to bring up the subsystem.

The *rjehalt* program can be used to prevent an RJE subsystem from rebooting itself when the remote system is not available for a known period of time. When the remote system is made available, the subsystem may be started in the normal way.

7.2 Manual Error Recovery

In order to manually recover from errors, one must know how to start and stop an RJE subsystem. There are two ways to start an RJE subsystem:

- *rje?load* — this program loads and starts the VPM script, and executes *rje?init*.
- *rje?init* — this program starts the *rje?* subsystem. In order to use this program, the VPM script must be loaded and started.

To stop the *rje?* subsystem, the *rje?halt* program should be executed. This stops the subsystem gracefully and will prevent a reboot.

The *rjeload* program must be used to start RJE for the first time (after a UNIX system reboot). Subsequently, as long as the script is running, execution sequences of *rjehalt* and *rjeinit* will stop and start RJE.

Manually starting and stopping RJE can be useful in tracking down problems. For example, if user jobs are not being submitted to the host machine, the following sequence can ease identification of the problem:

1. Halt the ailing subsystem.
2. Start a *snoop* process in the background with its output redirected to a file.
3. Restart the subsystem.
4. Scan the *snoop* output to determine where the problem is.

The *snoop* program is the most useful software tool for identifying RJE problems. Its uses are described in Section 7.5.

7.3 RJE Problems

This section describes problems that can occur in an RJE subsystem. These problems generally occur when the subsystem has not been set up properly. The following is a list of things to check to ensure that an RJE subsystem has been set up properly.

1. IBM description — the description of the remote UNIX machine must be consistent with the description in Section 2.2.
2. UNIX description — the file **/usr/rje/lines** must be set up properly. Section 2.3 describes this file in detail.
3. KMC/VPM setup — the VPM software must be installed and the proper VPM and KMC devices made. Each VPM device must correspond to the proper KMC device; see *vpm*(4).
4. Free space — as a general rule, all file systems must have a reasonable amount of free space. File systems containing RJE subsystems must have sufficient free space as described in Section 2.3 to ensure proper RJE operation.
5. Directories — each subsystem's directory and the controlling directory should be checked for the following:
   - All needed files exist.
   - The proper prefix is on each applicable RJE program.
   - The link count is correct for files that are linked.
   - All file and directory modes are correct.

   A sample subsystem directory and the controlling directory are shown in Section 3.
6. Initialization — peripherals information must be consistent on both systems (see Section 2.3). The line must be started on the IBM system, proper hardware connections made,

etc.

Problems with a subsystem are indicated by error messages. *Rjeinit* checks for obstacles in bringing up RJE. If an obstacle is found, an error message indicating the obstacle is printed on the error output. If a problem is encountered during normal operation, the message is logged in the errlog file. This file, error messages, the output from *snoop*, and the checklist above should be used to determine and fix any subsystem problems. Generally, if a subsystem is set up properly but will not operate, the problem is the way the VPM or KMC has been set up, the remote system, or the hardware.

7.4 KMC/VPM Problems

This section describes the KMC and VPM uses, and problems that can occur. After installing KMC hardware and making KMC devices, all VPM software and devices must be made (see *vpm*(4)). The program *rjeload* links the devices to be used by the corresponding RJE subsystem.

The following is a list of items to check when problems occur:

1. Proper hardware — the line unit must be compatible with the modem and have the proper settings (see Section 2.1). Be sure that the KMC address and interrupt vector are correct.
2. Proper Devices — the major and minor device numbers for the KMC and VPM devices must be correct. It should also be verified that the *rjeload* program is called with the correct KMC and VPM device names.
3. Script runs — verify that the VPM script is able to run. This is done by tracing the proper VPM with the proper *snoop* program. *Snoop* will print "started" entries for both the KMC and VPM script (see Section 5.1). If no output appears from *snoop* when *rjeload* is executed, either the KMC is not working properly, or the KMC or VPM has not been set up properly (see items 1 and 2). Output of any other type from *snoop* should indicate where the problem is occurring.

7.5 Trace Interpretation

This section describes how to interpret trace output from the *snoop* program, and gives several examples. Section 5.1 describes the format and meaning of trace output lines, and should be read before this section.

Lines with type TR are traces from the VPM script. All others are driver traces and indicate the following:

- CL — activity occurring when the device has been closed.
- OP — activity occurring when the device has been opened.
- RD — read from device occurred.
- WR — write to device occurred.
- RR — a receive buffer has been returned.
- RX — a transmit buffer has been returned.
- ST — start or stop activity.
- SC — script exit type, exit value is given.

Section 5.1 enumerates all possible trace lines for each type, and describes the event. The remainder of this section consists of example trace output and its interpretation. Comments describing events will appear after the "*" in trace output. If more than one VPM were running, sequence numbers might not appear in order. For clarity, example sequences will be in order.

7.5.1 Normal RJE startup

The following is an example of trace output when RJE has been started up. In this case the remote machine responds to the enquiry byte (ENQ). The RJE subsystem signs on to the machine, then follows the handshaking protocol (exchanging ACKs).

```
Tracing vpm0
0    ST     Startup    * KMC started
1    TR     Started    * Script started
2    TR     S-ENQ      * Enquiry byte sent
3    ST     Start      * VPM Driver start
4    OP     Opened     * VPM Device open
5    TR     R-ACK      * Received acknowledgement
6    TR     S-ACK      * Handshaking
7    WR     84 bytes   * Signon record written
8    TR     R-ACK      * Handshaking
9    TR     S-BLK      * Sent signon block
10   TR     R-ACK      * Block acknowledged
11   RX     Buf        * Transmit buffer returned
12   TR     S-ACK      * Handshaking
13   TR     R-ACK      *    .
14   TR     S-ACK      *    .
15   TR     R-ACK      *    .
16   TR     S-ACK      *    .
17   TR     R-ACK      *    .
18   TR     S-ACK      *    .
19   TR     R-ACK      *    .
20   TR     S-ACK      * Handshaking
```

If any jobs had been submitted via the *send* command, or jobs were waiting to be returned, the traces would reflect the transfers rather than handshaking (see Section 7.5.3).

7.5.2 RJE startup—IBM not responding

This example shows trace output when RJE has been started, but does not receive a response from the remote machine. In general, the RJE script will timeout if a response is not received from the remote machine within 3 seconds of the last transmission. When a timeout is detected while starting up, the enquiry byte (ENQ) is retransmitted. This is repeated 6 times before the script gives up. Other timeout responses will be discussed later.

```
Tracing vpm0
86   ST     Startup    * KMC started
87   TR     Started    * Script started
88   TR     S-ENQ      * Enquiry byte sent
89   ST     Start      * VPM Driver start
90   OP     Opened     * VPM device open
91   WR     84 bytes   * Signon record written
92   TR     TIMEOUT    * No response to enquiry
93   TR     S-ENQ      * Enquiry byte sent
94   TR     TIMEOUT    * No response
95   TR     S-ENQ      * Enquiry byte sent
96   TR     TIMEOUT    * No response
97   TR     S-ENQ      * Enquiry byte sent
98   TR     TIMEOUT    * No response
99   TR     S-ENQ      * Enquiry byte sent
```

```
0    TR    TIMEOUT    * No response
1    TR    S-ENQ      * Enquiry byte sent
2    TR    TIMEOUT    * No response
3    RR    Buf        * Receive buffer returned
4    RD    1 bytes    * 1 byte read (error)
5    SC    Exit(0)    * Script exits normally
6    CL    Clean      * Cleanup done
7    ST    Stopped    * KMC stopped
8    CL    Closed     * VPM device closed
```

The above sequence will be repeated approximately every minute until a positive response is received from the host. During that minute the RJE subsystem is dormant, and the *rjestat* command will report that IBM is not responding. When this occurs, either the IBM machine is not available, down, line not started, etc., or there is a communications problem somewhere from where the KMC transmits data to where it receives data. The RJE administrator should first verify that the IBM machine is up, and the communications line has been started. If so, a hardware trace of the communications line should be done to aid in detecting the problem.

