# User's Guide for the Ratfor Preprocessor

## Second Edition

T. Allen Akin
Terrell L. Countryman
Perry B. Flinn
Daniel H. Forsyth, Jr.
Jeanette T. Myers
Arnold D. Robbins
Peter N. Wan

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

July, 1984

TABLE OF CONTENTS

Ratfor Language Guide

## Ratfor Language Reference



## Ratfor Programming Under the Subsystem

## Appendixes

## Foreword

Ratfor ("Rational Fortran") is an extension of Fortran-66 that serves as the basis for the Software Tools Subsystem. It provides a number of enhancements to Fortran that facilitate structured design and programming, as well as enhance program readability and ease the burden of program coding.

This guide is intended to explain and demonstrate the use of Ratfor as a programming language within the Software Tools Subsystem. In addition, applications notes are provided to help users build on the experience of others.

# Ratfor Language Guide

## What is Ratfor?

The Ratfor ("Rational Fortran") language was introduced in the book Software Tools by Brian W. Kernighan and P. J. Plauger (Addison-Wesley, 1976). There, the authors use it as the medium for the development of programs that may be used as cooperating tools. Ratfor offers many extensions to Fortran that encourage and facilitate structured design and programming, enhance program readability and ease the burden of coding. Through some very simple mechanisms, Ratfor helps the programmer to isolate machine and implementation dependent sections of his code.

Among the many programs developed in Software Tools is a Ratfor preprocessor -- a program for converting Ratfor into equivalent ANSI-66 Fortran. 'Rp', the preprocessor described in this guide, is an original version based on the program presented in Software Tools.

## Differences Between Ratfor and Fortran

As we mentioned, Ratfor and Fortran are very similar. Perhaps the best introduction to their differences is given by Kernighan and Plauger in Software Tools:

> "But bare Fortran is a poor language indeed for programming or describing programs. . . . Ratfor provides modern control flow statements like those in PL/I, Cobol, Algol, or Pascal, so we can do structured programming properly. It is easy to read, write and understand, and readily translates into Fortran. . . . Except for a handful of new statements like **if - else, while,** and **repeat - until,** Ratfor is Fortran."

### Source Program Format

Case Sensitivity. In most cases, the format of Ratfor programs is much less restricted than that of Fortran programs. Since the Software Tools Subsystem encourages use of terminals with multi-case capabilities, 'rp' accepts input in both upper and lower case. 'Rp' is case sensitive. Keywords, such as **if** and **select,** must appear in lower case. Case is significant in identifiers; they may appear in either case, but upper case letters are not equivalent to lower case letters. For example, the words "blank" and "Blank" do not represent the same identifier. For circumstances in which case sensitivity is a bother, 'rp' accepts a command line option ("-m") that instructs it to ignore

the case of all identifiers and keywords. See the applications notes or the 'help' command for more details.

Blank Sensitivity. Unlike most Fortran compilers, 'rp' is very sensitive to blanks. 'Rp' requires that all words be separated by at least one blank or special character. Words containing imbedded blanks are not allowed. The best rule of thumb is to remember that if it is incomprehensible to you, it is probably incomprehensible to 'rp.' (Remember, we humans normally leave blank spaces between words and tend not to place blanks inside words. Such things make text difficult to understand.)

As a bad example, the following Ratfor code is incorrect and will not be interpreted properly:

```
subroutineexample(a,b,c)
integera,b,c

repeatx=x+1
   until(x>1)
```

A few well placed blanks will have to be added before 'rp' can understand it:

```
subroutine example(a,b,c)
integer a,b,c

repeat x=x+1
   until(x>1)
```

You should note that extra spaces are allowed (and encouraged) everywhere except inside words and literals. Extra spaces make a program much more readable by humans:

```
subroutine example (a, b, c)
integer a, b, c

repeat x = x + 1
   until (x > 1)
```

Card Columns. As should be expected of any interactive software system, 'rp' is completely insensitive to "card" columns; statements may begin and end at any position in a line. Lines may be of any length, but identifiers and quoted strings may not be longer than 100 characters. 'Rp' will output all statements beginning in column 7, and automatically generate continuation lines for statements extending past column 72. All of the following are valid Ratfor statements, although such erratic indentation is definitely frowned upon.

```
integer i, j
   i = 1
      j = 2
stop
   end
```

Multiple Statements per Line. 'Rp' also allows multiple statements per line, although indiscriminate use of this feature is not encouraged. Just place a semicolon between statements and 'rp' will generate two Fortran statements from them. You will find

```
integer i
real a
logical l
```

to be completely equivalent to

```
integer i; real a; logical l
```

Statement Labels and Continuation. You may wonder what happens to statement labels and continuation lines, since 'rp' pays no attention to card columns. It turns out that statement labels and continuation lines are not often necessary. While 'rp' minimizes the need for statement labels (except on **format** statements) and is quite intelligent about continuation lines, there are conventions to take care of those situations where a label is required or the need for a continuation line is not obvious to 'rp.'

A statement may be labeled simply by placing the statement number, starting in any column, before the statement. Any executable statement, including the Ratfor control statements, may be labeled, and 'rp' will place the label correctly in the Fortran output. It is wise to refrain from using five-digit statement numbers; 'rp' uses these statement labels to implement the Ratfor control statements, and consequently will complain if it encounters them in a source program. As examples of statement labels,

```
2     read (1, 10) a, b, c
                10  format (3e10.0)
      write (1, 20) a, b, c; 20 format (3f20.5)
         go to 2
```

all show statement numbers in use. You should note that with proper use of Ratfor and the Software Tools Subsystem support subroutines, statement labels are almost never required.

As for continuation lines, 'rp' is usually able to recognize when the current line needs to be continued. A line ending with a comma, unbalanced parentheses in a condition, or a missing statement (such as at the end of an **if)** are all situations in which 'rp' correctly anticipates a continuation line:

```
integer a, b, c, d,
   e, f, g

if (a == b & c == d & e == f &
    g == h & i == j & k == l)  call eql

if (a == b)

   c = -2
```

If an explicit continuation is required, such as in a long assignment statement, 'rp' can be made to continue a line by placing a trailing underscore ("_") at the end of the line. This underscore must be preceded by a space. You should note that the underscore is placed on the end of line to be continued, rather than on the continuation line as in Fortran. If you are unsure whether Ratfor will correctly anticipate a continuation line, go ahead and place an underscore on the line to be continued -- 'rp' will ignore redundant continuation indicators.

Identifiers may not be split between lines; continuation is allowed only between tokens. If you have an extremely long string constant that requires continuation, you can take advantage of the fact that 'rp' always concatenates two adjacent string constants. Just close the first part of the literal with a quote, space, and underscore, and begin the second part on the next line with a quote. 'Rp' will ignore the line break (because of the trailing underscore) and concatenate the two literals.

The following are some examples of explicit line continuations:

```
i = i + j + k + l + m + n + o + p + q + r + _
   s + t + u + v

1 format ("for inputs of ", i5, " and ", i5/ _
      "the expected output should be ", i5)

string heading   _
   "---------------------------------------------"  _
   "---------------------------------------------"
```

Comments. Comments, an important part of any program, can be entered on any line; a comment begins with a sharp sign ("#") and continues until the end of the line. In addition, blank lines and lines containing only comments may be freely placed in the source program. Here are some appropriate and (correct but) inappropriate uses of Ratfor comments:

```
if (i > 48)
   # do this only if i is greater than 48
   j = j + 1

data array / 1,     # element 1
             2,     # element 2
             3,     # element 3
             4/     # element 4

integer cnt,          # counter for controlling the
                      #    outer loop
        total_errs,   # total number of errors
                      #    encountered
        last_pass     # flag for determining the
                      #    last pass; init = 0
```

## Identifiers

A major difference between Ratfor and Fortran is Ratfor's acceptance of arbitrarily long identifiers. A Ratfor identifier may be up to 100 characters long, beginning with a letter, and may contain letters, digits, dollar signs, and underscores. However, it may not be a Ratfor or Fortran keyword, such as **if, else, integer, real,** or **logical.** Underscores are allowed in identifiers only for the sake of readability, and are always ignored. Thus, "these_tasks" and "the_set_asks" are equivalent Ratfor identifiers.

'Rp' guarantees that an identifier longer than six characters will be transformed into a unique Fortran identifier. Normally, the process of transforming Ratfor identifiers into Fortran identifiers is transparent; you need not be concerned with how this transformation is accomplished. The one notable exception is the effect on external symbols (i.e. subroutine and function names, common block names). When the declaration of a subprogram and its invocation are preprocessed together, in the same run, no problems will occur. However, if the subprogram and its invocation are preprocessed separately, there is no guarantee that a given Ratfor name will be transformed into the same Fortran name in the two different runs. This situation can be avoided in either of three ways: (1) use the **linkage** statement described in the next section, (2) use six-character or shorter identifiers for subprogram names, or (3) preprocess subprograms and their invocations in the same run.

Just for pedagogical reasons, here are a few correct and incorrect Ratfor identifiers:

Correct

```
long_name_1
long_name_2
prwf$$
I_am_a_very_long_Ratfor_name_that_is_perfectly_correct
a_a     # You should note that 'a_a', 'a__a', and 'aa'
a__a    # are all absolutely identical in Ratfor --
aa      # underscores are always ignored in identifiers,
AA      # but 'AA' is very different.
```

Incorrect

```
123_part    # starts with a digit
_part1      # starts with an underscore
part 2      # contains a blank
a*b         # contains an asterisk
```

The following paragraph contains a description of exactly how Ratfor identifiers are transformed into Fortran identifiers. You need not know how this transformation is accomplished to make full use of Ratfor; hence, you probably need not read the next paragraph.

If a Ratfor identifier is longer than six characters or contains an upper case letter, it is made unique by the following procedure:

(1) The identifier is padded with 'a's or truncated to five characters. Remaining characters are mapped to lower case.
(2) The first character is retained to preserve implicit typing.
(3) The sixth character is changed to a "uniquing character" (normally a zero).
(4) If necessary, the second, third, fourth, and fifth characters are altered to make sure there is no conflict with a previously used identifier.

'Rp' also examines six-character identifiers containing the uniquing character in the sixth position, to ensure that no conflicts arise.

## Integer Constants

Since it is sometimes necessary to use other than decimal integer constants in a program, 'rp' accepts integers in bases 2 through 16. Integers consisting of only digits are, of course, considered decimal integers. Other bases can be indicated with the following notation:

<base>r<number>

where <base> is the base of the number (in decimal) and <number> is number in the desired base (the letters 'a' through 'f' are

used to represent the digits '10' through '15' in bases greater than 10). For example, here are some Ratfor integer constants and the decimal values they represent:

| Number | Decimal Value |
|---|---|
| 8r77 | 63 |
| 16rff | 255 |
| -2r11 | -3 |
| 7r13 | 10 |

Some care must be exercised when using this form of constant to generate bit-masks with the high-order bit set. For example, to set the high-order bit in a 16-bit word, one might be tempted to use one of the constants

16r8000 or 8r100000

Either of these would cause incorrect results, because the value that they represent, in decimal, is 65536. This number, when encountered by Prime Fortran, is converted to a 32-bit constant (with the high order bit in the second word set). This is probably not the desired result. The only solutions to this problem (which occurs when trying to represent a negative twos-complement number as a positive number) are (1) use the correct twos-complement representation (-32768 in this case), or (2) fall back to Prime Fortran's octal constants (e.g. :100000).

### String Constants

Under the Software Tools Subsystem, character strings come in various flavors. Because various internal representations are used for character strings, Fortran Hollerith constants are not sufficient to easily provide all the different formats required.

All types of Ratfor string constants consist of a string body followed by a string format indicator. The body of a string constant consists of strings of characters bounded by pairs of quotes (either single or double quotes), possibly separated by blanks. All the character strings in the body (not including the bounding quotes) are concatenated to give the value of the string constant. For example, here are three string constant bodies that contain the same string:

```
"I am a string constant body"
"I" ' am ' "a" ' string ' "constant" ' body'
"I am a string "'constant body'
```

The string format indicator is an optional letter that determines the internal format to be used when storing the string. Currently there are five different string representations available:

omitted Fortran Hollerith string. When the string format indicator is omitted, a standard Fortran Hollerith constant is generated. Characters are left-justified, packed in words (two characters per word on the Prime), and unused positions on the right are filled with blanks.

c Single character constant. The 'c' string format indicator causes a single character constant to be generated. The character is right-justified and zero-filled on the left in a word. Only one character is allowed in the body of the constant. Since it is easy to manipulate and compare characters in this format, it is the preferred format for all single characters in the Software Tools Subsystem.

p Packed (Hollerith) period-terminated string. The 'p' format indicator causes the generation of a Fortran Hollerith constant containing the characters in the string body followed by a period. In addition, all periods in the string body are preceded by an escape character ("@"). The advantage of a "p" format string over a Fortran Hollerith string is that the length of the "p" format string can be determined at run time.

v PL/I character varying string. For compatibility with Prime's PL/I and because this data format is required by some system calls, the "v" format indicator will generate Fortran declarations to create a PL/I character varying string. The first word of the constant contains the number of characters; subsequent words contain the characters of the string body packed two per word. "V" format string constants may only be used in executable statements.

s EOS-terminated unpacked string. The "s" string format indicator causes 'rp' to generated declarations necessary to construct an array of characters containing each character in the string body in a separate word, right-justified and zero-filled (each character is in the same format as is generated by the "c" format indicator). Following the characters is a word containing a value different from any character value that marks the end of the string. This ending value is defined as the symbolic constant EOS. EOS-terminated strings are the preferred format for multi-character strings in the Subsystem, and are used by most Subsystem routines dealing with character strings. "S" format string constants may only be used in executable statements.

Here are some examples of strings and the result that would be generated for Prime Fortran. On a machine with a different character set or word length, different code might be generated.

| String Constant | Resulting Code |
|---|---|
| 'v'c | the integer constant 246 |
| "=doc="s | an integer array of length 6 containing 189, 228, 239, 227, 189, 0 |
| "a>b c>d"v | an integer array containing 7, "a>", "b ", "c>", "d " |
| ".main."p | the constant 9h@.main@.. |
| "Hollerith" | the constant 9hHollerith |

## Logical and Relational Operators

Ratfor allows the use of graphic characters to represent logical and relational operators instead of the Fortran ".EQ." and such. While use of these graphic characters is encouraged, it is not incorrect to use the Fortran operators. The following table shows the equivalent syntaxes:

| Ratfor | Fortran | Function |
|---|---|---|
| > | .GT. | Greater than |
| >= | .GE. | Greater or equal |
| < | .LT. | Less than |
| <= | .LE. | Less or equal |
| == | .EQ. | Equal to |
| ~= | .NE. | Not equal to |
| ~ | .NOT. | Logical negation |
| & | .AND. | Logical conjunction |
| \| | .OR. | Logical disjunction |

Note than the digraphs shown in the table must appear in the Ratfor program with no imbedded spaces.