7.5.3 Transmitting and Receiving

This example shows trace output from the start of job transmission through its return. For simplicity, only one job is being transmitted and returned.

```
Tracing vpm0
94   TR    R-ACK      * Handshaking
95   TR    S-ACK      *
96   TR    R-ACK      *
97   TR    S-ACK      * Handshaking
98   WR    4 bytes    * Open reader request written
99   TR    R-ACK      * Handshaking
0    TR    S-BLK      * Sent open request block
1    TR    R-OKBLK    * Received block (grant)
2    RX    Buf        * Transmit buffer returned
3    RR    Buf        * Receive buffer returned
4    TR    S-ACK      * Block acknowledged
5    RD    7 bytes    * Read 7 bytes (grant)
6    TR    R-ACK      * Handshaking
7    TR    S-ACK      * Handshaking
8    WR    481 bytes  * First block written
9    WR    470 bytes  * Second block written
10   TR    R-ACK      * Handshaking
11   TR    S-BLK      * First block sent
12   TR    R-ACK      * Block acknowledged
13   RX    Buf        * Transmit buffer returned
14   WR    470 bytes  * Third block written
15   TR    S-BLK      * Second block sent
16   TR    R-OKBLK    * Received block (on reader msg)
17   RX    Buf        * Transmit buffer returned
18   RR    Buf        * Receive buffer returned
19   WR    470 bytes  * Fourth block written
20   RD    66 bytes   * Read 66 bytes (on reader msg)
21   TR    S-BLK      * Third block sent
22   TR    R-ACK      * Block acknowledged
23   RX    Buf        * Transmit buffer returned
24   WR    147 bytes  * Fifth block written
```

```
25  TR   S-BLK      * Fourth block sent
26  TR   R-ACK      * Block acknowledged
27  RX   Buf        * Transmit buffer returned
 .                  *
 .                  * More of the same
 .                  *
93  TR   R-ACK      * Handshaking
94  TR   S-ACK      * Handshaking
95  TR   R-OKBLK    * Received block (request)
96  RR   Buf        * Receive buffer returned
97  TR   S-ACK      * Block acknowledged
98  RD   7 bytes    * Read open printer request
99  TR   R-ACK      * Handshaking
0   TR   S-ACK      *     .
1   TR   R-ACK      *     .
2   TR   S-ACK      *     .
3   TR   R-ACK      *     .
4   TR   S-ACK      * Handshaking
5   WR   4 bytes    * Printer grant written
6   TR   R-ACK      * Handshaking
7   TR   S-BLK      * Block sent (grant)
8   TR   R-OKBLK    * First block received
9   RX   Buf        * Transmit buffer returned
10  RR   Buf        * Receive buffer returned
11  TR   S-ACK      * Block acknowledged
12  RD   64 bytes   * Read first block
13  TR   R-OKBLK    * Second block received
14  RR   Buf        * Receive buffer returned
15  TR   S-ACK      * Block acknowledged
16  RD   505 bytes  * Read second block
17  TR   R-OKBLK    * Third block received
18  RR   Buf        * Receive buffer returned
19  TR   S-ACK      * Block acknowledged
20  TR   R-OKBLK    * Fourth block received
21  RR   Buf        * Receive buffer returned
22  TR   S-ACK      * Block acknowledged
23  TR   R-ACK      * Handshaking
24  TR   S-ACK      *     .
25  TR   R-ACK      *     .
26  TR   S-ACK      * Handshaking
27  RD   470 bytes  * Read third block
28  RD   494 bytes  * Read fourth block
29  TR   R-ACK      * Handshaking
30  TR   S-ACK      * Handshaking
 .                  *
 .                  * And so on
 .                  *
```

Requests and grants are part of the multi-leaving protocol. Appendix B of *OS/VS MVS JES2 Logic* (SY24-6000-1) describes this protocol in detail. When jobs are being transmitted and received simultaneously, as in a busier RJE subsystem, much less handshaking is involved. Rather than acknowledging blocks with ACKs, the protocol allows a block to be returned (this implies acknowledgement of the received block). The following example shows trace output at a busy time:

```
tracing vpm0
41   TR    R-OKBLK   * Received block
42   RX    Buf       *
43   RR    Buf       *
44   TR    S-BLK     * Sent block
45   WR    493 bytes *
46   RD    496 bytes *
47   TR    R-OKBLK   * Received block
48   RX    Buf       *
49   RR    Buf       *
50   RD    65 bytes  *
51   WR    4 bytes   *
52   TR    S-BLK     * Sent block
53   TR    R-OKBLK   * Received block
54   RX    Buf       *
55   RR    Buf       *
56   TR    S-BLK     * Sent block
57   WR    493 bytes *
58   RD    7 bytes   *
59   TR    R-OKBLK   * Received block
60   RX    Buf       *
61   RR    Buf       *
62   WR    493 bytes *
63   RD    496 bytes *
64   TR    S-BLK     * Sent block
65   TR    R-OKBLK   * Received block
```

Notice that since there is work to be done on both sides, acknowledgements are implied.

7.5.4 Timeout Error Recovery

This example shows activity resulting from timeouts occurring during normal operation. These timeouts were caused because the remote JES3 system has performance problems, and occasionally does not respond in the required three seconds.

```
Tracing vpm1
27   TR    S-ACK     * Handshaking
28   TR    R-ACK     *    .
29   TR    S-ACK     *    .
30   TR    TIMEOUT   * No response
31   TR    S-NAK     * Not acknowledged
32   TR    TIMEOUT   * No response
33   TR    S-NAK     * Not acknowledged
34   TR    R-ACK     * Response
35   TR    S-ACK     * Handshaking
36   TR    R-ACK     *    .
      .              *    .
      .              *    .
      .              *    .
54   TR    R-ACK     *    .
55   TR    S-ACK     * Handshaking
56   TR    TIMEOUT   * No response
57   TR    S-NAK     * Not acknowledged
58   TR    R-ACK     * Response
59   TR    S-ACK     * Handshaking
```

The response to these timeouts are NAKs (not acknowledged). RJE will respond this way up to six times before giving up and attempting a reboot. At this time *rjestat* would report that there are "Line Errors". NAK is a request to retransmit the previous response.

7.5.5 Communication Line Errors

This example shows trace output from an RJE subsystem that uses a dial-up connection. The phone line is noisy and is prone to dropping.

```
Tracing vpm1
63  TR   S-ACK     * Handshaking
64  TR   R-ACK     *   .
65  TR   S-ACK     *   .
66  TR   R-JUNK    * Noise on the line
67  TR   S-NAK     * Not acknowledged
68  TR   R-ACK     * Recovery
69  TR   S-ACK     *
70  TR   R-ACK     *
71  TR   S-ACK     *
72  TR   TIMEOUT   * Line has dropped
73  TR   S-NAK     * Attempting to recover
74  TR   TIMEOUT   *   .
75  TR   S-NAK     *   .
76  TR   TIMEOUT   *   .
77  TR   S-NAK     *   .
78  TR   TIMEOUT   *   .
79  TR   S-NAK     *   .
80  TR   TIMEOUT   *   .
81  TR   S-NAK     *   .
82  TR   TIMEOUT   *   .
83  TR   S-NAK     *   .
84  RR   Buf       * Receive buffer returned
85  RD   1 bytes   * 1 byte read (error)
86  SC   Exit(0)   * Script exits
87  CL   Clean     * Cleanup
88  ST   Stopped   * KMC Stopped
89  CL   Closed    * VPM device closed
```

The error read in the above sequence causes RJE to reboot and *rjestat* to report line errors. If this type of thing were to occur frequently, a different method of communication should be used.

7.5.6 Error Responses

As seen in the sections above, the response to most errors is to send a NAK. The only exception is when starting up (see Section 7.5.2). Whenever a NAK is received on either side, it indicates that the previous transmission was not properly received. This should be followed by retransmission of the previous data. Generally, NAKs should not occur frequently, and should be followed by recovery. If errors occur frequently or NAKs do not cause recovery, the line should be checked for problems.

On some IBM systems, (e.g., JES2), an I/O error is printed at the system console whenever a NAK is received. These I/O errors can also be helpful in detecting the problem; however, they will not be discussed here as they vary with the system. It is assumed that someone in IBM support can assist if needed.

January 1981

SED

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.

SCCS

Source Code Control System

User's Guide

L. E. Bonanni
C. A. Salemi

Bell Telephone Laboratories, Incorporated

Source Code Control System
User's Guide

LIST OF FIGURES

Source Code Control System User's Guide

L. E. Bonanni

Bell Laboratories
Piscataway, New Jersey 08854

C. A. Salemi

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

The Source Code Control System (SCCS) is a collection of PWB commands that help individuals or projects control and account for changes to files of text (typically, the source code and documentation of software systems). It is convenient to conceive of SCCS as a custodian of files; it allows retrieval of particular versions of the files, administers changes to them, controls updating privileges to them, and records who made each change, when and where it was made, and why. This is important in environments in which programs and documentation undergo frequent changes (because of maintenance and/or enhancement work), inasmuch as it is sometimes desirable to regenerate the version of a program or document as it was before changes were applied to it. Obviously, this could be done by keeping copies (on paper or other media), but this quickly becomes unmanageable and wasteful as the number of programs and documents increases. SCCS provides an attractive solution because it stores on disk the original file and, whenever changes are made to it, stores only the *changes;* each set of changes is called a "delta."

This document, together with relevant portions of [1], is a complete user's guide to SCCS. This manual contains the following sections:

- *SCCS for Beginners:* How to make an SCCS file, how to update it, and how to retrieve a version thereof.
- *How Deltas Are Numbered:* How versions of SCCS files are numbered and named.
- *SCCS Command Conventions:* Conventions and rules generally applicable to all SCCS commands.
- *SCCS Commands:* Explanation of all SCCS commands, with discussions of the more useful arguments.
- *SCCS Files:* Protection, format, and auditing of SCCS files, including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

2. SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a PWB system, create files, and use the text editor [1]. A number of terminal-session fragments are presented below. All of them should be tried: the best way to learn SCCS is to use it.

To supplement the material in this manual, the detailed SCCS command descriptions (appearing in [1]) should be consulted. Section 5 below contains a list of all the SCCS commands. For the time being, however, only basic concepts will be discussed.

2.1 Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *S*CCS *ID*entification string (SID), composed of at most four components, only the first two of which will concern us for now; these are the "release" and "level" numbers, separated by a period. Hence, the first delta is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.19", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

2.2 Creating an SCCS File—The "admin" Command

Consider, for example, a file called "lang" that contains a list of programming languages:

```
c
pl/i
fortran
cobol
algol
```

We wish to give custody of this file to SCCS. The following *admin* command (which is used to *administer* SCCS files) creates an SCCS file and initializes delta 1.1 from the file "lang":

```
admin  −ilang  s.lang
```

All SCCS files *must* have names that begin with "s.", hence, "s.lang". The −i keyletter, together with its value "lang", indicates that *admin* is to create a new SCCS file and *initialize* it with the contents of the file "lang". This initial version is a set of changes applied to the null SCCS file; it is delta 1.1.

The *admin* command replies:

```
No id keywords (cm7)
```

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described in Section 5.1 below. In the following examples, this warning message is not shown, although it may actually be issued by the various command.

The file "lang" should be removed (because it can be easily reconstructed by using the *get* command, below):

```
rm  lang
```

2.3 Retrieving a File—The "get" Command

The command:

```
get  s.lang
```

causes the creation (retrieval) of the latest version of file "s.lang", and prints the following messages:

```
1.1
5 lines
```

This means that *get* retrieved version 1.1 of the file, which is made up of 5 lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file; hence, the file "lang" is created.

The above *get* command simply creates the file "lang" read-only, and keeps no information whatsoever regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the *delta* command (see below), the *get* command must be informed of your intention to do so. This is done as follows:

```
get -e s.lang
```

The -e keyletter causes *get* to create a file "lang" for both reading and writing (so that it may be edited) and places certain information about the SCCS file in another new file, called the *p-file*, that will be read by the *delta* command. The *get* command prints the same messages as before, except that the SID of the version to be created through the use of *delta* is also issued. For example:

```
get -e s.lang
1.1
new delta 1.2
5 lines
```

The file "lang" may now be changed, for example, by:

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

2.4 Recording Changes—The "delta" Command

In order to record within the SCCS file the changes that have been applied to "lang", execute:

```
delta s.lang
```

Delta prompts with:

```
comments?
```

the response to which should be a description of why the changes were made; for example:

```
comments? added more languages
```

Delta then reads the *p-file*, and determines what changes were made to the file "lang". It does this by doing its own *get* to retrieve the original version, and by applying *diff*(1)[1] to the original version and the edited version.

1. All references of the form *name*(*N*) refer to item *name* in command writeup section *N* of [1].

When this process is complete, at which point the changes to "lang" have been stored in "s.lang", *delta* outputs:

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file "s.lang".

2.5 More about the "get" Command

As we have seen:

```
get s.lang
```

retrieves the latest version (now 1.2) of the file "s.lang". This is done by starting with the original version of the file and successively applying deltas (the changes) in order, until all have been applied.

For our example, the following commands are all equivalent:

```
get s.lang

get −r1 s.lang

get −r1.2 s.lang
```

The numbers following the −r keyletter are SIDs (see Section 2.1 above). Note that omitting the level number of the SID (as in the second example above) is equivalent to specifying the *highest* level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the *release* number (first component of the SID) of the delta being made. Since normal, automatic, numbering of deltas proceeds by incrementing the level number (second component of the SID), we must indicate to SCCS that we wish to change the release number. This is done with the *get* command:

```
get −e −r2 s.lang
```

Because release 2 does not exist, *get* retrieves the latest version *before* release 2; it also interprets this as a request to change the release number of the delta we wish to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to *delta* via the *p-file*. *Get* then outputs:

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version *delta* will create. If the file is now edited, for example, by:

```
ed lang
41
/cobol/d
w
35
q
```

and *delta* executed:

```
delta s.lang
comments? deleted cobol from list of languages
```

we will see, by *delta's* output, that version 2.1 is indeed created:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

2.6 The "help" Command

If the command:

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (co1)
```

The string "co1" is a code for the diagnostic message, and may be used to obtain a fuller explanation of that message by use of the *help* command:

```
help co1
```

This produces the following output:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Thus, *help* is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Fuller explanations of almost all SCCS messages may be found in this manner.

3. HOW DELTAS ARE NUMBERED

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree, in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the *release* number when making a delta, to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas, unless specifically changed again. Thus, the evolution of a particular file may be represented as in Figure 1.

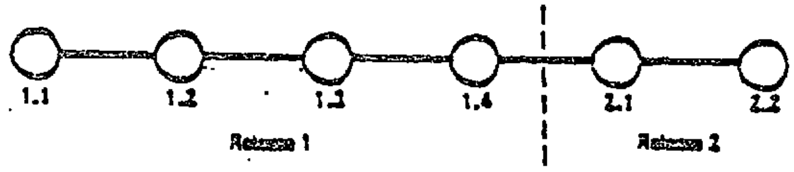


Figure 1. Evolution of an SCCS File

Such a structure may be termed the "trunk" of the SCCS tree. It represents the normal *sequential* development of an SCCS file, in which changes that are part of any given delta are dependent upon *all* the preceding deltas.

However, there are situations in which it is necessary to cause a *branching* in the tree, in that changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3, and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas, precisely as shown in Figure 1. Assume that a production user reports a problem in version 1.3, and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user, but will *not* affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a "branch" of the tree, and its name consists of *four* components, namely, the release and level numbers, as with trunk deltas, plus the "branch" and "sequence" numbers, as follows:

```
release.level.branch.sequence
```

The *branch* number is assigned to each branch that is a descendant of a particular trunk delta, with the first such branch being 1, the next one 2, and so on. The *sequence* number is assigned, in order, to each delta on a *particular branch*. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3. This is shown in Figure 2.

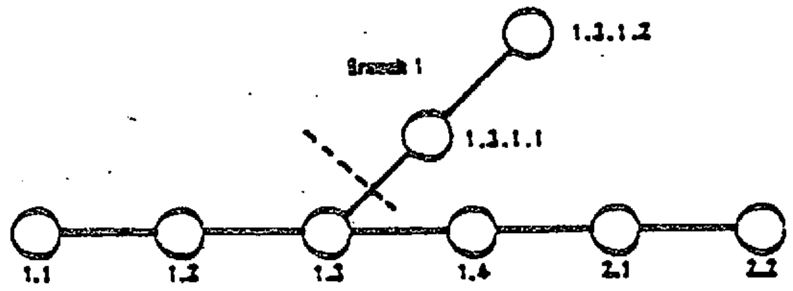


Figure 2. Tree Structure with Branch Deltas

The concept of branching may be extended to any delta in the tree; the naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch, independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is *not* possible to determine the *entire* path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.*n*. If a delta on this branch then has another branch emanating from *it*, all deltas on the new branch will be named 1.3.2.*n* (see Figure 3). The only information that may be derived from the name of delta 1.3.2.2 is that it is the *chronologically* second delta on the *chronologically* second branch whose *trunk* ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

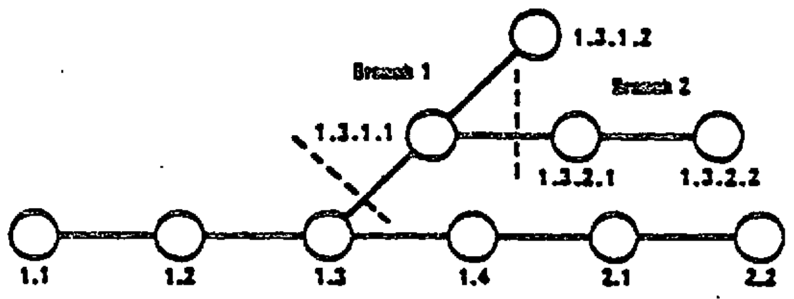


Figure 3. Extending the Branching Concept

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible, because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

4. SCCS COMMAND CONVENTIONS

This section discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to *all* SCCS commands, except as indicated below. SCCS commands accept two types of arguments: *keyletter* arguments and *file* arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (—), followed by a lower-case alphabetic character, and, in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (which may be names of files and/or directories) specify the file(s) that the given SCCS command is to process; naming a directory is equivalent to naming *all* the SCCS files within the directory. Non-SCCS files and unreadable[2] files in the named directories are silently ignored.

In general, file arguments may *not* begin with a minus sign. However, if the name "—" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the *name* of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines [1] with, for example, the *find*(1) or *ls*(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to *all* file arguments of that command. All keyletters are processed before any file arguments, with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments,

2. Because of permission modes (see *chmod*(1)).

however, are processed left to right.

Somewhat different argument conventions apply to the *help*, *what*, *sccsdiff*, and *val* commands (see Sections 5.5, 5.8, 5.9, and 5.11).

Certain actions of various SCCS commands are controlled by *flags* appearing in SCCS files. Some of these flags are discussed below. For a complete description of all such flags, see *admin*(1).

The distinction between the *real user* (see *passwd*(1)) and the *effective user* of a PWB system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a PWB system); this subject is further discussed in Section 6.1.

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file*, which ensures that the SCCS file will not be damaged should processing terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, is given the same mode (see *chmod*(1)) as the SCCS file, and is owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file*, called the *z-file*, whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the *process number* [1] of the command that creates it, and its existence is an indication to other commands that that SCCS file is being updated. Thus, other commands that modify SCCS files will not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file, and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files*; they may be useful in the event of system crashes or similar situations.

SCCS commands produce diagnostics (on the diagnostic output [1]) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The *code* in parentheses may be used as an argument to the *help* command (see Section 5.5) to obtain a further explanation of the diagnostic message.

Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of *that* file and to proceed with the next file, in order, if more than one file has been named.

5. SCCS COMMANDS

This section describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the *PWB User's Manual*, and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

Because the commands *get* and *delta* are the most frequently used, they are presented first. The other commands follow in approximate order of importance.

The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.
admin	Creates SCCS files and applies changes to parameters of SCCS files.
prs	Prints portions of an SCCS file in user specified format.
help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any PWB file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the *get* command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
val	Validates an SCCS file.

5.1 get

The *get* command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version, and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file;* its name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory [1] and is owned by the real user. The mode assigned to the *g-file* depends on how the *get* command is invoked, as discussed below.

The most common invocation of *get* is:

```
get  s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree, and produces (for example) on the standard output [1]:

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that:

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file (see Section 5.1.1 for a discussion of ID keywords).

The generated *g-file* (file "abc") is given mode 444 (read-only), since this particular way of invoking *get* is intended to produce *g-files* only for inspection, compilation, etc., and *not* for editing (i.e., *not* for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example:

```
get s.abc s.def
```

produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)
```

5.1.1 ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc., within the *g-file*, so as to have this information appear in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. *Identification (ID) keywords* appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an upper-case letter enclosed by percent signs (%). For example:

```
%I%
```

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, %H% is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and %M% is defined as the name of the *g-file*. Thus, executing *get* on an SCCS file that contains the PL/I declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by *get*, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by *get*, although the presence of the i flag in the SCCS file causes it to be treated as an error (see Section 5.2 for further information).

For a complete list of the approximately twenty ID keywords provided, see *get*(1).

5.1.2 Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the *trunk* of the SCCS file tree. However, if the SCCS file being processed has a d (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the −r keyletter of *get*.

The −r keyletter is used to specify an SID to be retrieved, in which case the d (default SID) flag (if any) is ignored. For example:

```
get −r1.3 s.abc
```

retrieves version 1.3 of file "s.abc", and produces (for example) on the standard output:

```
1.3
64 lines
```

A branch delta may be retrieved similarly:

```
get −r1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3
234 lines
```

When a two- or four-component SID is specified as a value for the −r keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in:

```
get −r3 s.abc
```

causes retrieval of the *trunk* delta with the highest level number within the given release, if the given release exists. Thus, the above command might output:

```
3.7
213 lines
```

If the given release does not exist, *get* retrieves the *trunk* delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file "s.abc", and that release 7 is actually the highest-numbered release below 9, execution of:

```
get −r9 s.abc
```

might produce:

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file "s.abc" below release 9. Similarly, omission of the sequence number, as in:

```
get −r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch, if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The −t keyletter is used to retrieve the latest ("top") version in a particular *release* (i.e., when no −r keyletter is supplied, or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get −r3 −t s.abc
```

might produce:

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

5.1.3 Retrieval with Intent to Make a Delta

Specification of the −e keyletter to the *get* command is an indication of the intent to make a delta, and, as such, its use is restricted. The presence of this keyletter causes *get* to check:

1. The *user list* (which is the list of *login* names and/or *group IDs* of users allowed to make deltas (see Section 6.2)) to determine if the login name or group ID of the user executing *get* is on that list. Note that a *null* (empty) user list behaves as if it contained *all* possible login names.
2. That the *release* (R) of the version being retrieved satisfies the relation:

   floor ≤ R ≤ ceiling

   to determine if the release being accessed is a protected release. The *floor* and *ceiling* are specified as *flags* in the SCCS file.
3. That the *release* (R) is not *locked* against editing. The *lock* is specified as a flag in the SCCS file.
4. Whether or not *multiple concurrent edits* are allowed for the SCCS file as specified by the j flag in the SCCS file (multiple concurrent edits are described in Section 5.1.5).

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the −e keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a *writable g-file* already exists, *get* terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are *not* substituted by *get* when the −e keyletter is specified, because the generated *g-file* is to be subsequently used to create another delta, and replacement of ID keywords would cause them to be permanently changed within the SCCS file. In view of this, *get* does not need to check for the presence of ID keywords within the *g-file*, so that the message:

```
No id keywords (cm7)
```

is never output when *get* is invoked with the −e keyletter.

In addition, the −e keyletter causes the creation (or updating) of a *p-file*, which is used to pass information to the *delta* command (see Section 5.1.4).

The following is an example of the use of the **−e** keyletter:

```
get −e s.abc
```

which produces (for example) on the standard output:

```
1.3
new delta 1.4
67 lines
```

If the **−r** and/or **−t** keyletters are used together with the **−e** keyletter, the version retrieved for editing is as specified by the **−r** and/or **−t** keyletters.

The keyletters **−i** and **−x** may be used to specify a list (see *get*(1) for the syntax of such a list) of deltas to be *included* and *excluded*, respectively, by *get*. Including a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it to be *not* applied. This may be used to undo, in the version of the SCCS file to be created, the effects of a previous delta. Whenever deltas are included or excluded, *get* checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. (Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*.) Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists, and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

☞ *The* **−i** *and* **−x** *keyletters should be used with extreme care.*

The **−k** keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of *get* with the **−e** keyletter, or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **−k** keyletter is identical to one produced by *get* executed with the **−e** keyletter. However, no processing related to the *p-file* takes place.

5.1.4 Concurrent Edits of Different SIDs

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of *get* commands with the **−e** keyletter may be executed on the same file, provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed, see Section 5.1.5).

The *p-file* (which is created by the *get* command invoked with the **−e** keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, is given mode 644 (readable by everyone, writable only by the owner), and is owned by the effective user. The *p-file* contains the following information for each delta that is still "in progress":[3]

- The SID of the retrieved version.
- The SID that will be given to the new delta when it is created.
- The login name of the real user executing *get*.

The first execution of "get −e" causes the *creation* of the *p-file* for the corresponding SCCS file. Subsequent executions only *update* the *p-file* by inserting a line containing the above information. Before inserting this line, however, *get* checks that no entry already in the *p-file*

3. Other information may be present, but is not of concern here. See *get*(1) for further discussion.

specifies as already retrieved the SID of the version to be retrieved, unless multiple concurrent edits are allowed.

If both checks succeed, the user is informed that other deltas are in progress, and processing continues. If either check fails, an error message results. It is important to note that the various executions of *get* should be carried out from different directories. Otherwise, only the first execution will succeed, since subsequent executions would attempt to over-write a *writable g-file*, which is an SCCS error condition. In practice, such multiple executions are performed by different users,[4] so that this problem does not arise, since each user normally has a different working directory [1].

Table 1 shows, for the most useful cases, what version of an SCCS file is retrieved by *get*, as well as the SID of the version to be eventually created by *delta*, as a function of the SID specified to *get*.

5.1.5 Concurrent Edits of the Same SID

Under normal conditions, *gets* for editing (−e keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, *delta* must be executed before a subsequent *get* for editing is executed at the same SID as the previous *get*. However, multiple concurrent edits (defined to be two or more *successive* executions of *get* for editing based on the same retrieved SID) *are* allowed if the j flag is set in the SCCS file. Thus:

```
get −e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by:

```
get −e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of *delta*. In this case, a *delta* command corresponding to the first *get* produces delta 1.2 (assuming 1.1 is the latest (most recent) trunk delta), and the *delta* command corresponding to the second *get* produces delta 1.1.1.1.

5.1.6 Keyletters That Affect Output

Specification of the −p keyletter causes *get* to write the retrieved text to the standard output, rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names:

```
get −p s.abc > arbitrary-filename
```

The −p keyletter is particularly useful when used with the "!" or "$" arguments of the PWB *send*(1) command. For example:

```
send MOD=s.abc REL=3 compile
```

4. See Section 6.1 for a discussion of how different users are permitted to use SCCS commands on the same files.

TABLE 1. Determination of New SID

Case	*SID Specified**	*—b Keyletter Used†*	*Other Conditions*	*SID Retrieved*	*SID of Delta to be Created*
1.	none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
2.	none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
3.	R	no	R > mR	mR.mL	R.1§
4.	R	no	R = mR	mR.mL	mR.(mL+1)
5.	R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
6.	R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
7.	R	—	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB+1).1
8.	R	—	Trunk successor in release > R and R exists	R.mL	R.mL.(mB+1).1
9.	R.L	no	No trunk successor	R.L	R.(L+1)
10.	R.L	yes	No trunk successor	R.L	R.L.(mB+1).1
11.	R.L	—	Trunk successor in release ≥ R	R.L	R.L.(mB+1).1
12.	R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
13.	R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
14.	R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
15.	R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
16.	R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB+1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus 1) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

† The —b keyletter is effective only if the b flag (see *admin*(1)) is present in the file. In this table, an entry of "—" means "irrelevant".

‡ This case applies if the d (default SID) flag is *not* present in the file. If the d flag *is* present in the file, then the SID obtained from the d flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

§ This case is used to force the creation of the *first* delta in a *new* release.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

if file "compile" contains:

```
//plicomp job job-card-information
//step1 exec plickc
//pli.sysin dd *
~—s
~!get —p —rREL MOD
/*
//
```

will *send* the highest level of release 3 of file "s.abc". Note that the line "~—s", which causes *send*(1) to make ID keyword substitutions before detecting and interpreting control lines, is necessary if *send*(1) is to substitute "s.abc" for MOD and "3" for REL in the line "~!get —p —rREL MOD".

The −s keyletter suppresses all output that is *normally* directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent non-diagnostic messages from appearing on the user's terminal, and is often used in conjunction with the −p keyletter to "pipe" the output of *get*, as in:

```
get −p −s s.abc | nroff
```

The −g keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute:

```
get −g −r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file, or it generates an error message, if it does not. Another use of the −g keyletter is in regenerating a *p-file* that may have been accidentally destroyed:

```
get −e −g s.abc
```

The −l keyletter causes the creation of an *l-file*, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory, with mode 444 (read-only), and is owned by the real user. It contains a table (whose format is described in *get*(1)) showing which deltas were used in constructing a particular version of the SCCS file. For example:

```
get −r2.3 −l s.abc
```

generates an *l-file* showing which deltas were applied to retrieve version 2.3 of the SCCS file. Specifying a *value* of "p" with the −l keyletter, as in:

```
get −lp −r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. Note that the −g keyletter may be used with the −l keyletter to suppress the actual retrieval of the text.

The −m keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The −n keyletter causes each line of the generated *g-file* to be preceded by the value of the %M% ID keyword (see Section 5.1.1) and a tab character. The −n keyletter is most often used in a pipeline with *grep*(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get −p −n −s directory | grep pattern
```

If both the −m and −n keyletters are specified, each line of the generated *g-file* is preceded by the value of the %M% ID keyword and a tab (this is the effect of the −n keyletter), followed by the line in the format produced by the −m keyletter. Because use of the −m keyletter and/or the −n keyletter causes the contents of the *g-file* to be modified, such a *g-file* must *not* be used for creating a delta. Therefore, neither the −m keyletter nor the −n keyletter may be specified together with the −e keyletter.

See *get*(1) for a full description of additional *get* keyletters.

5.2 delta

The *delta* command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and, therefore, a new version of the file.