For example, the two following **if** statements are equivalent in every way:

```
if (a .eq. b .or. .not. (c .ne. d .and. f .ge. g))

if (a == b | ~ (c ~= d & f >= g))
```

In addition to graphics representing Fortran operators, two additional operators are available in any logical expression parsed by 'rp' (i.e. anywhere but assignment statements). These operators, '&&' ("and if") and '||' ("or if") perform the same action as the logical operators '&' and '|', except that they guarantee that the expression is evaluated from left to right, and that evaluation is terminated when the truth value of the expression is known. They may appear within the scope of the '~' operator, but they may not grouped within the scope of '&' and

'|'.

These operators find use in situations in which it may be illegal or undesirable to evaluate the right-hand side of a logical expression based on the truth value of the left-hand side. For example, in

```
while (i > 0 && str (i) == ' 'c)
   i = i - 1
```

it is necessary that the subscript be checked before it is used. The order of evaluation of Fortran logical expressions is not specified, so in this example, it would be technically illegal to use '&' in place of '&&'. If the value of 'i' were less than 1, the illegal subscript reference might be made regardless of the range check of the subscript. The Ratfor short-circuited logical operators prevent this problem by insuring that "i > 0" is evaluated first, and if it is false, evaluation of the expression terminates, since its value (false) is known.

## Assignment Operators

Ratfor provides shorthand forms for the Fortran idioms of the form

```
<variable> = <variable> <operator> <expression>
```

In Ratfor, this assignment can be simplified to the form

```
<variable> <assignment operator> <expression>
```

with the use of assignment operators. The following assignment operators are available:

| Operator | Use | Result |
|---|---|---|
| += | <v> += <e> | <v> = <v> + (<e>) |
| -= | <v> -= <e> | <v> = <v> - (<e>) |
| *= | <v> *= <e> | <v> = <v> * (<e>) |
| /= | <v> /= <e> | <v> = <v> / (<e>) |
| %= | <v> %= <e> | <v> = mod (<v>, <e>) |
| &= | <v> &= <e> | <v> = and (<v>, <e>) |
| \|= | <v> \|= <e> | <v> = or (<v>, <e>) |
| ^= | <v> ^= <e> | <v> = xor (<v>, <e>) |

The Ratfor assignment operators may be used wherever a Fortran assignment statement is allowable. Regrettably, the assignment operators provide only a shorthand for the programmer; they do not affect the efficiency of the object code.

The assignment operators are especially useful with subscripted variables; since a complex subscript expression need appear only once, there is no possibility of mistyping or forgetting to change one. Here are some examples of the use of assignment operators

```
i += 1
fact *= i + 10
subs (2 * i - 2, 5 * j - 23) -= 1
int %= 10 ** j
mask &= 8r12
```

For comparison, here are the same assignments without the use of assignment operators:

```
i = i + 1
fact = fact * (i + 10)
subs (2*i-2, 5*j-23) = subs (2*i-2, 5*j-23) - 1
int = mod (int, (10 ** j))
mask = and (mask, 8r12)
```

## Fortran Statements in Ratfor Programs

Ratfor provides the escape statement to allow Fortran statements to be passed directly to the output without the usual processing, such as case mapping and automatic continuation. The escape statement has three forms, summarized below. In the first form listed below, the first non-blank character of the Fortran statement is output in column seven. In the second form, the first non-blank character of the Fortran statement is output in column seven, but column six contains a "$" to continue a previous Fortran statement to that stream. In the third form, the Fortran statement is output starting in column one, so that the user has full control of the placement of items on the line. The following is a summary of this description:

| Escape Statement Format | Output Column |
|---|---|
| %<stream><Fortran statement> | 7 |
| %<stream>&<Fortran statement> | 6 |
| %<stream>%<Fortran statement> | 1 |

"Stream" can take on the following values:

| | |
|---|---|
| 1 | declaration |
| 2 | data |
| 3 | code |

If no stream is specified (i.e. %%<Fortran statement>), the Fortran statement is sent to the code stream.

Escaped statements must occur inside a program unit, i.e., between a **function** or **subroutine** statement, and its corresponding **end** statement. Otherwise 'rp' gets confused about where the escaped statements should go, since it won't have any streams open. If you have a large amount of self contained FORTRAN that you want 'rp' to include in its output, you can accomplish this in two steps. First, put '%1%' at the beginning of each line, and then put the FORTRAN at the beginning of your ratfor source file.

## Incompatibilities

Even with the great similarities between Fortran and Ratfor, an arbitrary Fortran program is not necessarily a correct Ratfor program. Several areas of incompatibilities exist:

- In Ratfor, blanks are significant -- at least one space must separate adjacent identifiers.

- The Ratfor **do** statement, as we shall soon see, does not contain the statement number following the "do". Instead, its range extends over the next (possibly compound) statement.

- Two word Fortran key phrases such as **double precision, block data,** and **stack header** must be presented as a single Ratfor identifier (e.g. "blockdata" or "block_data").

- Fortran statement functions must be preceded by the Ratfor keyword **stmtfunc.** To assure that they will appear in the correct order in the Fortran, they should immediately precede the **end** statement for the program unit.

- Hollerith literals (i.e. 5HABCDE) are not allowed anywhere in a Ratfor program. Instead, 'rp' expects all Hollerith literals to be enclosed in single or double quotes (i.e. "ABCDE" or 'ABCDE'). 'Rp' will convert the quoted string into a proper Fortran Hollerith string.

- 'Rp' does not allow Fortran comments. In Ratfor, comments are introduced by a sharp sign ("#") appearing anywhere on a line, and continue to the end of the line.

- 'Rp' does not accept the Fortran continuation convention. Continuation is implicit for any line ending with a comma, or any conditional statement containing unbalanced parentheses. Continuation between arbitrary words may be indicated by placing an underscore, preceded by at least one space, at the end of the line to be continued.

- 'Rp' does not ignore text beyond column 72.

- Fortran and Ratfor keywords may not be used as identifiers in a Ratfor program. Their use will result in unreasonable behavior.

## Ratfor Text Substitution Statements

'Rp' provides several text substitution facilities to improve the readability and maintainability of Ratfor programs. You can use these facilities to great advantage to hide tedious implementation details and to assist in writing transportable code.

### Define

The Ratfor **define** statement bears a vague similarity to the non-standard Fortran **parameter** declaration, but is much more flexible. In Ratfor, any legal identifier may be defined as almost any string of characters. Thereafter, 'rp' will replace all occurrences of the defined identifier with the definition string. In addition, identifiers may be defined with a formal parameter list. Then, during replacement, actual parameters specified in the invocation are substituted for occurrences of the formal parameters in the replacement text.

Defines find their principle use in helping to clarify the meaning of "magic numbers" that appear frequently. For example,

```
while (getlin (line, -10) ~= -1)
   call putlin (line, -11)
```

is syntactically correct, and even does something useful. But what? The use of **define** to hide the magic numbers not only allows them to be changed easily and uniformly, but also gives the program reader a helpful hint as to what is going on. If we rewrite the example, replacing the numbers by defined identifiers, not only are the numbers easier to change uniformly at some later date, but also, the reader is given a little bit of a hint as to what is intended.

```
define (EOF, -1)
define (STANDARD_INPUT, -10)
define (STANDARD_OUTPUT, -11)

while (getlin (line, STANDARD_INPUT) ~= EOF)
   call putlin (line, STANDARD_OUTPUT)
```

The last example also shows the syntax for definitions without formal parameters.

Often there are situations in which the replacement text must vary slightly from place to place. For example, let's take the last situation in which the programmer must supply "STANDARD_INPUT" and "STANDARD_OUTPUT" in calls to the line input and output routines. Since this occurs in a large majority of cases, it would be more convenient to have procedures named, say "getl" and "putl" that take only one parameter and assume "STANDARD_INPUT" or "STANDARD_OUTPUT". We could, of course,

write two new procedures to fill this need, but that would add more code and more procedure calls. Two **define** statements will serve the purpose very well:

```
define (STANDARD_INPUT, -10)
define (STANDARD_OUTPUT, -11)
define (getl (ln), getlin (ln, STANDARD_INPUT))
define (putl (ln), putlin (ln, STANDARD_OUTPUT))

while (getl (line) ~= EOF)
   call putl (line)
```

In this case, when the string "getl (line)" is replaced, all occurrences of "ln" (the formal parameter) will be replaced by "line" (the actual parameter). This example will give exactly the same results as the first, but with a little less typing when "getl" and "putl" are called often.

The full syntax for a **define** statement follows:

```
define (<identifier> [(<formal params>)], <replacement>)
```

When such a **define** statement is encountered, <replacement> is recorded as the value of <identifier>. At any later time, if <identifier> is encountered in the text, it is replaced by the text of <replacement>. If the original **define** contained a formal parameter list, the list of actual parameters following <identifier> is collected, and the actual parameters are substituted for the corresponding formal parameters in <replacement> before the replacement is made.

There is a file of "standard" definitions used by all Subsystem programs called "=incl=/swt_def.r.i". The **define** statements in this file are automatically inserted before each source file (unless 'rp' is told otherwise by the "-f" command line option). For information on the exact contents of this file, see Appendix D.

There are also a few other facts that are helpful when using **define:**

- The <replacement> may be any string of characters not containing unbalanced parentheses or unpaired quotes

- <Formal parameters> must be identifiers.

- <Actual parameters> may be any string of characters not containing unbalanced parentheses, unpaired quotes, or commas not surrounded by quotes or parentheses.

- Formal parameter replacement in <replacement> occurs even inside of quoted strings. For example,

```
define (assert (cond), {
   if (~(cond))
      call error ("assertion cond not valid"p)}
assert (i < j)
```

would generate

```
{
   if (~(i < j))
      call error ("assertion i < j not valid"p)}
```

- During replacement of an identifier defined without a formal parameter list, an actual parameter list will never be accessed. For example,

```
define (ARRAYNAME, table1)
ARRAYNAME (i, j) = 0
```

would generate

```
table1 (i, j) = 0
```

- The number of actual and formal parameters need not match. Excess formal parameters will be replaced by null strings; excess actual parameters will be ignored.

- A **define** statement affects only those identifiers following it. In the following example, STDIN would **not** be replaced by -11, unless a **define** statement for STDIN had occurred previously:

```
l = getlin (buf, STDIN)
define (STDIN, -11)
```

- A **define** statement applies to all lines following it in the input to 'rp', regardless of subroutine, procedure, and source file boundaries.

- After replacement, the substituted text itself is examined for further defined identifiers. This allows such definition sequences as

```
define (DELCOMMAND, LETD)
define (LETD, 100)
```

to result in the desired replacement of "100" for "DELCOMMAND". Actual parameters are not reexamined until the entire replacement string is reexamined.

- Identifiers may be redefined without error. The most recent definition supersedes all previous ones. Storage space used by superseded definitions is reclaimed.

Here are a few more examples of how defines can be used:

Before Defines Have Been Processed:

```
define (NO, 0)
define (YES, 1)
define (STDIN, -11)
define (EOF, -2)
define (RESET (flag), flag = NO)
define (CHECK_FOR_ERROR (flag, msg),
   if (flag == YES)
      call error (msg)
   )
define (FATAL_ERROR_MESSAGE,
   "Fatal error -- run terminated"p)
define (PROCESS_LINE,
   count = count + 1
   call check_syntax (buf, count, error_flag)
   )

while (getlin (buf, STDIN) ~= EOF) {
   RESET (error_flag)
   PROCESS_LINE
   CHECK_FOR_ERROR (error_flag, FATAL_ERROR_MESSAGE)
   }
```

After Defines Have Been Processed:

```
while (getlin (buf, -11) ~= -2) {
   error_flag = 0
   count = count + 1
   call check_syntax (buf, count, error_flag)
   if (error_flag == 1)
      call error ("Fatal error -- run terminated"p)
   }
```

## Undefine

The Ratfor **undefine** statement allows termination of the range of a **define** statement. The identifier named in the **undefine** statement is removed from the define table if it is present; otherwise, no action is taken. Storage used by the definition is reclaimed. For example, the statements

```
define (xxx, a = 1)
xxx
undefine (xxx)
xxx
```

would produce the following code:

```
a = 1
xxx
```

## Include

The Ratfor **include** statement allows you to include arbitrary files in a Ratfor program (much like the COBOL **copy** verb). The syntax of an **include** statement is as follows:

```
include "<file name>"
```

If the file name is six or fewer characters in length and contains only alphanumeric characters, the quotes may be omitted. For the sake of uniformity, we suggest that the quotes always be used.

When 'rp' encounters an **include** statement, it begins taking input from the file specified by <file name>. When the end of the included file is encountered, 'rp' resumes reading the preempted file. Files named in **include** statements may themselves contain **include** statements; this nesting may continue to an arbitrary depth (which, by the way, is arbitrarily limited to five).

For an example of **include** at work, assume the existence of the following files:

```
f1:
     include "f2"
     i = 1
     include "f3"

f2:
     include "f4"
     m = 1

f3:
     j = 1

f4:
     k = 1
```

If "f1" were the original file, the following text is what would actually be processed:

```
k = 1
m = 1
i = 1
j = 1
```

## Ratfor Declarations

There are several declarations available in Ratfor in addition to those usually supported in Fortran. They provide a way of conveniently declaring data structures not available in Fortran, assist in supporting separate compilation, allow declaration of local variables within compound statements, and allow the declaration of internal procedures. Declarations in Ratfor may be intermixed with executable statements.

### String

The **string** statement is provided as a shorthand way of creating and naming EOS-terminated strings. The structure and use of an EOS-terminated string is described in the section on Subsystem Conventions. Here it is sufficient to say that such a string is an integer array containing one character per element, right justified and zero filled, and ending with a special value (EOS) designating the "end of string." Since Fortran has no construct for specifying such a data structure, it must either be declared manually, as a Ratfor string constant, or by the Ratfor **string** statement.

The **string** statement is a declaration that creates a named string in an integer array using a Fortran **data** statement. The syntax of the **string** statement is as follows:

```
string <name> <quoted string>
```

where <name> is the Ratfor identifier to be used in naming the string and <quoted string> specifies the string's contents. As you might expect, either single or double quotes may be used to delimit <quoted string>. In either case, only the characters between the quotes become part of the string; the quotes themselves are not included.

**String** statements are quite often used for setting up constant strings such as file names or key words. For instance,

```
string file_name "//mydir/myfile"
string change_command "change"
string delete_command "delete"
```

define such character arrays.

### Stringtable

The **stringtable** statement creates a rather specialized data structure -- a marginally indexed array of variable length strings. This data structure provides the same ease of access as an array, but it can contain entries of varying sizes. A **stringtable** declaration defines two data items: a marginal index and a table body. The marginal index is an integer array containing

indices into the table body. The first element of the marginal index is the number of entries following in the marginal index. Subsequent elements of the marginal index are pointers to the beginning of items in the table body. Since the beginning of the table body is always the beginning of an item, the second entry of the marginal index is always 1.