Invocation of the *delta* command requires the existence of a *p-file* (see Sections 5.1.3 and 5.1.4). *Delta* examines the *p-file* to verify the presence of an entry containing the user's login name. If none is found, an error message results. *Delta* also performs the same permission checks that *get* performs when invoked with the —e keyletter. If all checks are successful, *delta* determines what has been changed in the *g-file*, by comparing it (via *diff*(1)) with its own, temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal *get* at the SID specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing *delta*, because the user who retrieved the *g-file* must be the one who will create the delta. However, if the login name of the user appears in more than one entry (i.e., the same user executed *get* with the —e keyletter more than once on the same SCCS file), the —r keyletter must be used with *delta* to specify an SID that uniquely identifies the *p-file* entry[5]. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of *delta* is:

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a newline character. The user's response may be up to 512 characters long, with newlines *not* intended to terminate the response escaped by "\".

If the SCCS file has a v flag, *delta* first prompts with:

```
MRs?
```

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR[6] numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?".

The —y and/or —m keyletters are used to supply the commentary (comments and MR numbers, respectively) on the command line, rather than through the standard input. For example:

```
delta —y"descriptive comment" —m"mrnum1 mrnum2" s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The —m keyletter is allowed only if the SCCS file has a v flag. These keyletters are useful when *delta* is executed from within a *Shell procedure* (see *sh*(1)).

The commentary (comments and/or MR numbers), whether solicited by *delta* or supplied via keyletters, is recorded as part of the entry for the delta being created, and applies to *all* SCCS files processed by the same invocation of *delta*. This implies that if *delta* is invoked with more than one file argument, and the first file named has a v flag, all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

5. The SID specified may be either the SID retrieved by *get*, or the SID *delta* is to create.

6. In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc. (collectively called here Modification Requests, or MRs) and that it is desirable or necessary to record such MR number(s) within each delta.

When processing is complete, *delta* outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by *delta* do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and *delta* is likely to find a description that differs from the user's perception. However, the *total* number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If, in the process of making a delta, *delta* finds no ID keywords in the edited *g-file*, the message:

```
No id keywords (cm7)
```

is issued after the prompts for commentary, but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values, or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a *get* without the −e keyletter (recall that ID keywords are replaced by *get* in that case), or by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file may never have had any ID keywords. In any case, it is left up to the user to determine what remedial action is necessary, but the delta is made, unless there is an I flag in the SCCS file, indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*.[7] If there is only *one* entry in the *p-file*, then the *p-file* itself is removed.

In addition, *delta* removes the edited *g-file*, unless the −n keyletter is specified. Thus:

```
delta −n s.abc
```

will keep the *g-file* upon completion of processing.

The −s ("silent") keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the −s keyletter together with the −y keyletter (and possibly, the −m keyletter) causes *delta* neither to read the standard input nor to write the standard output.

The differences between the *g-file* and the *d-file* (see above), which constitute the delta, may be printed on the standard output by using the −p keyletter. The format of this output is similar to that produced by *diff*(1).

5.3 admin

The *admin* command is used to *administer* SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

7. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file*, which is described in Section 4 above.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files, and are discussed in Section 6.3 below.

Newly-created SCCS files are given mode 444 (read-only) and are owned by the effective user.

Only a user with write permission in the directory containing the SCCS file may use the *admin* command upon that file.

5.3.1 Creation of SCCS Files

An SCCS file may be created by executing the command:

```
admin −ifirst s.abc
```

in which the value ("first") of the −i keyletter specifies the name of a file from which the text of the *initial* delta of the SCCS file "s.abc" is to be taken. Omission of the value of the −i keyletter indicates that *admin* is to read the standard input for the text of the initial delta. Thus, the command:

```
admin −i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message:

```
No id keywords (cm7)
```

is issued by *admin* as a warning. However, if the same invocation of the command also sets the i flag (not to be confused with the −i keyletter), the message is treated as an error and the SCCS file is not created. Only *one* SCCS file may be created at a time using the −i keyletter.

When an SCCS file is created, the *release* number assigned to its first delta is normally "1", and its *level* number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The −r keyletter is used to specify the release number to be assigned to the first delta. Thus:

```
admin −ifirst −r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the −i keyletter.

5.3.2 Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (−y keyletter) and/or MR numbers[8] (−m keyletter) in exactly the same manner as for *delta*. If comments (−y keyletter) are omitted, a comment line of the form:

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (−m keyletter), the v flag must also be set (using the −f keyletter described below). The v flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a *delta commentary* (see *sccsfile*(5)) in the SCCS file. Thus:

```
admin −ifirst −mmrnum1 −fv s.abc
```

Note that the −y and −m keyletters are only effective if a new SCCS file is being created.

8. The creation of an SCCS file may sometimes be the direct result of an MR.

5.3.3 Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for *descriptive text* (see Section 6.2) may be initialized or changed through the use of the −t keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file, although its contents may be arbitrary, and it may be arbitrarily long.

When an SCCS file is being created and the −t keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command:

```
admin −ifirst −tdesc s.abc
```

specifies that the descriptive text is to be taken from file "desc".

When processing an *existing* SCCS file, the −t keyletter specifies that the descriptive text (if any) currently in the file is to be *replaced* with the text in the named file. Thus:

```
admin −tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of "desc"; omission of the file name after the −t keyletter as in:

```
admin −t s.abc
```

causes the *removal* of the descriptive text from the SCCS file.

The *flags* (see Section 6.2) of an SCCS file may be initialized and changed, or deleted through the use of the −f and −d keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See *admin*(1) for a description of all the flags. For example, the i flag specifies that the warning message stating there are no ID keywords contained in the SCCS file should be treated as an error, and the d (default SID) flag specifies the default version of the SCCS file to be retrieved by the *get* command. The −f keyletter is used to set a flag and, possibly, to set its value. For example:

```
admin −ifirst −fi −fmmodname s.abc
```

sets the i flag and the m (module name) flag. The value "modname" specified for the m flag is the value that the *get* command will use to replace the %M% ID keyword. (In the absence of the m flag, the name of the *g-file* is used as the replacement for the %M% ID keyword.) Note that several −f keyletters may be supplied on a single invocation of *admin*, and that −f keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The −d keyletter is used to delete a flag from an SCCS file, and may only be specified when processing an existing file. As an example, the command:

```
admin −dm s.abc
```

removes the m flag from the SCCS file. Several −d keyletters may be supplied on a single invocation of *admin*, and may be intermixed with −f keyletters.

SCCS files contain a list (*user list*) of login names and/or group IDs of users who are allowed to create deltas (see Sections 5.1.3 and 6.2). This list is empty by default, which implies that *anyone* may create deltas. To add login names and/or group IDs to the list, the −a keyletter is used. For example:

```
admin −axyz −awql −a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The −a keyletter may be used whether *admin* is creating a new SCCS file or processing an existing one, and may appear several times. The −e keyletter is used in an analogous manner if one wishes to remove ("erase") login names or group IDs from the list.

5.4 prs

Prs is used to print on the standard output all or parts of an SCCS file (see Section 6.2) in a format, called the output *data specification*, supplied by the user via the −d keyletter. The data specification is a string consisting of SCCS file *data keywords*[9] interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example:

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, :F: is defined as the data keyword for the SCCS file name currently being processed, and :C: is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see *prs*(1).

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example:

```
prs −d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output:

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the −r keyletter. For example:

```
prs −d":F:: :I: comment line is: :C:" −r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the −r keyletter is *not* specified, the value of the SID defaults to the most recently created delta.

In addition, information from a *range* of deltas may be obtained by specifying the −l or −e keyletters. The −e keyletter substitutes data keywords for the SID designated via the −r keyletter and all deltas created *earlier*. The −l keyletter substitutes data keywords for the SID designated via the −r keyletter and all deltas created *later*. Thus, the command:

```
prs −d:I: −r1.4 −e s.abc
```

may output:

```
1.4
1.3
1.2.1.1
1.2
1.1
```

9. Not to be confused with *get ID keywords*.

and the command:

```
    prs −d:I: −r1.4 −l s.abc
```

may produce:

```
    3.3
    3.2
    3.1
    2.2.1.1
    2.2
    2.1
    1.4
```

Substitution of data keywords for *all* deltas of the SCCS file may be obtained by specifying both the −e and −l keyletters.

5.5 help

The *help* command prints explanations of SCCS commands and of messages that these commands may print. Arguments to *help*, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, *help* prompts for one. *Help* has no concept of *keyletter* arguments or *file* arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will *not* terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example:

```
    help ge5 rmdel
```

produces:

```
    ge5:
    "nonexistent sid"
    The specified sid does not exist in the
    given file.
    Check for typos.

    rmdel:
       rmdel −rSID name ...
```

5.6 rmdel

The *rmdel* command is provided to allow *removal* of a delta from an SCCS file, though its use should be reserved for those cases in which incorrect, global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree. In Figure 3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, then deltas 1.3.2.1 and 2.1 can be removed, and so on.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed, or be the owner of the SCCS file and its directory.

The −r keyletter, which is mandatory, is used to specify the *complete* SID of the delta to be removed (i.e., it must have two components for a trunk delta, and four components for a branch delta). Thus:

```
rmdel −r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, *rmdel* checks that the *release* number (R) of the given SID satisfies the relation:

floor $\leq$ R $\leq$ ceiling

Rmdel also checks that the SID specified is *not* that of a version for which a *get* for editing has been executed and whose associated *delta* has not yet been made. In addition, the login name or group ID of the user must appear in the file's *user list*, or the *user list* must be empty. Also, the release specified can not be *locked* against editing (i.e., if the l flag is set (see *admin*(1)), the release specified *must* not be contained in the list). If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the *delta table* of the SCCS file (see Section 6.2) is changed from "D" (for "delta") to "R" (for "removed").

5.7 cdc

The *cdc* command is used to *change* a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the *rmdel* command, except that the delta to be processed is *not* required to be a leaf delta. For example:

```
cdc −r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The *new* commentary is solicited by *cdc* in the same manner as that of *delta*. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing *cdc* and the time of its execution.

Cdc also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc −r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

5.8 what

The *what* command is used to find identifying information within *any* PWB file whose name is given as an argument to *what*. Directory names and a name of "−" (a lone minus sign) are *not* treated specially, as they are by other SCCS commands, and no *keyletters* are accepted by the command.

What searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the %Z% ID keyword (see *get*(1)), and prints (on the standard output) what follows that string until the first double quote ("), greater than (>), backslash (\), newline, or (non-printing) NUL character. Thus, for example, if the SCCS file "s.prog.c" (which is a C program), contains the following line (the %M% and %I% ID keywords were defined in Section 5.1.1):

```
char id[] "%Z%%M%:%I%";
```

and then the command:

```
get -r3.4 s.prog.c
```

is executed, and finally the resulting *g-file* is compiled to produce "prog.o" and "a.out", then the command:

```
what prog.c prog.o a.out
```

produces:

```
prog.c:
    prog.c:3.4
prog.o:
    prog.c:3.4
a.out:
    prog.c:3.4
```

The string searched for by *what* need not be inserted via an ID keyword of *get*; it may be inserted in any convenient manner.

5.9 sccsdiff

The *sccsdiff* command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the −r keyletter, whose format is the same as for the *get* command. The two versions *must* be specified as the first two arguments to this command in the order in which they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the *pr*(1) command (which actually prints the differences) and must appear before any file names. SCCS files to be processed are named last. Directory names and a name of "−" (a lone minus sign) are *not* acceptable to *sccsdiff*.

The differences are printed in the form generated by *diff*(1). The following is an example of the invocation of *sccsdiff*:

```
sccsdiff -r3.4 -r5.6 s.abc
```

5.10 comb

Comb generates a *Shell procedure* (see *sh*(1)) which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated Shell procedure is written on the standard output.

Named SCCS files are reconstructed by discarding unwanted deltas and combining specified other deltas. The intended use is for those SCCS files that contain deltas that are so old that they are no longer useful. It is *not* recommended that *comb* be used as a matter of routine; its use should be restricted to a *very* small number of times in the life of an SCCS file.

In the absence of any keyletters, *comb* preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate "middle" deltas on the trunk and on all branches of the tree. Thus, in Figure 3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated. Some of the keyletters are summarized as follows:

The −p keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The −c keyletter specifies a *list* (see *get*(1) for the syntax of such a list) of deltas to be preserved. All other deltas are discarded.

The −s keyletter causes the generation of a Shell procedure, which, when run, produces *only* a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that *comb* be run with this keyletter (in addition to any others desired) *before* any actual reconstructions.

It should be noted that the Shell procedure generated by *comb* is *not* guaranteed to save any space. In fact, it is possible for the reconstructed file to be *larger* than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

5.11 val

Val is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

Val checks for the existence of a particular delta when the SID for that delta is *explicitly* specified via the −r keyletter. The string following the −y or −m keyletter is used to check the value set by the t or m flag respectively (see *admin* (1) for a description of the flags).

Val treats the special argument "−" differently from other SCCS commands (see Section 4). This argument allows *val* to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end-of-file. This capability allows for one invocation of *val* with different values for the keyletter and file arguments. For example:

```
val −
−yc −mabc s.abc
−mxyz −ypl1 s.xyz
```

first checks if file "s.abc" has a value "c" for its *type* flag and value "abc" for the *module name* flag. Once processing of the first file is completed, *val* then processes the remaining files, in this case "s.xyz", to determine if they meet the characteristics specified by the keyletter arguments associated with them.

Val returns an 8-bit code which is a disjunction of the possible errors detected. That is, each bit set indicates the occurrence of a specific error (see *val*(1) for a description of the possible errors and their codes). In addition, an appropriate diagnostic is printed unless suppressed by the −s keyletter. A return code of "0" indicates all named files met the characteristics specified.

6. SCCS FILES

This section discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

6.1 Protection

SCCS relies on the capabilities of the PWB operating system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the *release lock* flag, the *release floor* and *ceiling* flags, and the *user list* (see Section 5.1.3).

New SCCS files created by the *admin* command are given mode 444 (read only). It is recommended that this mode *not* be changed, as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755, which allows only the *owner* of the directory to modify its contents.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files (see Section 6.3). The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

SCCS files must have only *one* link (name). The reason for this is that those commands that modify SCCS files do so by creating a temporary copy of the file (called the *x-file*, see Section 4) and, upon completion of processing, remove the old file and rename the *x-file*. If the old file has more than one link, removing it and renaming the *x-file* would break the link. Rather than process such files, SCCS commands produce an error message. All SCCS files *must* have names that begin with "s.".

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files[10]. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (for example, in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the *admin* command). This user is termed the *SCCS administrator* for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the *get*, *delta*, and, if desired, *rmdel* and *cdc* commands.

The interface program must be owned by the SCCS administrator, and must have the *set user ID on execution* bit on (see *chmod*(1)), so that the effective user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to *inherit* the privileges of the interface program for the duration of that command's execution. In this manner, the owner of an SCCS file can modify it at will. Other users whose *login* names or *group* IDs are in the *user list* for that file (but who are *not* its owners) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of *delta* and, possibly, *rmdel* and *cdc*. The project-dependent interface program, as its name implies, must be custom-built for each project.

6.2 Format

SCCS files are composed of lines of ASCII text[11] arranged in six parts, as follows:

Checksum	A line containing the "logical" sum of all the characters of the file (*not* including this checksum itself).
Delta Table	Information about each delta, such as its type, its SID, date and time of creation, and commentary.
User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.

10. Previously, the Operating System under which SCCS executed allowed for only 256 unique user IDs. This presented the situation in which several users needed to share user IDs (and thus shared identical file permissions). The Operating System currently in use (Version 7 of UNIX) allows for 65,536 unique user IDs, and it is recommended that each user have a unique user ID.

11. Previous versions of SCCS up to and including Version 3 used non-ASCII files. Therefore, files created by earlier versions of SCCS are incompatible with the current version of SCCS.

Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in *sccsfile*(5); the *checksum* is the only portion of the file which is of interest below.

It is important to note that because SCCS files are ASCII files, they may be processed by various PWB commands, such as *ed*(1), *grep*(1), and *cat*(1). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly), or when it is desired to simply "look" at the file.

☞ *Extreme care should be exercised when modifying SCCS files with non-SCCS commands.*

6.3 Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file, or portions of it (i.e., one or more "blocks") can be destroyed. SCCS commands (like most PWB commands) issue an error message when a file does not exist. In addition, SCCS commands use the *checksum* stored in the SCCS file to determine whether a file has been *corrupted* since it was last accessed (possibly by having lost one or more blocks, or by having been modified with, for example, *ed*(1)). *No* SCCS command will process a corrupted SCCS file except the *admin* command with the −h or −z keyletters, as described below.

It is recommended that SCCS files be audited (checked) for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the *admin* command with the −h keyletter on all SCCS files:

```
admin −h s.file1 s.file2 ...
   or
admin −h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message:

```
corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not detect *missing* files. A simple way to detect whether *any* files are missing from a directory is to periodically execute the *ls*(1) command on that directory, and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner in which the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local PWB operations group to request a restoral of the file from a backup copy. In the case of minor damage, repair through use of the editor *ed*(1) may be possible. In the latter case, after such repair, the following command must be executed:

```
admin −z s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption which may have existed in that file will no longer be detectable.

REFERENCES

[1] Bell Laboratories, *Documents for Use with the PWB Time-Sharing System.*

ADDENDUM

The following changes to the Source Code Control System are effective with the UNIX™ System III release.

1. Modified commands

Three SCCS commands have been modified:

1. comb
2. get
3. sccsdiff

Modifications to each of these commands are described below.

1.1 comb

- enhancement

  Comb generates a shell procedure that, when executed, will (hopefully) reduce the size of an SCCS file. Because of temporary file naming conventions, two or more *comb* generated shell procedures could not be executed concurrently. Temporary files are now uniquely named so that simultaneous execution is possible.

1.2 get

- enhancement

  Previously the -*i* and -*x* keyletters (for forced inclusion or exclusion of deltas to produce the generated file) would imply the -*k* keyletter. That is, the generated file would be created with mode 644 and identification keyword replacement would be suppressed. The -*i* and -*x* keyletters no longer imply the -*k* keyletter.

- coding error correction

  Under certain circumstances, temporary files that should only have existed for the duration of the execution of *get* would not be removed when *get* terminated. Temporary files are now properly removed.

1.3 sccsdiff

- new capability

  A new keyletter (-_s_), which takes a numeric argument, allows the user to specify the file segmentation size that _bdiff_(1) (used by _sccsdiff_) will pass to _diff_(1). This can be useful when a high system load causes _diff_ to fail due to lack of space, etc.

- change

  The output of _sccsdiff_ is no longer piped through _pr_(1) by default. A new keyletter (-_p_) specifies that the output is to be piped through _pr_ but arguments can not be passed to _pr_ as was the previous case. This alleviates _sccsdiff_ knowing anything about _pr_.

2. New Commands

Two new commands have been added to SCCS:

sact	print current SCCS file editing activity
unget	undo the effect of a previous _get_(1) for editing of an SCCS file.

The manual entries for these commands are provided in the UNIX™ System III User's Manual

Function and Use of an SCCS Interface Program

L. E. Bonanni
A. Guyton (4/1/80 revision)

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

This memorandum discusses the use of a Source Code Control System Interface Program to allow more than one user to use SCCS commands upon the same set of files.

1. INTRODUCTION

In order to permit UNIX† users with different user identification numbers (user IDs) to use SCCS commands upon the same files, an SCCS interface program is provided to temporarily grant the necessary file access permissions to these users. This memorandum discusses the creation and use of such an interface program. This memorandum replaces an earlier version dated March 1, 1978.

2. FUNCTION

When only one user uses SCCS, the real and effective user IDs are the same, and that user ID owns the directories containing SCCS files. However, there are situations (for example, in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the owner of the SCCS files and be the one who will administer them (e.g., by using the *admin* command). This user is termed the *SCCS administrator* for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the *get*, *delta*, and, if desired, *rmdel*, *cdc*, and *unget* commands.[1]

The interface program must be owned by the SCCS administrator, must be executable by non-owners, and must have the *set user ID on execution* bit on (see *chmod*(1)[2]), so that, when executed, the *effective* user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to *inherit* the privileges of the SCCS administrator for the duration of that command's execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose *login* names are in the *user list*[3] for that file (but who are *not* its owners) are given the necessary permissions only for the duration of the execution of the interface program, and are thus able to modify the SCCS files only through the use of *delta* and, possibly, *rmdel* and *cdc*.

† UNIX is a trademark of Bell Laboratories.

1. Other SCCS commands either do not require write permission in the directory containing SCCS files or are (generally) reserved for use only by the administrator.

2. All references of the form *name*(*N*) refer to item *name* in section *N* of the *UNIX User's Manual*.

3. This is the list of login names of users who are allowed to modify an SCCS file by adding or removing deltas. The login names are specified using the *admin*(1) command.

3. A BASIC PROGRAM

When a UNIX program is executed it is passed (as argument 0) the *name* by which it is invoked, followed by any additional user-supplied arguments. Thus, if a program is given a number of *links* (names), it may alter its processing depending upon which link is used to invoke it. This mechanism is used by an SCCS interface program to determine which SCCS command it should subsequently invoke (see *exec*(2)).

A generic interface program (**inter.c**, written in C) is shown in *Attachment I*. Note the reference to the (unsupplied) function **filearg**. This is intended to demonstrate that the interface program may also be used as a pre-processor to SCCS commands. For example, function **filearg** could be used to modify file arguments to be passed to the SCCS command by supplying the *full* path name of a file, thus avoiding extraneous typing by the user. Also, the program could supply any additional (default) keyletter arguments desired.

4. LINKING AND USE

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program **inter.c** resides in directory **/x1/xyz/sccs**. Thus, the command sequence:

```
cd /x1/xyz/sccs
cc ... inter.c −o inter ...
```

compiles **inter.c** to produce the executable module **inter** (... represents arguments that may also be required). The proper mode and the *set user ID on execution* bit are set by executing:

```
chmod 4755 inter
```

Finally, new links are created, by (for example):[4]

```
ln inter get
ln inter delta
ln inter rmdel
```

Subsequently, *any* user whose shell parameter PATH (see *sh*(1)) specifies that directory **/x1/xyz/sccs** is to be searched first for executable commands, may execute, for example:

```
get −e /x1/xyz/sccs/s.abc
```

from any directory to invoke the interface program (via its link *get*). The interface program then executes **/usr/bin/get** (the actual SCCS *get* command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname **/x1/xyz/sccs**, so that the user would only have to specify:

```
get −e s.abc
```

to achieve the same results.

5. CONCLUSION

An SCCS interface program is used to permit users having different user IDs to use SCCS commands upon the same files. Although this is its primary purpose, such a program may also be used as a pre-processor to SCCS commands since it can perform operations upon its arguments.

4. The names of the links may be arbitrary, provided the interface program is able to determine from them the names of SCCS commands to be invoked.

Attachment I

SCCS Interface Program **inter.c**

```
main(argc, argv)
int argc;
char *argv[];
{
        register int i;
        char cmdstr[LENGTH]

        /*
        Process file arguments (those that don't begin with "-").
        */
        for (i = 1; i < argc; i++)
                if (argv[i][0] != '-')
                        argv[i] = filearg(argv[i]);

        /*
        Get "simple name" of name used to invoke this program
        (i.e., strip off directory-name prefix, if any).
        */
        argv[0] = sname(argv[0]);

        /*
        Invoke actual SCCS command, passing arguments.
        */
        sprintf(cmdstr, "/usr/bin/%s", argv[0]);
        execv(cmdstr, argv);
}
```