The syntax of a **stringtable** declaration is as follows:

```
string_table <marginal index>, <table body>,
   [ / ] <item> { / <item> }
```

<Marginal index> and <table body> are identifiers that will be declared as the marginal index and table body, respectively. <Item> is a comma-separated list of single-character constants (with a "c" string format indicator), integers, or EOS-terminated character strings (with no string format indicator -- a little inconsistency here). The values contained in an <item> are stored contiguously in <table body> with no separator values (save for an EOS at the end of each EOS-terminated string). An entry is made in the marginal index containing the position of the first word of each <item>.

For example, assume that you have a program in which you wish to obtain one of three integer values based on an input string. You want to allow an arbitrary number of synonyms in the input (like "add", "insert", etc.).

```
string_table cmdpos, cmdtext,
   / ADD,      "add" _
   / ADD,      "insert" _
   / CHANGE,   "change" _
   / CHANGE,   "update" _
   / DELETE,   "delete" _
   / DELETE,   "remove"
```

This declaration creates a structure something like the following:

```
   cmdpos                    cmdtext

1:    6
2:    1          1:  ADD,    'a'c, 'd'c, 'd'c, EOS
3:    6          6:  ADD,    'i'c, 'n'c, 's'c, 'e'c,
                             'r'c, 't'c, EOS
4:   14         14:  CHANGE, 'c'c, 'h'c, 'a'c, 'n'c,
                             'g'c, 'e'c, EOS
5:   22         22:  CHANGE, 'u'c, 'p'c, 'd'c, 'a'c,
                             't'c, 'e'c, EOS
6:   29         29:  DELETE, 'd'c, 'e'c, 'l'c, 'e'c,
                             't'c, 'e'c, EOS
7:   36         36:  DELETE, 'r'c, 'e'c, 'm'c, 'o'c,
                             'v'c, 'e'c, EOS
```

There are several routines in the Subsystem library that can be used to search for strings in one of these structures. You can find details on the use of these procedures in the reference manual/'help' entries for 'strlsr' and 'strbsr'.

## Linkage

The sole purpose of the **linkage** declaration is to circumvent problems with transforming Ratfor identifiers to Fortran identifiers when compiling program modules separately. To relax the restriction that externally visible names (subroutine, function, and common block names) must contain no more than six characters, each separately compiled module must begin with an identical **linkage** declaration containing the names of all external symbols -- subroutine names, function names, and common block names (the identifiers inside the slashes -- not the variable names). Except for text substitution statements, the **linkage** declaration must be the first statement in each module. The order of names in the statement is significant -- as a general rule, you should **include** the same file containing the **linkage** declaration in each module.

**Linkage** looks very much like a Fortran type declaration:

```
linkage identifier1, identifier2, identifier3
```

Each of the identifiers is an external name (i.e. subroutine, function, or common block name). If this statement appears in each source module, with the identifiers in exactly the same order, it is guaranteed that in all cases, each of these identifiers will be transformed into the same unique Fortran identifier. For Subsystem-specific information on the mechanics of separate compilation, you can see the section in the applications notes devoted to this topic.

## Local

With the **local** declaration, you can indicate that certain variables are "local" to a particular compound statement (or block) just as in Algol. **Local** declarations are most often used inside internal procedures (which are described later), but they can appear in any compound statement.

The type declarations for local variables must be preceded by a **local** declaration containing the names of all variables that are to be local to the block:

```
local i, j, a

integer i, j
real a
```

The **local** statement must precede the first appearance of a variable inside the block. While this isn't the greatest syntax

in the world, it is easy to implement local variables in this fashion.

Scope rules similar to those of most block-structured languages apply to nested compound statements: A local variable is visible to all blocks nested within the block in which it is declared. Declaration of a local variable obscures a variable by the same name declared in an outer block.

There are several cautions you must observe when using local variables. 'Rp' is currently not well-versed in the semantics of Fortran declarations and therefore cannot diagnose the incorrect use of **local** declarations. Misuse can then result in semantic errors in the Fortran output that are often not caught by the Fortran compiler. If the declaration of a variable within a block appears before the variable is named in a **local** declaration, 'rp' will not detect the error, and an "undeclared variable" error will be generated in the Fortran. External names (i.e. function, subroutine, and common block names) must never be named in a **local** declaration, unless you want to declare a local variable of the same name. Finally, the formal parameters of internal procedures should never appear in a **local** declaration in the body of the procedure, again, unless you want to declare a local variable of the same name.

Here is an example showing the scopes of variables appearing in a **local** declaration:

```
### level 0
subroutine test

integer i, j, k

{  ### level 1
   local i, m; integer i, m
   # accessible: level 0 j, k; level 1 i, m
   {  ### level 2
      local m, k; real m, k
      # accessible: level 0 j; level 1 i; level 2 m, k
      }
   }

end
```

## Ratfor Control Statements

As was said by Kernighan and Plauger in Software Tools, except for the control structures, "Ratfor is Fortran." The additional control structures just serve to give Fortran the capabilities that already exist in Algol, Pascal, and PL/I.

### Compound Statements

Ratfor allows the specification of a compound statement by surrounding a group of Ratfor statements with braces ("{}"), just like **begin - end** in Algol or Pascal, or **do - end** in PL/I. A compound statement may appear anywhere a single statement may appear, and is considered to be equivalent to a single statement when used within the scope of a Ratfor control statement.

There is normally no need for a compound statement to appear by itself -- compound statements usually appear in the context of a control structure -- but for completeness, here is an example of a compound statement.

```
{     # end of line -- set to beginning of next line
   line = line + 1
   col = 1
   end_of_line = YES
   }
```

### If - Else

The Ratfor **if** statement is much more flexible than its Fortran counterpart. In addition to allowing a compound statement as an alternative, the Ratfor **if** includes an optional **else** statement to allow the specification of an alternative statement. Here is the complete syntax of the Ratfor **if** statement:

```
if (<condition>) <statement1>
[else <statement2>]
```

<Condition> is an ordinary Fortran logical expression. If <condition> is true, <statement1> will be executed. If <condition> is false and the **else** alternative is specified, <statement2> will be executed. Otherwise, if <condition> is false and the **else** alternative has not been specified, no action occurs.

Both <statement1> and <statement2> may be compound statements or may be further **if** statements. In the case of nested **if** statements where one or more **else** alternatives are not specified, each **else** is paired with the most recently occurring **if** that has not already been paired with an **else**.

Although deep nesting of **if** statements hinders understanding, one situation often occurs when it is necessary to select one and only one of a set of alternatives based on several conditions. This can be nicely represented with a chain of **if - else if - else if . . . else** statements. For example,

```
if (color == RED)
   call process_red
else if (color == BLUE | color == GREEN)
   call process_blue_green
else if (color == YELLOW)
   call process_yellow
else
   call color_error
```

could be used to select a routine for processing based on color.

**While**

The Ratfor **while** statement allows the repetition of a statement (or compound statement) as long as a specified condition is met. The Ratfor **while** loop is a "test at the top" loop exactly like the Pascal **while** and the PL/I **do while.** The **while** statement has the following syntax:

```
while (<condition>)
   <statement>
```

If <condition> is false, control passes beyond the loop to the next statement in the program; if <condition> is true, <statement> is executed and <condition> is retested. As should be expected, if <condition> is false when the **while** is first entered, <statement> will be executed zero times.

The **while** statement is very handy for controlling such things as skipping blanks in strings:

```
while (str (i) == BLANK)
   i = i + 1
```

And of course, <statement> may also be a compound statement:

```
while (getlin (buf, STDIN) ~= EOF) {
   call process (buf)
   call output (buf)
   }
```

**Repeat**

The Ratfor **repeat** loop allows repetitive execution of a statement until a specified condition is met. But, unlike the **while** loop, the test is made at the bottom of the loop, so that the controlled statement will be executed at least once. The

**repeat** loop has syntax as follows:

```
repeat
   <statement>
   [until (<condition>)]
```

When the **repeat** statement is encountered, <statement> is executed. If <condition> is found to be false, <statement> is reexecuted and the <condition> is retested. Otherwise control passes to the statement following the **repeat** loop. If the **until** portion of the loop is omitted, the loop is considered an "infinite repeat" and must be terminated within <statement> (usually with a **break** or **return** statement). Pascal users should note that the scope of the Ratfor **repeat** is only a single <statement> (which of course may be compound).

**Repeat** loops, as opposed to **while** loops, are used when the controlled statement must be evaluated at least once. For example,

```
repeat
   call get_next_token (token)
   until (token ~= BLANK_TOKEN)
```

The "infinite repeat" is often useful when a loop must be terminated "in the middle:"

```
repeat {
   call get_next_input (inp)
   call check_syntax (inp, error_flag)
   if (error_flag == NO)
      return
   call syntax_error (inp)    # go back and get another
   }
```

## Do

Ratfor provides access to the Fortran **do** statement. The Ratfor **do** statement is identical to the Fortran **do** except that it does not use a statement label to delimit its scope. The Ratfor **do** statement has the following syntax:

```
do <limits>
   <statement>
```

<Limits> is the normal Fortran notation for the limits of a **do**, such as "i = 1, 10" or "j = 5, 20, 2". The same restrictions apply to <limits> as apply to the limits in the Fortran **do**. <Statement> is any Ratfor statement (which may be compound).

The Ratfor **do** statement is just like the standard Fortran one-trip **do** loop -- <statement> will be executed at least once, regardless of the limits. Also, the value of the **do** control variable is not defined on exit from the loop.

The **do** loop can be used for array initialization and other such things that can never require "zero trips", since it produces slightly more efficient object code than the **for** statement (which we will get to next).

```
do i = 1, 10
   array (i) = 0
```

One slight irregularity in the Ratfor syntax occurs when <statement> appears on the same line as the **do**. Since 'rp' knows very little about Fortran, it assumes that the <limits> continue until a statement delimiter. This means that the <limits> must be followed by a semicolon if <statement> is to begin on the same line. This often occurs when a compound statement is to be used:

```
do i = 1, 10; {
   array_1 (i) = 0
   array_2 (i) = 0
   }
```

**For**

The Ratfor **for** statement is an all-purpose looping construct that takes the best features of both the **while** and **do** statements, while allowing more flexibility. The syntax of the **for** statement is as follows:

```
for (<initialize>; <condition>; <reinitialize>)
   <statement>
```

When the **for** is executed, the statement represented by <initialize> is executed. Then, if <condition> is true, <statement> is executed, followed by the statement represented by <reinitialize>. Then, <condition> is retested, etc. Any or all of <initialize>, <condition>, or <reinitialize> may be omitted; the semicolons, however, must remain. If <initialize> or <reinitialize> is omitted, no action is performed in their place. If <condition> is omitted, an "infinite loop" is assumed. (Both <initialize> or <reinitialize> may be compound statements).

As you can see, the **for** loop with <initialize> and <reinitialize> omitted is identical to the **while** loop. With the addition of <initialize> and <reinitialize>, a zero-trip **do** loop can be constructed. For instance,

```
for (i = 1; i <= 10; i += 1) {
   array_1 (i) = 0
   array_2 (i) = 0
   }
```

is identical to the last **do** example, but given a certain combination of limits, the **for** loop would execute <statement> zero times while the **do** loop would execute it once.

The **for** loop can do many things not possible with a **do** loop, since the **for** loop is not constrained to the ascending incrementation of an index. As an example, assume a list structure in which "list" contains the index of the first item in a list, and the first position in each list item contains the index of the next. The **for** statement could be used to serially examine the list:

```
for (ptr = list; ptr ~= NULL; ptr = array (ptr)){
   [ examine the item beginning at array (ptr + 1) ]
   }
```

**Break**

The **break** statement allows the early termination of a loop. The statement

```
break [<level>]
```

will cause the immediate termination of <level> loops, where <level>, if specified, is an integer in the range 1 to the depth of loop nesting at the point the **break** statement appears. Where <level> is omitted, only the innermost loop surrounding the **break** is terminated.

In the following example, the **break** statement will cause the termination of the inner **for** loop if a blank is encountered in 'str':

```
while (getlin (str, STDIN) ~= EOF) {
   for (i = 1; str (i) ~= EOS; i += 1)
      if (str (i) == BLANK)
         break

   str (i) = EOS        # output just the first word
   call putlin (str, STDOUT)
   call putch (NEWLINE, STDOUT)
   }
```

Replacing the **break** statement with "break 1" would have exactly the same effect. However, replacing it with "break 2" would cause termination of both the inner **for** and outer **while** loops. Unless this fragment is nested inside other loops, a value greater than 2 would be an error.

**Next**

The **next** statement is very similar to the **break** statement, except that a statement of the form

```
next [<level>]
```

causes termination of <level> - 1 nested loops (zero when <level>

is omitted). Execution then resumes with the next iteration of the innermost active loop. <Level>, if specified, is again an integer in the range 1 to the depth of loop nesting that specifies which loop (from inside out) is to begin its next iteration.

In this example, the **next** statement will cause the processing to be skipped when an array element with the value "UNUSED" is encountered.

```
for (i = 1; i <= 10; i += 1)
   for (j = 1; j <= 10; j += 1) {
      if (array (i, j) == UNUSED)
         next

      # process array (i, j)

      }
```

When an array element with the value "UNUSED" is encountered, execution of the **next** statement causes the <reinitialize> portion of the innermost **for** statement, "j += 1", to be executed before the next iteration of the inner loop begins. You should note that when used with a **for** statement, **next** always skips to the <reinitialize> part of the appropriate **for** loop.

If the statement "next 2" had been used in place of "next", the inner **for** loop would have been terminated, and the "i += 1" of the outer **for** loop would have been executed in preparation for its next iteration.

### Return

The Ratfor **return** statement normally behaves exactly like the Fortran **return** statement in all but one case. In this case, Ratfor allows a parenthesized expression to follow the keyword **return** inside a function subprogram. The value of this expression is then assigned to the function name as the value of the function before the return is executed. This is just another shorthand and does not provide any additional functionality.

Normally in a Fortran function subprogram, you place an assignment statement that assigns a value to the function name before the **return** statement, like this:

```
integer function calc (x, y, z)
...
calc = x + y - z
return
...
```

If you like, Ratfor allows you to express the same actions with one line less code:

```
integer function calc (x, y, z)
...
return (x + y - z)
...
```

This segment performs exactly the same function as the preceding segment.

## Select

The Ratfor **select** statement allows the selection of a statement from several alternatives, based either on the value of an integer variable or on the outcome of several logical conditions. A **select** statement of the form

```
select
   when (<expression list 1>)
      <statement 1>
   when (<expression list 2>)
      <statement 2>
   ...
   when (<expression list n>)
      <statement n>
[ifany
   <statement n+1>]
[else
   <statement n+2>]
```

(where <expression list> is a comma-separated list of logical expressions) performs almost the same function as a chain of **if - else if . . . else** statements. Each <logical expression> is evaluated in turn, and when the first true expression is encountered, the corresponding statement is executed. If any **when** alternative is selected, the statement in the **ifany** part is executed. If none of the **when** alternatives are selected, the statement in the **else** part is executed.

Although its function is very similar to an **if - else** chain, a **select** statement has two distinct advantages. First, it allows the "ifany" alternative -- a way to implement a rather frequently encountered control structure without repeated code or procedure calls. Second, it places all the logical expressions in the same basic optimization block, so that even a dumb Fortran compiler can optimize register loads and stores.

For example, assume that we want to check to see if the variable 'color' contains a valid color, namely 'RED', 'YELLOW', 'BLUE', or 'GREEN'. If it does, we want to executed one of the three subroutines 'process_red', 'process_yellow', or 'process_blue_green' and set the flag 'color_valid' to YES. Otherwise, we want to set the 'color_valid' to NO. A **select** statement performs this trick nicely, with no repeated code:

```
select
   when (color == RED)
      call process_red
   when (color == YELLOW)
      call process_yellow
   when (color == BLUE, color == GREEN)
      call process_blue_green
ifany
   color_valid = YES
else
   color_valid = NO
```

The second variant of the select statement allows the selection of a statement based on the value of an integer (or character) expression. It has almost exactly the same syntax as the logical variant:

```
select (<integer expression>)
   when (<expression list 1>)
      <statement 1>
   when (<expression list 2>)
      <statement 2>
   ...
   when (<expression list n>)
      <statement n>
[ifany
   <statement n+1>]
[else
   <statement n+2>]
```

Using this variant, a statement is selected when one of its corresponding integer expressions has the same value as the <integer expression> following the 'select'. The **ifany** and **else** clause behave as they do in the logical variant. The most visible difference, though, is that the order of evaluation of the integer expressions is not specified. If two values in two expression lists are identical, it is difficult to say which of the statements will be executed; it can only be said that one and only one will be executed.

The integer variant offers one further advantage. If elements in the expression lists are integer or single-character constants, 'rp' will generate Fortran computed **goto** statements, rather than Fortran **if** statements, where possible. This code is usually considerably faster and more compact than the code generated by **if** statements.

The example given for the logical variant of **select** would really be much more easily done with the integer variant:

```
select (color)
   when (RED)
      call process_red
   when (YELLOW)
      call process_yellow
   when (BLUE, GREEN)
      call process_blue_green
ifany
   color_valid = YES
else
   color_valid = NO
```

As a final example of **select,** the following program fragment selects an insert, update, delete, or print routine based on the input codes "i", "u", "d" or "p":

```
while (getlin (buf, STDIN) ~= EOF)

   select (buf (1))
      when ('i'c, 'I'c)    # insert record
         call insert_record
      when ('u'c, 'U'c) {  # update record
         call delete_record
         call insert_record
         }
      when ('d'c, 'D'c)    # delete record
         call delete_record
      when ('p'c, 'P'c)    # print record
         ;
   ifany                   # always print after command
      call print_record
   else                    # illegal input
      call command_error
```

This example shows the use of both a compound statement within an alternative (the "update" action deletes the target record and then inserts a new version), and a null statement consisting of a single semicolon.

### Procedure

Procedures are a convenient and useful structuring mechanism for programs, but in Fortran there often reasons for restricting the unbridled use of procedures. Among these reasons are (1) the run-time expense of procedure calls, and argument and common block addressing; (2) external name space congestion; and (3) difficulty in detecting errors in parameter and common-block correspondence. Ratfor attempts to address these problems by allowing declaration of procedures within Fortran subprograms that are inexpensive to call (an assignment and two **gotos),** are not externally visible, and allow access to global variables. In addition, when correctly declared, Ratfor internal procedures can call each other recursively without requiring recursive procedures in the host Fortran.

Currently, Ratfor internal procedures do not provide the same level of functionality as Fortran subroutines and functions: internal procedure parameters must be scalars and are passed by value, internal procedures cannot be used as functions (they cannot return values), and no automatic storage is available with recursive integer procedures. But even with these restrictions, internal procedures can significantly improve the readability and modularity of Ratfor code.

Internal procedures are declared with the Ratfor **procedure** statement. Internal procedures may be declared anywhere in a program, but a declaration must appear before any of its calls. Here is an example of a non-recursive procedure declaration:

```
# putchar --- put a character in the output string
   procedure putchar (ch) {

      character ch

      str (i) = ch
      i += 1
      }
```

This procedure has one parameter, "ch", which must appear in a type declaration inside the procedure.

Internal procedures always exit by falling through the end of the compound statement. A **return** statement in an internal procedure will return from the Fortran subprogram in which the internal procedure is declared.

After the above declaration, "putchar" can be subsequently called in one of two ways:

```
putchar ('='c)

-or-

call putchar ('='c)
```

The second form is preferable, so that a procedure can be converted to a subroutine, and vice-versa. The number of parameters in the call must always match the number of parameters in the declaration. If parameter list is omitted in the declaration, then it also must be omitted in its calls.

If "putchar" were recursive, the declaration would be

```
procedure putchar (ch) recursive 128
```

The value "128" is an integer constant that is the maximum number of recursive calls to "putchar" outstanding at any one time.

Since internal procedures may be mutually recursive, and since they must be declared textually before they are used, procedures may be declared "forward" by separating the procedure

declaration from its body. Here is "putchar" declared using a "forward" declaration:

```
procedure putchar (ch) forward

...

# putchar --- put a character in the output string
   procedure putchar {

      character ch

      str (i) = ch
      i += 1
      }
```

As you can see, the parameters must appear in the "forward" declaration; they may appear in the body declaration, but are ignored. For maximum efficiency, all internal procedures should be presented in a "forward" declaration. The procedure bodies should then be declared after the final **return** or **stop** statement in the body of the Fortran subprogram, but before the terminating **end** statement (then the program never has to jump around the procedure body).

In general, a **procedure** declaration contains five parts: the word "procedure", the procedure name, an optional list of formal parameters, an optional "recursive <integer>" part, and either a compound statement or the word "forward". An internal procedure call consists of three parts: optionally the word "call", the procedure name, and an optional parameter list.

# Ratfor Language Reference

This section contains a summary of the Ratfor syntax and source program format. In addition to serving as a reference for Ratfor, it can also be used by someone who is familiar with Fortran and wants to quickly gain a reading knowledge of Ratfor.

## Differences Between Ratfor and Fortran

### Source Program Format

- 'Rp' is sensitive to letter case. Keywords must appear in lower case. Case is significant in identifiers.

- 'Rp' is blank sensitive in that words (sequences of letters, digits, dollar signs, and underscores) must be separated by special characters or blanks.

- 'Rp' is not sensitive to card columns. Statements may begin at any position on a line.

- 'Rp' allows multiple statements per line by separating the statements with semicolons.

- A Ratfor statement may be labeled by placing the numeric label in front of the statement. The label must be separated from the statement by at least one space.

- 'Rp' will expect a continuation line if it encounters a line ending with a trailing comma, a condition with unbalanced parentheses, a missing statement following a control statement, or a line ending with a trailing underscore.

- Any line may contain a comment. Comments begin with a sharp sign ("#") and continue until the end of the line.

### Identifiers

Ratfor identifiers consist of letters, digits, underscores, dollar signs, and may be up to 100 characters long. An identifier must begin with a letter. Underscores may be included for readability, but are completely ignored. An identifier may not be the same as a Fortran or Ratfor keyword. 'Rp' transforms all long Ratfor identifiers into unique Fortran identifiers.

## Integer Constants

'Rp' allows integer constants of the form "<base>r<number>" where <base> is an integer between 2 and 16. The letters "a" - "f" are used for digits in bases greater than 10.

## String Constants

String constants in Ratfor consist of a string body and a string format indicator. The string body is a group of strings, bounded by quotes, and possibly separated by blanks. The string format indicator designates the data representation to be used for the characters in the string body. It has one of the following values:

omitted — Fortran Hollerith string. A standard Fortran Hollerith constant is generated. Characters are left-justified, packed in words (two characters per word on the Prime), and unused positions on the right are filled with blanks.

c — Single character constant. A single character constant is generated. The character is right-justified and zero-filled on the left in a word. Only one character is allowed in the body of the constant. This is the preferred format for all single characters in the Software Tools Subsystem.

p — Packed (Hollerith) period-terminated string. The 'p' format indicator causes the generation of a Fortran Hollerith constant. All periods in the string body are preceded by an escape character ("@").

v — PL/I character varying string. Fortran declarations are generated to create a PL/I character varying string. "V" format string constants may only be used in executable statements.

s — EOS-terminated unpacked string. Fortran declarations are generated to construct an array in which each element contains one character of the string body, right-justified and zero-filled (each character is in the same format as is generated by the "c" format indicator). Following the characters is a word containing the value EOS. EOS-terminated strings are the preferred format for multi-character strings in the Subsystem. "S" format string constants may only be used in executable statements.

## Logical and Relational Operators

Ratfor allows the use of graphic characters to represent logical and relational operators instead of the Fortran ".EQ." and such. These characters will be replaced by their Fortran

equivalents during preprocessing. The following table shows the equivalent syntaxes:

| Ratfor | Fortran | Function |
|---|---|---|
| > | .GT. | Greater than |
| >= | .GE. | Greater or equal |
| < | .LT. | Less than |
| <= | .LE. | Less or equal |
| == | .EQ. | Equal to |
| ~= | .NE. | Not equal to |
| ~ | .NOT. | Logical negation |
| & | .AND. | Logical conjunction |
| \| | .OR. | Logical disjunction |
| && | (none) | Short-circuited conjunction |
| \|\| | (none) | Short-circuited disjunction |

Note that the digraphs shown in the table must appear in the Ratfor program with no imbedded spaces. The short-circuited operators may appear only in the <condition> part of Ratfor control statements.

### Assignment Operators

Assignment operators provide a shorthand for the common Fortran idiom "<v> = <v> <op> <expr>". Assignment operators may appear anywhere a Fortran assignment statement may appear. The following assignment operators are available in Ratfor:

| Operator | Use | Result |
|---|---|---|
| += | <v> += <e> | <v> = <v> + (<e>) |
| -= | <v> -= <e> | <v> = <v> - (<e>) |
| *= | <v> *= <e> | <v> = <v> * (<e>) |
| /= | <v> /= <e> | <v> = <v> / (<e>) |
| %= | <v> %= <e> | <v> = mod (<v>, <e>) |
| &= | <v> &= <e> | <v> = and (<v>, <e>) |
| \|= | <v> \|= <e> | <v> = or (<v>, <e>) |
| ^= | <v> ^= <e> | <v> = xor (<v>, <e>) |

### Escape Statements

Escape statements can be used to output Fortran statements that will not be touched by the Ratfor preprocessor. The escape statement has three possible forms. In the first form listed below, the first non-blank character of the Fortran statement is output in column seven. In the second form, the first non-blank character of the Fortran statement is output in column seven, but column six contains a "$" to continue a previous Fortran

statement to that stream. In the third form, the Fortran statement is output starting in column one, so that the user has full control of the placement of items on the line. The following is a summary of this description:

| Escape Statement Format | Output Column |
|---|---|
| %<stream><Fortran statement> | 7 |
| %<stream>&<Fortran statement> | 6 |
| %<stream>%<Fortran statement> | 1 |

"Stream" can take on the following values:

| | |
|---|---|
| 1 | declaration |
| 2 | data |
| 3 | code |

If no stream value is given, it is assumed to be the code stream. Escaped statements have to come between a **function** or **subroutine** statement and the corresponding **end** statement.

## Incompatibilities

Even with the great similarities between Fortran and Ratfor, an arbitrary Fortran program is not necessarily a correct Ratfor program. Several areas of incompatibilities exist:

- Blanks are significant -- at least one space or special character must separate adjacent keywords and identifiers.

- The Ratfor **do** statement does not contain a statement number following the "do". Its range always extends over the next statement.

- Two word Fortran key phrases such as **double precision** must be presented as a single Ratfor identifier (e.g. "doubleprecision" or "double_precision").

- Fortran statement functions must be preceded by the Ratfor keyword **stmtfunc.** To assure that they will appear in the correct order in the Fortran, they should immediately precede the **end** statement of the program unit.

- Hollerith literals (i.e. 5HABCDE) are not allowed anywhere in a Ratfor program. Instead, 'rp' expects all Hollerith literals to be enclosed in single or double quotes (i.e. "ABCDE" or 'ABCDE').

- 'Rp' does not allow Fortran comments. Ratfor comments must be introduced by a sharp sign ("#").

- 'Rp' does not accept the Fortran continuation convention. Continuation is implicit for any line ending

with a comma, or any conditional statement containing unbalanced parentheses. Continuation between arbitrary words may be indicated by placing an underscore, preceded by at least one space, at the end of the line to be continued.

- 'Rp' does not ignore text beyond column 72.

- Fortran and Ratfor keywords may not be used as identifiers in a Ratfor program. Their use will result in unreasonable behavior.

## Ratfor Text Substitution Statements

**define (<identifier> [(<formal params>)], <replacement text>)**

When a **define** statement is encountered in a source program, <replacement text> is recorded as the replacement for <identifier>. If <identifier> is encountered later in the program, it will be replaced by <replacement text>. If <formal params> was present in the definition of <identifier>, and the subsequent occurrence of <identifier> is followed by a parenthesized, comma-separated list of strings, occurrences of the formal parameters in <replacement text> will be replaced by the corresponding strings in the actual parameter list.

<Identifier> must be an alphabetic Ratfor identifier, while <replacement text> may contain any characters except unmatched quotes or parentheses. <Formal params> must be a comma-separated list of identifiers; corresponding actual parameters may contain any characters except unmatched quotes, unbalanced parentheses, or unnested commas. During replacement, <replacement text> is also examined for occurrences of **defined** identifiers. Formal parameter replacement occurs on identifiers in <replacement text>, even if the identifiers are surrounded by quotes or parentheses. Redefinition of an <identifier> causes the new <replacement text> to replace the old.

**undefine (<identifier>)**

The **undefine** statement removes the definition of <identifier> from the list of defined identifiers. Subsequent occurrences of <identifier> in the program will not be replaced unless <identifier> appears in a subsequent **define** statement.

**include '<path name>'**

An **include** statement instructs 'rp' to begin taking input from the file specified by <path name>. When the end of the file is reached, 'rp' resumes taking input from the file containing the **include** statement. The path name may be surrounded by either

single or double quotes. The file specified by <path name> may contain further **include** statements, up to a maximum depth of 5.

## Ratfor Declarations

**linkage <identifier> { , <identifier> }**

The **linkage** declaration is used to guarantee that long external names are transformed into the same unique Fortran name. Names are transformed as they are presented in the **linkage** declaration. The same **linkage** statement should appear as the first statement of each separately compiled source module, and should contain the names of all subroutines, functions, and common blocks in the program.

**local <identifier> { , <identifier> }**

The **local** declaration allows the declaration of variables with names local to the scope of a compound statement (block). The **local** declaration should appear inside a compound statement and must precede all occurrences of the identifiers to be declared local to the block. All identifiers appearing in a **local** declaration must subsequently appear in a type declaration in the same compound statement.

**string <name> <quoted string>**

The **string** statement generates declarations to produce an EOS-terminated string in the integer array <name>. <Quoted string> must be surrounded by either single or double quotes.

**stringtable <index>, <body>, [ / ] <item> { / <item }**

The **stringtable** declaration creates a marginally indexed array of integers and character strings. <Index> and <body> are variables to be declared as the index and body arrays respectively. <Body> is a one-dimensional array in which the values generated by the <item>s are stored consecutively. The first element of <index> contains the number of remaining elements in <index>; subsequent elements each contain the index in <body> of the first position of the corresponding <item>.

<Item>s are comma-separated lists of integers, single-character constants, and strings (with no string format indicators). Integers and EOS-terminated strings are generated and stored consecutively in <body>. The first position of each <item> in <body> is stored in the corresponding entry of <index>.

## Ratfor Control Statements

**break [<integer>]**

The **break** statement allows the user to terminate the execution of a for, while, or repeat loop and resume control at the first statement following the loop. The <integer> specifies the number of loops to terminate; if absent, 1 is assumed (only the innermost loop is terminated). If the integer is N, then the N innermost loops currently active are terminated.

**do <limits>; <statement>**

The **do** statement provides a means of accessing the local Fortran do-statement. <Limits> includes whatever parameters are necessary to satisfy Fortran, minus the statement number of the last statement to be performed, which is generated by Ratfor. The semicolon must not be used if the statement to be iterated does not appear on the same line as the **do**.

**for '(' <init>; <condition>; <reinit> ')' <statement>**

The **for** statement is a very general looping construct. <init> is a statement to be executed before loop entry; it is frequently used to initialize a counter. <Condition> is a condition to be satisfied for every iteration; the condition is tested at the top of the loop. <Condition> becoming false is the most often used method of terminating the loop. <Reinit> is a statement to be executed at the bottom of the loop, just before a jump is made to the top to test the <condition>. <Reinit> is usually used to increment or decrement a counter. <Statement> may be any legal Ratfor statement.

**if '(' <condition> ')' <statement> [else <statement>]**

**If** is a generalization of the Fortran logical-if statement. If the condition is true, the first <statement> is executed. If the optional **else** clause is missing, control is then passed to the statement following the **if**; otherwise, the <statement> following the **else** is executed before passing control.

**next [<integer>]**

The **next** statement complements the **break** statement. It is used to force the next iteration of a for, repeat or while loop to occur. The parameter <integer> specifies the number of levels of nested loops to jump out; if omitted, the innermost loop is continued; otherwise, for <integer> = 2, the next-to-innermost loop is continued, etc.

```
procedure <procid> [ '(' <id> {, <id> } ')' ]
      [recursive <integer>]
      ( forward | <compound statement> )

[call] <procid> [ '(' <expr> {, <expr> } ')' ]
```

The **procedure** declaration allows the declaration of internal Ratfor procedures. <Procid> is the name of the internal procedure. Formal parameters (scalar, pass-by-value) are declared following the <procid>. Formal parameters must appear in a type declaration in the body of the procedure. If the procedure is to be called recursively, the **recursive <integer>** clause must be included; <integer> is the maximum number of recursive calls in process at any given time. Following the heading, either a compound statement or the word **forward** must appear. If the **forward** option is used, a **procedure** declaration containing <compound statement> must follow at some point in the program unit. Formal parameters specified on the second declaration may be present, but are ignored.

A <procid> must be defined before it is referenced by a call. The call can appear exactly as a Fortran call, or the word **call** can be omitted. Actual parameters must correspond in number to formal parameters. If the formal parameters list is omitted in the declaration, no actual parameter list may be present.

```
repeat <statement> [until '(' <condition> ')']
```

The **repeat** statement is used to generate a loop with the iteration test at the bottom. The <statement> is performed, then the <condition> checked; if false, the <statement> is repeated. If true, control passes to the statement following the **until.** If the **until** is omitted, the loop is repeated indefinitely, and must be terminated with a **stop, break,** or **goto.**

```
return ['(' <expression> ')']
```

The **return** statement behaves exactly like its Fortran counterpart, except that if the optional parenthesized expression is included inside a function subprogram, the value of <expression> will be assigned to the function name as the function value before the return is executed.

```
select
    {when '(' <condition> {, <condition>} ')' <statement> }
    [ifany <statement>] [else <statement>]

select '(' <integer expr> ')'
    {when '(' <integer expr> {, <integer expr>} ')' <statement>
    [ifany <statement>] [else <statement>]
```

**Select** is a generalization of the **if** statement. In its first alternative, the **when** <conditions>s are evaluated in order;

the <statement> associated with the first one found to be true is executed. If any <condition> is found true, the <statement> associated with **ifany** is executed; if none are found true, the <statement> associated with **else** is executed.

Similarly, in the second alternative, the <integer expr> associated with **select** is evaluated. The result is then compared to the <integer expr>s associated with the **when** parts in an unspecified order. When an equal comparison is made, the <statement> following the corresponding **when** is executed. If an equal comparison is made, the <statement> following **ifany** is executed; if no equal comparison is made, the <statement> following **else** is executed.

**while '(' <condition> ')' <statement>**

The **while** statement is the basic test-at-the-top loop. The <condition> is evaluated; if true, the <statement> is executed and the loop is repeated, otherwise control passes to the statement following the loop.

# Ratfor Programming Under the Subsystem

This chapter describes the use of Ratfor in the programming environment provided by the Software Tools Subsystem. In addition to demonstrating use of the Ratfor preprocessor, Fortran compiler, and linking loader, the programming conventions necessary for the use of the Subsystem support subprograms are described.

In this chapter, a number of programming conventions are presented. Since very few of the conventions can be enforced by the Subsystem, adherence to these conventions must be left to up to the programmer. Many conventions, such as those dealing with indentation and comment placement, are shown because they assist in producing readable, maintainable programs. Violation of these conventions, while not critical, may result in unmaintainable programs and extended debugging times. Other conventions, such as those dealing with character string representations and input/output, are crucial to the proper operation of the Subsystem and its support subprograms. Violation of these conventions can and will cause undesirable results.

## Requirements for Ratfor Programs

The Software Tools Subsystem is not an operating system. Rather, it is a collection of cooperating user programs. To run successfully under the Subsystem, a program must cooperate with it. Several things are required of Subsystem programs:

- The program must terminate with a **stop** statement, or a call to the routine "error". The program must not "call exit" or invoke any of the Primos error reporting subroutines with the the "immediate return" key. A program's failure to terminate properly will also cause the Subsystem command interpreter to be terminated, leaving the user face-to-face with Primos.

- The program should not have initialized common blocks (i.e. **block data**). Initialize the common areas with executable statements. (To link a program that must have initialized common, see appendix b.)

- Local variables in a subprogram are placed on the stack unless they appear in a **data** or **save** declaration. The value of variables not appearing in one of these declarations is not defined on entry to a subprogram.

Several conventions apply to the file containing the Ratfor source statements:

- The file name should end with the suffix ".r".

- Any number of program units (main program, functions, and subroutines) may be included in the file, but the main program must be first.

- All variables and functions must be declared in type statements (the Primos Fortran compiler enforces this restriction, except in the case of function names).

- Each program unit must end with an **end** statement.

- Since **defines** apply globally to all subsequent program units, a main program and all of its associated subprograms can be contained in the same file. Only one copy of definitions need be included at the beginning of the source file.

## Running Ratfor Programs Under the Subsystem

Three steps are required to obtain an executable program from Ratfor source statements. The first step, preprocessing, produces ANSI Fortran statements from the Ratfor source statements. The second step, compilation, results in a relocatable binary module, which lacks all of the Primos, Fortran and Subsystem subroutines. The last step, linking, produces an executable object program by linking the relocatable binary module with the Primos, Fortran and Subsystem support routines necessary for its execution. The object program produced during linking may then be executed.

### Preprocessing

In the preprocessing step, the Ratfor preprocessor, 'rp,' is used to translate Ratfor source statements into semantically equivalent ANSI Fortran statements acceptable to the Primos Fortran compiler. The Ratfor preprocessor is invoked with a command line of the following syntax:

rp [-o <output file>] <input file> [<rp options>]

If you do not want a conventionally named output file, you may specify the option "-o <output file>", where <output file> is the name you want given to the Fortran output. If you do not include a "-o <output file>" option, 'rp' will name the output file by appending ".f" to the name of the first <input file>. If the name of the first <input file> ends in ".r", the ".r" will be replaced by the ".f".

Next comes a list of the files containing Ratfor source statements to be preprocessed. 'Rp' reads the files in the order

specified on the command line and treats the contents as if they were together in one big file. This means that **defines** in each input file apply to all subsequent input files.

Finally, there are preprocessor options which may be specified to change the output in some way or affect preprocessor operation. For a complete list of available options and a more detailed description of the command line syntax, see Appendix F.

In spite of all this complicated stuff, the 'rp' preprocessor is quite easy to use if you follow the recommended naming conventions for files. For instance, if you have a Ratfor program in a file called "prog.r", you can have it preprocessed by just typing

```
rp prog.r
```

This command will cause the program contained in "prog.r" to be preprocessed, and the Fortran output to be produced on the file "prog.f" (which is exactly what the Fortran compiler expects).

Here are some more examples to show other ways in which 'rp' can be called:

```
# preprocess the files "p1.r", "p2.r", and "p3.r"
#    and produce Fortran output on "p1.f"

  rp p1.r p2.r p3.r


# preprocess the files "p1.r", "p2.r", and "p3.r"
#    and produce Fortran output on "ftn_out"

  rp p1.r p2.r p3.r -o ftn_out


# preprocess the file "p1.r", produce the Fortran
#    on "ftn_out" and include code to produce
#    subprogram level trace

  rp -t p1.r -o ftn_out
```

## Compiling

After turning your Ratfor source code into Fortran with the preprocessor, the next step is to compile the Fortran code. Since the Subsystem uses the Primos Fortran compiler, the 'fc' command just produces a sequence of Primos commands to cause the compilation. The following command will call the Fortran compiler for a compilation:

```
fc [<options>] <input> [-b [<binary>]] [-l [<listing>]]
```

The Fortran source code must be in the file <input>. The

relocatable binary output will be placed in the file <binary>, unless "-b <binary>" is omitted. Then, following Subsystem conventions, the binary file name is constructed by appending the input file name with ".b"; if the input file ends with ".f", the "f" will be replaced by the "b". Normally no listing is produced; however, if one is requested, it will appear on the file <listing>, or if the listing file name is omitted, the name will be constructed by appending the ".l" to the input file name; again, if the input file name ends in ".f", the "f" will be replaced with the "l".

<Options> is a series of single letter options that specify how the compiler is to generate the object code. Since there are too many options to completely describe here, we will only mention a few of the more important ones. For those who wish to make full use of the Fortran compiler, or for those just curious, the Software Tools Subsystem Reference Manual, or the 'help' command will give complete information.

Here are brief descriptions of the options of interest:

-v Generate pseudo-assembly code describing the object code produced.

-i Unless otherwise specified, consider all integers to be "long" (32-bit) rather than "short" (16-bit). (This is useful for programs ported from machines with longer word lengths.)

-t Insert code to produce a statement-level trace during execution.

Of course, more than one of these options may be specified.

Again, even though all of this looks very complicated, it is really very simple, if you have used the Subsystem file naming conventions. If you have your Fortran code in a file named "prog.f" (remember where Ratfor put its output), you may compile it, using the default options, by just entering

```
fc prog.f
```

The command will call the Fortran compiler to produce binary output in the file "prog.b". Just for completeness, here are some other examples of 'fc' commands:

```
# Compile "p1.f" to produce the binary "p1.b" and
#    and a listing on "p1.l"

  fc p1.f -l


# Compile "p1.f" to produce the binary "bin" and
#    the listing on "list"

  fc p1.f -b bin -l list


# Compile "p3.f", produce a pseudo-assembly code
#    listing and default to 32-bit integers

  fc -v -i p3.f -l
```

One problem you may encounter when using 'fc' is that the Primos Fortran compiler pays no attention to i/o redirection when it is writing error messages to the terminal. This is a problem common to all Primos commands called from the Subsystem. If you want to record the terminal output of the Fortran compiler, you must use the Primos command output facility. This facility is accessed through the Subsystem 'como' command; for details, see the Software Tools Subsystem Reference Manual or use the 'help' command.

**Linking**

The last step in preparing the program for execution is linking. The linking step fixes the memory locations of the Subsystem common areas; assigns the binary module for each subprogram to an absolute memory location; and links in the required Subsystem support routines, Fortran run-time routines, and Primos system calls. The memory image file produced by this step may then be executed. It should be noted here that programs linked under the Subsystem can run only under the Subsystem; they may not run without it.

The 'ld' command is used to invoke the Primos loader to to do the linking. Its syntax is as follows:

```
ld [-u] <binary file> . . . [-l <library file>] . . .
     [-t -m] [-o <output file>]
```

This is not the entire syntax accepted by 'ld,' but a complete discussion requires detailed knowledge of the Primos loaders. For more information, see the Subsystem reference manual.

The "-u" option causes the loader to print a list of undefined subprograms. Any number of binary files to be included may be listed. The only restriction is that the main program must be the first binary subprogram encountered -- it must be the first program unit in a binary file, and that binary file must be

the first <binary file> to appear on the command line. Any number of libraries (residing in "=lib=") may then be specified with the "-l" option. The "-t -m" options cause a load map to be produced on a file with the name as the output file (or first <binary file>, if an output file is not specified) with ".m" appended. If the file name ends with ".b", the ".b" is replaced by the ".m". The "-o" option specifies the name of the output file. If the "-o" option is omitted, the output file will have the same name as the first <binary file>, with ".o" appended. If the name of the first <binary file> ends in ".b", the ".b" will be replaced by the ".o".

Even though linking is a mysterious process, it need not be traumatic. Most of the time, you will be linking a single binary file with no additional libraries. For instance, if you had a binary file named "prog.b," you could produce an object program by just typing the command

```
ld prog.b
```

The Primos loader would be invoked, and after a great deal of garbage was printed on the terminal, the executable program "prog.o" would be produced.

The only thing that you must do is look for the message "LOAD COMPLETE" lurking somewhere near the end of this garbage. If you find this message, it means that all of the external references in your program (subroutine and function calls) have been satisfied, and linking is complete. If you don't find this message, there are unsatisfied references in your program. You may then call 'ld' with the "-u" option and the loader will print the names of the unsatisfied references on the terminal. You will probably then find that these references are caused by misspelled subprogram names, missing subprograms, or undimensioned arrays (remember, the Fortran compiler treats undimensioned arrays as functions calls, so you may not always get an error message from the compiler).

Again, for completeness, here are some examples of 'ld' at work:

```
# link the binary files "p1.b", "p2.b", and "p3.b"
#    to produce "p1.o" as output

  ld p1.b p2.b p3.b


# link the binary file "nprog.b",
#    include the library "vshlib",
#    and produce the output file "nprog"

  ld nprog.b -l vshlib -o nprog


# link the binary files "np1" and "np2",
#    produce a load map,
#    and output "my_new_prog"

  ld np1 np2 -t -m -o my_new_prog
```

The Primos loader also pays no attention to i/o redirection. If you want to catch its terminal output, you must use the Primos 'como' commands. For details, see the reference manual or use the 'help' command.

**Executing**

Executing a Subsystem program is the easiest step of all. All you have to do to execute it is to type its name. For instance, if your object program was named "prog.o", all you need type is

```
prog.o
```

to make it go. Because the shell also looks in your current directory for executable programs, "prog.o" is now a full-fledged Subsystem command. You may give it arguments on its command line, redirect its standard inputs and outputs, include it in pipelines, or use it as a function. Of course to be able to do all of these things properly, it must observe the Subsystem conventions and use the Subsystem I/O routines.

**Shortcuts**

There are several shortcuts that speed things up and save typing when developing programs.

Shell Programs. Shell programs can be a great help when performing repetitive tasks. Quite often one of these tasks is preprocessing, compiling, and linking a program during its development. A simple shell program can save a great deal of

typing in this situation. For instance, let's say we are writing a Ratfor program that is in the file "np.r". We are in the process of adding new features to "np" and will probably compile and test it several times. We can make a very simple shell program that will keep us from having to type 'rp,' 'fc,' and 'ld' commands every time we want to make a test run. All we have to do make a file containing these three commands with 'cat':

```
] cat >cnp
rp np.r
fc np.f
ld -u np.b -o np
<control-c>
]
```

Now the file "cnp" contains the following text:

```
rp np.r
fc np.f
ld -u np.b -o np
```

All we need do now to preprocess, compile, and link our program is just type the name of the shell program as a command:

```
cnp
```

and the shell will execute all of the commands contained in it.

The 'Rfl' Command. Of course, it is so common to preprocess, compile, and link a program, there is an already-built shell program that works nicely in most cases. 'Rfl' contains the necessary commands to preprocess, compile and link a Ratfor program contained in a file whose name ends with ".r". All you have to do is type

```
rfl np.r
```

and 'rfl' will execute the necessary commands to produce an executable file named "np". (note that the executable file is named "np" and not "np.o"!) 'Rfl' can also do some other handy things that you can find out about in the Subsystem reference manual.

Storing Source Programs Separately. When you write fairly large programs or test modules independently, it is often convenient to store the programs in separate files. If this is the case, creating an executable program is just a little bit more complicated. The easiest solution is to just name all of the programs on the 'rp' command line, like this:

```
rp p1.r p2.r p3.r
```

'Rp' will preprocess all of the files together and produce output on the file "p1.f". The **define** statements in "p1.r" will still be in effect when "p2.r" is preprocessed, etc. so "p1.r", "p2.r", and "p3.r" might just as well be together in one file.

Compiling Programs Separately. A little bit harder, but sometimes much faster, is to preprocess and compile the modules separately and then combine them during linking. There are two things that you have to watch. The first problem with separate compilation is that **define** statements in one file cannot affect subprograms in the other files. For a large program that would benefit from separate compilation, this nastiness can be avoided by placing all of the **defines** together in one file and placing an **include** for that file at the beginning of each of the files containing the program. The **defines** will then be applied uniformly to all parts of the program.

The second thing is that since Ratfor chooses unique Fortran names in the order it is presented with "long" Ratfor names, it cannot guarantee that a long name in one file will be transformed into exactly the same Fortran name as the same long name in a second file (although the probability is quite high). To avoid problems, either subprogram names that are cross-referenced in the separate binary files should be given six-character or shorter names, or a **linkage** declaration containing the names of all subroutines, function, and common blocks should be inserted at the beginning of each module. It is usually easiest to handle the **linkage** declaration just like the **define** statements: put it in a separate file, and add an **include** statement for it at the beginning of each module.

Then, the program units in each file may be preprocessed and compiled separately. The binary files from the separate compilations are linked together by just listing the names of all of the files on the 'ld' command:

```
ld p1.b p2.b p3.b
```

The only restriction is that the main program must appear first. The object file from this example would be named "p1.o", but this could have been overridden by including the "-o <output file>" option.

When compiling parts of a program separately, you should be aware that incorrect use of the **linkage** declaration can cause totally irrational behavior of the program with no other indication of error. Since no checking is done on the **linkage** declaration, you must be certain that every external name appears in the statement. More importantly, when you add a subroutine, function, or common block, you must remember to change the **linkage** declaration. In addition, if you do not add the name to the very end of the declaration, you must immediately recompile all modules! If you compile separately, and are confronted with a situation in which your program is misbehaving for no apparent reason, re-check the **linkage** declaration and recompile all the modules.

## Debugging

Debugging unruly programs under Primos is at best a grueling task, as currently there is almost no run-time debugging support. Except for a couple of machine-language level debuggers, you'll get very little help from Primos (except for some nasty error messages) while debugging programs. This means that such techniques as top-down design, reading other programmers' code, and reasonably careful desk checking will pay off in the long run. But even with all the care in the world, some bugs will creep through (especially on an unfamiliar system). The next few paragraphs will be devoted to techniques for exterminating these stubborn bugs.

For an experienced user, a load map, the Primos DMSTK command, and VPSD (the V-mode symbolic debugger) can very quickly isolate the location, if not the cause, of a bug. With more complicated programs that are dependent on the internal structure of the machine and operating system, such machine level debugging cannot always be avoided. If you find yourself in such as position, you can begin to learn some of these things by examining the following reference manuals:

MAN 1671 System Reference Manual, Prime 100-200-300

MAN 2798 System Reference Manual, Prime 400

FDR 3059 The PMA Programmer's Guide

FDR 3057 User Guide for the Fortran Programmer

Most often, the bug can be found by one or more of the following techniques:

(1) Inserting 'print' calls to display the intermediate results within the program.

(2) Using the Ratfor subroutine trace.

(3) Using the Fortran statement number and assignment trace.

It is usually quickest to use the Ratfor subroutine trace (by including the "-t" option on the 'rp' command line). Although this trace lists only subroutine nesting, it will narrow down where a program is blowing up to a single subprogram. If the program is very modular and contains mostly small subprograms, quite often, the error can be spotted.

If the Ratfor trace fails to pinpoint the problem, the Fortran statement and assignment trace will give a great deal more information (possibly hundreds of pages). The Fortran trace can be produced by specifying the "-t" option on the 'fc' command. The Fortran code produced by 'rp' must be examined to locate the statement numbers, but given the large number of statement labels generated by 'rp,' study of this trace can

isolate the problem practically to within one statement.

The above debugging methods are quick and easy to use when the program contains a catastrophic error that causes an error termination or an infinite loop. While this is sometimes the case, more often a subtle error is the problem. In finding these errors, there is no substitute for carefully inserted debugging code (such as calls to 'print') at critical points in the program.

The rest of this section is devoted to a brief description of many of the terminal errors that may do away with programs (and the Subsystem). Most terminal errors cause the Subsystem command interpreter to be terminated along with the user's delinquent program. You can tell that you've been booted into Primos by the appearance of the "OK," or "ER!" prompt. All error messages that cause an exit to Primos are briefly explained in appendix A-4 of the Prime Fortran Programmer's Guide (FDR3057). Some very common programming errors can cause cryptic error messages with explanations that are close to unintelligible. Hopefully, most of these messages are described below.

Many Primos error messages are dead giveaways of program errors. Messages that begin with four asterisks are from the Fortran runtime packages -- they usually indicate such things as division by zero or extraction of the square root of a negative number. For example,

```
**** SQRT -- ARGUMENT < 0
OK,
```

results from extracting the square root of a number less than zero.

Other, more mysterious, error messages can also be caused by simple program errors.

```
Error: condition "POINTER_FAULT$" raised at <addr>
```

can be caused by referencing a subprogram which has not been included in the object file. An obvious indication of a missing subprogram is the failure to get the

```
LOAD COMPLETE
```

message from 'ld'. (Note that the Fortran compiler treats references to undimensioned arrays as function calls!) A more insidious cause of the "POINTER FAULT" message is a reference to an unspecified argument in a subprogram; i.e. the calling routine specifies three arguments and the called routine expects four. The error occurs when the unspecified argument is referenced in the subprogram, not during the subprogram call.

```
Error: condition "ACCESS_VIOLATION$" raised at <addr>
Error: condition "RESTRICTED_INST$" raised at <addr>
Error: condition "ILLEGAL_SEGNO$" raised at <addr>
```

```
     Error: condition "ARITH$" raised at <addr>
     Program halt at <addr>
```

all can result from a subscript exceeding its bounds. Because the program may have destroyed parts of its code, the memory addresses sometimes given may well be meaningless. Even so, you may locate the routine in which the program blew up by using the Primos DMSTK command and a load map. For instance, given the following scenario (ellipsis indicate irrelevant information),

```
     Error: condition "POINTER_FAULT$" raised at 3.4000.001000.
     Abort (a), Continue (c) or Call Primos (p)? p
     OK, dmstk
     ...
     Stack Segment is 6002.

        6) 001464: Condition Frame for "POINTER_FAULT$"; ...
           Raised at 3.4000.017202; LB= 0.4000.017402, ...

        7) 001374: Fault Frame; fault type= 000064
           Returns to 3.4000.017202; LB= 0.4000.017402, ...
           Fault code= 100000, Fault addr= 3.4000.017204
           Registers at time of fault:
     ...
```

The numbers following "LB=" on the underlined portion of the stack dump show the address of the data area of the procedure executing when the fault occurred. The segment number portion of this address (the four-digit part) tells who the routine belongs to:

| Segment | Use |
|---|---|
| 0000 - 0033 | Operating System |
| 2030 | Software Tools Shell |
| 2031 | Software Tools Screen Editor |
| 2035 | Software Tools Library |
| 2050 | Fortran Library |
| 4000 - 4037 | User Program |
| 4040 | Software Tools Common |
| 4041 | Software Tools Stack |
| 6001 | Fortran Library |
| 6002 | Primos Ring 3 Stack |

If the executing routine is not part of your program, you can trace back the stack (see below) until you find which of your subprograms made the call. If the segment number begins with "4", you need only look down the right-most two columns of the load map (see the 'ld' command) for the two numbers (4000 17402 in this case). If you get an exact match, just look across to the name on the left -- this is the subprogram that was executing. Otherwise, if none of the numbers match then either the program has clobbered itself and jumped into nowhere, you left off an argument to a library subprogram, or one of the library routines has caused an exception trap with no fault vector.

Subsequent entries in the stack dump (following the information in the last scenario) can be used to find what procedure calls were in process when the error occurred. The entries are of the following form:

```
Stack Segment is 4041.

   8) 002222: Owner=  (LB= 0.4000.017402).
      Called from 3.4000.017700; returns to 3.2035.017702.

   9) 002156: Owner=  (LB= 0.4000.013026).
      Called from 3.4000.013442; returns to 3.2030.013450.
...
```

Each entry on the Subsystem stack (segment 4041) represents a procedure call in process. You can use the numbers following the "LB=" and the load map to trace back through the "stack" of procedure calls, just as with the "fault frame" mentioned above.

If you find yourself at a complete and total loss at finding why your program is blowing up, here is a list of some of the errors that have caused us great anguish:

- Subscript out of range. This error can cause any number of strange results.

- Undefined subprogram. This error can be detected by the lack of a "LOAD COMPLETE" message from the 'ld' command.

- Too few arguments passed. This error almost always causes a "POINTER_FAULT$" when the missing argument is referenced.

- Code and initialized local data requires more that one segment (64K words). The load map shows how much space is allocated. No linkage or procedure frame should appear in any segment other than 4000.

- Delimiter character is missing in a packed string. This includes periods in packed strings passed to 'print' and 'input'. This error causes the program to run wild, writing all over the place.

- Type declaration is missing for a function. This error can causes failure of routines such as 'open' which return an integer result. The Primos Fortran compiler does not flag undeclared functions. This error may also cause an erratic real-to-integer conversion error or cause the program to take an exception trap.

- A subprogram is changing the value of a constant. If you pass a single constant as a function or subroutine argument, and the subprogram changes the corresponding parameter, the values of all occurrences of that constant in the calling program will be changed. With this error, it is quite possible for the constant 12 to have the value -37 at some time during execution.

## Performance Monitoring

In most cases, it is very difficult to determine how much processing time is required by different parts of a program. Since it is nearly impossible to determine which parts of a program are "inefficient", especially before the program is written, it often more effective to write a program in the most simple and straightforward manner, and then use performance monitoring tools to find where the program is spending its time. It has many times been our experience to find even though parts of a program are coded inefficiently, only a very small amount of time is wasted.

There are two available methods for obtaining an execution time "profile" of a Ratfor program. The first method provides statistics on the number of calls to and the amount of time spent in each subprogram. The second method provides a count of the number of times each statement in the program is executed.

To invoke the subroutine profile, just preprocess (in one run) all the subprograms to be profiled. Add the "-p" option to the 'rp' command line when the programs are preprocessed. Then compile, link and execute the program normally. When the program terminates (it must execute a **stop** statement, and not call "error"), type the command

```
profile
```

'Profile' accesses the files "timer_dictionary" (output by 'rp') and "_profile" (output by your program) and prints the subroutine profile to standard output.

To invoke the statement count profile, put all the subprograms to be profiled (you must also include the main program) in a single file. Then preprocess the file with 'rp' and the "-c" option. Compile, link, and execute the program. When the program terminates normally, type the command

```
st_profile myprog.r
```

(Of course, assuming your source file name is "myprog.r".) A listing of the program with execution count for each line will be printed.

When running a profile, there are several things to keep in mind. First, the program with the profiling code can be more than twice as large as the original program. Second, the program can run an order of magnitude more slowly. Third, there can be a considerable delay between the execution of the **stop** statement and the actual end of the program. Finally, you should remember that the main program and all subprograms to be profiled must be preprocessed at the same time.

## Conditional Compilation

Conditional compilation is a handy trick for inserting debugging code or setting compile-time options for programs. Conditional compilation can be approximated in Ratfor by defining an identifier, such as "DEBUG" to a sharp sign or null (for off and on respectively). Lines in the Ratfor program beginning with the identifier "DEBUG" (i.e. debugging code) are not compiled if "DEBUG" is defined to be "#", but are compiled normally if "DEBUG" is defined as a null string.

For instance, the following example shows how conditional compilation can be used to "turn off" print statements at compile time:

```
define (DEBUG, #)

   fd = open (fn, READ)
DEBUG  call print (ERROUT, "fd returned:*i*n"s, fd)
...
   len = getlin (str, fd)
DEBUG  call print (ERROUT, "str read: *s"s, str)
```

In this example, all lines beginning with "DEBUG" are ignored, unless the **define** statement is replaced with

```
define (DEBUG, )
```

Then, all lines beginning with "DEBUG" will be compiled normally.

## Portability

If your intent is to produce portable Fortran code, the Ratfor preprocessor, 'rp' can be invoked with the following four options:

**-h** Produce Hollerith-format string constants rather than quoted string constants. This option useful in producing character strings in the proper format needed by your Fortran compiler.

**-v** Output "standard" Fortran. This option causes 'rp' to generate only standard Fortran constructs (as far as we know). This option does not detect non-standard Fortran usage in Ratfor source code; it only prevents 'rp' from generating non-standard constructs in implementing its data and control structures.

**-x** Translate character codes. 'Rp' uses the character correspondences in a translation file to convert characters into integers when it builds Fortran "data" statements containing EOS-terminated or PL/I strings. If the option is not specified, 'rp' converts the characters using the native Prime character set.

**-y** Do not output "call swt". This option keeps 'rp' from generating a "call swt" in place of all "stop" statements, which are required for Fortran programs to run under the Subsystem.

The following option for 'fc' may also help:

**-i** Consider all integers to be "long" (32-bit) rather than short.

## Source Program Format Conventions

After considering many program formatting styles, we have concluded that the convention used by Kernighan and Plauger in Software Tools is the most expedient in terms of clarity and ease of modification. As a consequence, we have tried to be consistent in the use of this convention throughout the Subsystem to provide uniformly readable and modifiable code. We present the convention here in the hope that you can use it to the same advantage.

### Statement Placement

The placement of statements in program units is perhaps the most important part of the formatting convention. Through uniform placement of statements, many documents can be produced directly directly from the source code. For instance, the skeleton for Section 2 of the Subsystem Reference Manual was produced originally from the subprogram headers of the Subsystem library subprograms. Then the detail was filled in using the text editor.

The order of a program unit (including a main program) should be as follows:

1. A comment line of the following format:

   # <program name> --- <one-line description>

2. The **subroutine** or **function** statement (or nothing if it is a main program).

3. The declarations of all arguments passed to the sub-program, if any.

4. A blank line

5. Declarations for all local variables in the program unit.

6. A blank line.

7. Executable program statements.

8. The **end** statement.

9. Three blank lines.

Of course, extra blank lines should be used freely to separate different logical groups of declarations and different logical blocks of executable statements.

As an example, here is the source code for the subroutine "cant" taken directly from the Subsystem library:

```
# cant --- print cant open file message
   subroutine cant (str)
   character str (ARB)

   call putlin (str, ERROUT)
   call error (": can't open.")

   return
   end
```

## Indentation

The indentation convention is very simple. It is based on the idea that a statement should be indented three spaces to the right of the innermost statement controlling it. Braces are placed as unobtrusively as possible, without affecting the ease of adding or deleting statements.

Statements, with the exception of the program heading comment, are placed three spaces to the right of the left margin. All statements are placed in this position, unless they are subordinate to a control statement. In this case, they are placed three spaces to the right of the beginning of the controlling statement.

Braces do not affect the placement of statements. An opening brace is placed on the line with the controlling statement. A closing brace is placed on a separate line three spaces to the right of the beginning of the controlling statement.

Multiple statements per line are forbidden, except when a chain of **if - else if . . . else** statements is used to implement a case structure. In this event, the **else if** is considered a single statement, appearing on the same line, and subsequent lines are indented only three spaces to the right.

If all of this seems terribly confusing, here are some examples that show the indentation convention in action (the bars are just to show you the matching of braces):

```
for (i = 1; str (i) ~= EOS; i += 1) {
|  if (str (i) == 'a'c) {
|  |  j = ctoi (str (2), i)
|  |  select (j)
|  |  |  when (1)
|  |  |  |  call alt1
|  |  |  when (2)
|  |  |  |  call alt2
|  |  |  when (3) {
|  |  |  |  call alt1
|  |  |  |  call alt2
|  |  |  |  }
|  |  else
|  |     call error ("number must be >= 1 and <= 3"s)
|  ---}
|  else if (str (i) == 's'c)
|     repeat {
|     |  j = ctoi (str (2), i)
|     |  status = getnext (j)
|     ---} until (status == EOF)
|  else {
|  |  call clean_up
|  |  stop
|  ---}
---}
```

## Subsystem Definitions

The use of the **define** statement plays a large part in producing readable, maintainable programs. Hiding implementation details with **define** statements not only produces more readable code, but allows changes in the implementation details to be made without necessitating changes in applications programs. The development of a large part of the Subsystem would have been greatly hindered if it had not been possible to redefine the constant "STDIN" from "1" to "-11", with no more than recompilation.

The Subsystem definitions file, "=incl=/swt_def.r.i" exists primarily to hide the dirty details of the Subsystem support routines from Ratfor programmers. We sincerely believe that the character string "EOF" is inherently more meaningful than the string "-1". (Would you believe that after three years of using the Subsystem, the author of this section had to look up the value assigned to "EOF" in order to write the preceding sentence?)

Of course, the use of the Subsystem definitions also allow the developers to change these values when necessary. Of course, these changes force recompilation of all existing programs, but we feel that this is a small price to pay for the availability of more advanced features. All users of the Subsystem support routines are therefore warned that the values of the Subsystem definitions may change between versions of the Subsystem. (At

Georgia Tech, this may be daily.) Programs that depend on the specific values of the symbolic constants may well cease to function when a new version of the Subsystem is installed.

Appendix D contains specific information about (but not specific specific values for) the standard Subsystem definition file. As a general rule, all symbolic constants mentioned in Section 2 of the Subsystem Reference Manual can be found in "=incl=/swt_def.r.i".

## Using the Subsystem Support Routines

Many of the capabilities available to a Subsystem programmer are provided through the Subsystem support routines. The Subsystem support routines consist of well over one hundred Ratfor and PMA subprograms that either perform common tasks, insulate the user from Primos and Fortran, or conceal the internal mechanisms of the Subsystem. By default, the library containing all of these routines ("=lib=/vswtlb") is included in the linking of all Subsystem programs. Therefore, no special actions need be taken to call these routines.

If you notice that there are some "holes" in the functionality of the Subsystem library, you are probably quite correct. The Subsystem library has grown to its present size through the effort of many of its users. The instance often arises that a routine is required to fill a specific function. In keeping with the Software Tools methodology, instead of writing a very specific routine, we ask that the author write a slightly more general routine that can be used in a variety of instances. The routine can then be documented and placed in the Subsystem library for the benefit of all users. Many of the support routines, including the dynamic storage management routines, have come from just such instances. The "holes" in the Subsystem library are just waiting for someone to fill them; if you need a routine that isn't there, please write it for us.

### Termination

The subprogram 'swt' terminates the program and causes a return to the Subsystem command interpreter. Any Subsystem files left open by the program are closed. Ratfor automatically inserts a "call swt" any time it encounters a Fortran **stop** statement. All Ratfor programs should **stop** rather than "call exit". Fortran and PMA programs should invoke 'swt' to terminate.

### Character Strings

Most of the support routines use characters that are unpacked, one per word (i.e. integer variable), right-justified with

zero fill, rather than the Fortran default, two characters per word, left-justified, with blank fill (for an odd last character). In addition to the simplicity of manipulating unpacked strings, the unpacked format represents characters as small, positive integers. Thus, character values can be used in comparisons and as indexes without conversion.

Most of the support routines that manipulate character strings expect them to be stored in an integer array, one character per word, right-justified and zero-filled, and terminated with a word containing the symbolic constant 'EOS'. Strings of this format are usually called EOS-terminated strings.

Support for the use of unpacked characters is provided in several ways: (1) the Subsystem I/O routines perform conversion to and from unpacked format, (2) single-character constants 'a'c, 'b'c, ','c, etc. are provided for use in place of single-character Hollerith literals, and (3) the Ratfor **string** statement is provided to initialize EOS-terminated strings.

In a few cases, it is more convenient to use a Hollerith literal instead of an EOS-terminated string. Since it is impossible to tell the length of a Hollerith literal at run time, Hollerith literals used with the Subsystem are required to contain a delimiter character (usually a period) as the last character. Hollerith literals or integer arrays that contain Hollerith-format characters and end with a delimiter character are referred to as packed strings.

Following are brief descriptions for the most generally useful character manipulation routines. For specific information, see the Software Tools Subsystem Reference Manual.

Equal. 'Equal' is an integer function that takes two EOS-terminated strings as arguments. If the two strings are identical, 'equal' returns YES; otherwise it returns NO. For example,

```
string dash_x "-x"
integer equal
...
if (equal (argument, dash_x) == YES)
   call cross_ref
```

Index. 'Index' is used to find the position of a character in an EOS-terminated string. If the character is in the string, its position is returned, otherwise zero is returned. 'Index' is very similar to the built-in function of the same name in PL/I. Example:

```
string options "acx"
integer ndx
integer index
...
ndx = index (options, opt_character)
select (ndx)
   when (1)
      call list_all
   when (2)
      call list_common
   when (3)
      call cross_reference
else
   call remark ("illegal option"s)
```

This example selects one of a number of subroutines to be executed depending on a single-character option specifier. Of course, this particular example could be done with just **select** alone. 'Index' is also useful in character transliteration and conversion from character to binary integer.

Length. 'Length' is an integer function that returns the length of an EOS-terminated string. The length of a string is zero if and only if its first character is an EOS; it is the number of characters before the EOS in all other cases. 'Length' is often useful in deciding where to start appending additional text, as in the following example:

```
integer len
integer length
...
len = length (str)
call scopy (new_str, 1, str, len + 1)
```

Mapdn and Mapup. These functions accept a single character as an argument and if the character is alphabetic, force it to lower or upper case, respectively. 'Mapdn' and 'mapup' quite often find use in mapping option letters to a single case before comparison. Since non-alphabetic characters are not modified, these routines may be used safely even if non-alphabetic characters appear. In addition, these routines provide a very good place to isolate character set dependencies. For example,

```
character c
character mapdn
...
if (mapdn (c) == 'a'c) {
   # handle 'a' option
...
else if (mapdn (c) == '1'c) {
   # handle '1' option
```

Mapstr. 'Mapstr' provides case mapping for alphabetic characters in EOS-terminated strings. As arguments 'mapstr'

takes a string and the symbolic constant 'LOWER' or 'UPPER'. Alphabetic characters in the string are then forced to lower or upper case, depending on the constant specified.

Scopy. The subroutine 'scopy' is used for copying EOS-terminated strings. It requires four arguments: the source string, the position from which to start copying, the destination string, and the position at which filling begins in the destination string. Since Ratfor provides no string assignment, 'scopy' is normally used to provide the capability. The simple movement of a string from one place to another is coded as

```
character str1 (MAXLINE), str2 (MAXLINE)
...
call scopy (str1, 1, str2, 1)
```

'Scopy' is also capable of appending one string to another, as in the following example:

```
character str1 (MAXLINE), str2 (MAXLINE)
...
call scopy (str1, 1, str2, length (str2) + 1)
```

Note that 'scopy' makes no attempt to avoid writing past the end of 'str2'!

Type. 'Type' is another of the routines that is intended to isolate character dependencies. Type is a function that takes a single character as an argument. If that character is a letter, 'type' returns the constant 'LETTER'; if the character is a digit, 'type' returns the constant 'DIGIT'; otherwise, 'type' returns the character. 'Type' often finds use in a lexical analyzer:

```
character c
character type

if (type (c) == LETTER) {
   # collect identifier
...
else if (type (c) == DIGIT) {
   # collect integer
...
else {
   # handle special character
```

## File Access

File access is one of the more important aspects of the Subsystem. It is through the Subsystem i/o routines that device independence and i/o redirection are accomplished; moreover, the Subsystem routines provide a much less complicated interface than comparable Primos routines.

The basic method of access to a Subsystem file is through the contents of an integer variable called a **file descriptor.** File descriptors can be set by one of several routines or they can be set to one of the six standard descriptors representing the six standard ports provided to all Subsystem programs.

Quite often, the standard ports provide all of the file access required by a program. Values for the standard port descriptors can be accessed from **defines** contained in "=incl=/swt_def.r.i" ('Rp' automatically includes this file in each run). The following table gives the symbolic names for the three standard input and three standard output ports available:

| Input Ports | Output Ports |
|---|---|
| STDIN1 (or STDIN) | STDOUT1 (or STDOUT) |
| STDIN2 | STDOUT2 |
| STDIN3 (or ERRIN) | STDOUT3 (or ERROUT) |

These constants may be used wherever a file descriptor is required by a Subsystem i/o routine.

Other files may be accessed or created through the routines 'open', 'create', and 'mktemp' that are described later. At the moment, it is sufficient to say that these routines are functions that return a file descriptor that may be used in other Subsystem i/o calls.

Once a file descriptor has been obtained, the file it references may be read with the routines 'getlin', 'getch', or 'input'; written with the routines 'putlin', 'putch', or 'print'; positioned with the routines 'wind' or 'rewind'; or closed with the routines 'close' or 'rmtemp'.

Open and Close. 'Open' takes an EOS-terminated path name and a mode (one of the constants READ, WRITE, or READWRITE) as arguments and returns the value of a file descriptor or the symbolic constant ERR as a function value. 'Open' is normally used to make a file available for processing in the specified mode. If the mode is READ, 'open' will open the file for reading; if the file doesn't exist or cannot be read (i.e. no read permission), 'open' will return ERR. If the mode is WRITE or READWRITE, 'open' will open an existing file or create a new file for writing or reading and writing, if possible; otherwise it will return ERR. If 'open' opens an existing file, it will never destroy the contents, even if mode is WRITE. To be certain that a "new" file is empty, use 'create' instead of 'open'.

'Close' takes a file descriptor as its argument; it closes and releases the file attached to the descriptor. If 'close' is called with a standard port, it takes no action.

Opening and closing a file is really very easy. This example opens a file named "=extra=/news/index" and returns the file descriptor in 'fd'. If the file can't be opened, the program will terminate with a call to 'cant'.

```
file_des fd
integer open
string fn "=extra=/news/index"

fd = open (fn, READ) # open "=extra=/news/index"
if (fd == ERR)
   call cant (fn)

<process the contents of =extra=/news/index>

call close (fd)      # release the file
stop
```

If the file can't be opened, 'cant' will print the message

```
=extra=/news/index: can't open
```

and terminate the program.

Create. 'Create' takes the same arguments as 'open', but also truncates the file (makes it empty) to be sure that there are no remnants of its previous contents.

Mktemp and Rmtemp. Quite often, programs need temporary files for their internal use only. 'Mktemp' and 'rmtemp' allow the creation of unique temporaries in the directory "=temp=". 'Mktemp' requires only a mode (READ, WRITE, or READWRITE) as an argument and returns a file descriptor as its function value. 'Rmtemp' takes a file descriptor as its argument and destroys and closes the temporary file. (One should use caution, for if a descriptor for a permanent file is passed to 'rmtemp', that file will also be destroyed.)

Typical use of 'mktemp' and 'rmtemp' usually involves the writing and reading of an intermediate file:

```
file_des fd
integer mktemp

fd = mktemp (READWRITE) # create a temporary file

<code to write the intermediate file>

call rewind (fd)        # reposition the temporary

<code to read the intermediate file>

call rmtemp (fd)        # close and destroy the temporary
```

Wind and Rewind. The subroutines 'wind' and 'rewind' allow the positioning of an open file to its end and beginning, respectively. Both take a file descriptor as an argument. Usually, 'rewind' is used when a program creates a file and then wishes to read it back; 'wind' is often used when a program wants to add to the end of an existing file.

A program wishing to extend a file would make a call to 'wind' just after successfully opening the file to be extended:

```
file_des fd
integer open
string fn "myfile"

fd = open (fn, READWRITE)
if (fd == ERR)
   call cant (fn)
call wind (fd)    # file is now positioned at the
                  #    end, ready for appending.
```

Trunc. 'Trunc' truncates an open file. Truncating a file means releasing all of its disk space, hence making it empty, but retaining its name and attributes. 'Trunc' takes a file descriptor as its argument.

Remove. 'Remove' removes a file by name, deleting it from the disk directory. It takes an EOS-terminated string as its argument, and returns the constant OK or ERR, depending on whether or not it could remove the file. ('Remove' will also delete a Primos segment directory without complaining.)

Cant. 'Cant' is a handy routine for handling exceptions when opening files. For its argument, 'cant' takes an EOS-terminated string containing a file name. It prints the message

```
<file name>: can't open
```

and then terminates the program.

Getlin. All Subsystem character input is done through 'getlin'. 'Getlin' takes a character array (at least MAXLINE long) and a file descriptor and returns a line of input in the array as an EOS-terminated string. Although the last character in the string is normally a NEWLINE character, if the line is longer than MAXLINE, no NEWLINE will be present and the rest of the line will be obtained on the next call to 'getlin'. For its function value, 'getlin' returns the length of the line delivered, (including the NEWLINE, if any) or the constant EOF if end-of-file was encountered.

Most line-oriented i/o is done with 'getlin'. For instance, using 'getlin' with its analog 'putlin', a program to select only those lines beginning with the letter "a" can be written very quickly:

```
character buf (MAXLINE)
integer getlin

while (getlin (buf, STDIN) ~= EOF)
   if (buf (1) == 'a'c)
      call putlin (buf, STDOUT)
```

'Getlin' is guaranteed to never return a line longer than the symbolic constant MAXLINE (including the terminating EOS).

If needed, there are a number of routines that you can call to convert the character string returned by 'getlin' into other formats, such as integer and real. Most of these routines are described later in the section on "Type Conversion".

Getch. 'Getch' returns one character at a time from a file; it requires a character variable and a file descriptor as arguments; it returns the character obtained, or the constant EOF, in the supplied argument and as the function value. Calls to 'getch' and 'getlin' may be interleaved; 'getlin' will pick up the rest of a line not read by 'getch'.

'Getch' is very useful in lexical analyzers or just when counting characters. For instance, the following routine counts both characters and lines at the same time:

```
character c
integer c_count, l_count
integer getch

c_count = 0
l_count = 0
while (getch (c, STDIN) ~= EOF) {
   c_count = c_count + 1
   if (c == NEWLINE)
      l_count = l_count + 1
   }
```

This example assumes that since each line ends with a NEWLINE character, lines can be counted by counting the NEWLINEs.

Input. 'Input' is a rather general routine created to provide easy access to both interactive and file input. For interactive input, 'input' will prompt at the terminal, accept input, and call the proper conversion routines to produce the desired data formats. In case of unexpected input (like letters in an integer), it will ask for a line to be retyped. For file input, 'input' recognizes that its input is not coming from a terminal (even if from a standard port) by turning off all prompting. It will then accept fixed or variable-length fields from the file under control of the format string.

'Input' requires a variable number of arguments: a file descriptor, a format string, and as many destination fields as required by the format string. It returns the constant EOF as its function value if it encountered end-of-file; otherwise it returns OK.

The file descriptor passed to 'input' describes the file to be read. All prompting output (if any) always appears on the terminal. The format string passed to 'input' indicates what prompting information is to be output and what data format to expect as input. Prompts to be output are specified as literal

characters; i.e. to output "Input X:", the characters "Input X:" would appear in the format string. Prompting characters may only appear at the beginning of the string and immediately after "skip-newline" ("*n") format codes. Data items to be input are described by an asterisk followed by optionally one or two numbers and a letter. For instance the code to input a decimal integer would be "*i" and the code to input a double precision floating point number would be "*d".

When a call to 'input' is executed, the format string is interpreted from left to right. When leading literal characters are encountered, they are output as a prompt. When the first format code is encountered, a line is read from the file, the corresponding item is obtained from the input line, and the item is placed in the next item in the argument list. More items are removed from the input line until the end of the format string is reached or a newline appears in the input. If the end of the format string is encountered, the rest of the input line is discarded, and 'input' returns OK. Otherwise, if a newline is encountered in the input, fields designated by the format are filled with empty strings, blanks, or zeroes, until the format string is exhausted, or a code ("*n") to skip the NEWLINE and read a new line is encountered.

The format string must contain exactly as many input indicators as there are receiving data items in the call. In any case, the maximum number of input items per call is 10.

Before we go any further, here is an example of an 'input' call to obtain three integers:

```
call input (STDIN, "Type i:  *i*nType j:  *i*nType k:  *i"s,
   i, j, k)
```

If this statement were executed the following might appear at the terminal (user input is boldfaced):

```
Type i:  22 <newline>
Type j:  476 <newline>
Type k:  1 <newline>
```

We could also type all three integers on the same line, and 'input' would omit the prompting for the second and third numbers:

```
Type i:  22 476 1 <newline>
```

There are a number of input indicators available for use in the format string. Since there are a large number of them with many available options, only a few are mentioned in the following table. For further information, see the Subsystem reference manual.

| Item Data Type | Input Representation |
|---|---|

| | | |
|---|---|---|
| *n | skip newline | If there is a NEWLINE at the current position, skip over it and read another line. Otherwise do nothing. ('Input' will never read more than one line per call, unless this format code is present. |
| *i | 16 bit integer | Input an integer with optional plus or minus sign, followed by a string of digits, delimited by a blank or newline. Leading blanks are ignored. The input radix can be changed by preceding the number with "<radix>r" (e.g. octal should be expressed by "8r"). |
| *l | 32 bit integer | Same as "*i". |
| *r | 32 bit real | Input a real number with optional plus or minus sign, followed by a possible empty string of digits, optionally followed by a decimal point and a possibly empty string of digits. Scaling by a power of 10 may be indicated by an "e" followed by an optional plus or minus sign, followed by a string of digits. The number is delimited by a blank; leading blanks are ignored. |
| *d | 64 bit real | Same as "*r". |
| *s | string | Input a string of characters delimited by a blank or newline. No more than MAXLINE characters will be delivered, regardless of input size. Use "*1s" to read in a single character. (Admittedly, this is an inconsistency; there really should be a "*c" format.) |

Fixed size input fields can be requested by placing the desired field size immediately following the asterisk in the format code. For instance, to read three integers requiring five spaces each, you can use the following format string:

```
"*5i*5i*5i"
```

You can also change the delimiting character of a field from its default value of a blank. Just place two commas followed by the new delimiter immediately after the asterisk. For instance, two strings delimited by slashes can be input with the following format string:

```
*,,/s*,,/s
```

Regardless of the delimiter setting, a newline is always treated as a delimiter. One caution: if the delimiter is not a blank, leading blanks in strings are not ignored.

Readf. You can use 'readf' to read binary (memory-image) files that were created with 'writef'. 'Readf' is the fastest way to read files, since no data conversion is performed. However, use of 'readf' and 'writef' tend to make a program dependent on machine word size, and hence, non-portable.

'Readf' takes three arguments: a receiving data array, the maximum number of words to be read, and a Subsystem file descriptor. When called, 'readf' attempts to read the number of words requested; if there are not that many in the file, it returns all that are left. If there are no words left in the file at all, 'readf' returns EOF as its function value; otherwise, it returns the number of words actually read as its function value.

Putlin. 'Putlin' is the primary output routine of the Subsystem. It takes an EOS-terminated string and a file descriptor as arguments, and writes the characters in the string on the file specified by the descriptor. There is no restriction on the length of the input string; 'putlin' will write characters until it sees an EOS. 'Putlin' **does not** supply a newline character at the end of the line; if one is to be written, it must appear in the string. For a simple example, see the description of 'getlin'.

Putch. A single character can be output to a file with 'putch'; it takes a character and a file descriptor as arguments and writes the character on the file specified by the descriptor. Calls to 'putch' and 'putlin' can be interleaved as desired.

Print. 'Print' is a general output routine that accepts a format string and up to ten output data items. Interpreting the format string, 'print' calls the appropriate type conversion routines to produce character data, and outputs the characters as directed by the format string. 'Print' requires several arguments: a file descriptor; an EOS-terminated format string; and zero to ten output data arguments, depending on how many are required by the format string.

The format string contains two kinds of items: literal items which are output when they are encountered, and output items, which cause the next data argument to be converted to character format and output. Literal items are just characters in the string; i.e. to output "X =", the format string would contain "X =". Output items consist of an asterisk, followed by two optional numbers, followed by a letter. For instance an output item for an integer is "*i" and an output item for single precision floating point is "*r". The next example shows the output of three integers:

```
call print (STDOUT, "i = *i, j = *i, k = *i*n"s,
   i, j, k)
```

If this call were executed, the following might be the result:

```
i = 342, j = 1, k = -3382
```

Some of the more useful output items are described in the following table:

| Item | Data Representation |
|---|---|
| *i | short (16 bit) integer |
| *l | long (32 bit) integer |
| *r | single precision (32 bit) real |
| *d | double precision (64 bit) real |
| *p | packed, period-terminated string |
| *s | EOS-terminated string |
| *c | single character |
| *n | newline |

It is possible to exert much more control over the format of output using 'print'; for more information, see the Subsystem reference manual.

Writef. 'Writef' is the companion routine to 'readf'; it writes words to a binary (memory-image) file. It is the fastest of the output routines, since it performs no data conversion. It is called with three arguments: a data array containing the words to be written, the number of words to write, and a Subsystem file descriptor. Here is an example fast file-to-file copy using 'readf' and 'writef' together.

```
integer l, buf (1024)
integer readf
file_des in_fd, out_fd

repeat {
   l = readf (buf, 1024, in_fd)
   if (l == EOF)
      break
   call writef (buf, l, out_fd)
   }
```

Fcopy. 'Fcopy' is a very simple routine that copies files. You open and position the input and output files and call 'fcopy' with the input and output file descriptors. It then copies lines from the input file to the output file. 'Fcopy' uses a great deal of "secret knowledge" of the workings of the Subsystem input-output routines, and as a consequence, it copies disk-file to disk-file very quickly (even when the descriptors are of standard ports).

Markf and Seekf. 'Markf' and 'seekf' are companion routines that implement random access on disk files. 'Markf' takes a file descriptor as argument and returns a "file_mark" (currently a 32-bit integer). 'Seekf' takes the file mark along with a file descriptor and sets the file pointer so that the file is positioned at the same place as when the "mark" was taken.

To be used portably, 'markf' and 'seekf' may only be used between calls to 'readf' and 'writef', or immediately after input

or output of a newline character (i.e. at the ends of lines). In addition, a call to 'putlin' or 'putch' on a file effectively (although not actually) destroys information following the current position of the file. For example, if you want to write a line in a file, go off and do other operations on the file, and then be able to re-read the line later, you can use 'markf' and 'seekf':

```
file_mark fm
file_mark markf
file_des fd
character line (MAXLINE)

fm = markf (fd)
call putlin (line, fd)

### perform other operations on 'fd'

call seekf (fm, fd)
call getlin (line, fd)  # get 'line' back
```

Non-portably, you can assume that a "file mark" is a zero-relative word number within the file -- to get word number 12 in the file, just execute

```
call seekf (intl (12), fd)
call readf (word, 1, fd)
```

(Remember: file marks are 32 bits, not 16! We use 'intl' here to make "12" into a 32 bit integer.) Keep in mind that this "secret knowledge" is useful only with "readf" and "writef", not with any other input or output routine. Blank compression is used in line oriented files, so the position of a line is dependent not only on length of previous lines, but also on their content. This usually makes the position of a line in a file quite unpredictable.

Getto. 'Getto' exists primarily to interface with the Primos file system calls. 'Getto' takes a path name (in an EOS-terminated string) as its first argument. It follows the path and sets the current directory to that specified for the file in the path name. It then packs the file name into its second argument, a 16 word array (with blank padding), ready for a call to the Primos file system. It fills its 3-word third argument with the password of the last node of the path (if there was one). Its fourth argument, an integer, is set to YES if 'getto' changed the attach point, and NO otherwise.

'Getto' often finds use when functions other than those supported by Subsystem routines need to be performed, such as setting the passwords on a directory:

```
integer pfn (16), opw (3), npw (3), pw (3), att
integer getto
string fn "=vars=/system"

if (getto (fn, pfn, pw, att) == ERR)
   call print (ERROUT, "can't get to *s*n"s, fn)
call spas$$ (pfn, 32, opw, npw) # set passwords
if (att == YES)
   call follow (EOS, 0)    # attach back to home
```

## Type Conversion

There are a very large number of type conversion routines available to convert most data types into character strings and back. Because keeping up with all the conversion routine names and calling sequences can be quite a chore, two routines 'decode' and 'encode' exist to handle conversion details in a consistent format. These two routines are described at the end of this section.

Most of the "character-to-something" routines require at least two arguments. The first argument is usually the character string, and the second is an integer variable indicating the first of the characters to be converted. The result of conversion is then returned as the function value, and the position variable is updated to indicate the first position past the characters used in the conversion.

For example, the simplest "character-to-integer" routine, 'ctoi' requires the two arguments mentioned above. Since it skips leading blanks, but stops at the first non-digit character, it can be called several times in succession to grab several blank-separated integers on a line:

```
character str (MAXLINE)
integer i, k (4), pos
integer ctoi
...
pos = 1
do i = 1, 4
   k (i) = ctoi (str, pos)
if (str (pos) ~= EOS)
   call remark ("illegal character in input"s)
```

This routine will assume unspecified values to be zero, but complain if non-numeric, non-blank characters are specified.

Here is a list of all of the currently supported "character-to-something" routines.

ctoc — Character-to-character; copies character strings and pays attention to the maximum length parameter.

| | |
|---|---|
| ctod | Character-to-double precision real; handles general floating point input. |
| ctoi | Character-to-integer (16 bit); does not handle plus and minus signs; decimal only. |
| ctop | Character-to-packed-string; converts to packed format with no delimiter character. |
| ctor | Character-to-single precision real; handles general floating point input. |
| ctov | Character-to-PL/I-character-varying; converts to PL/I character varying format. |
| gctoi | Generalized-character-to-integer (16 bit); handles plus and minus signs; in addition to program-specified radix, accepts an optional user-specified radix from 2-16. |
| gctol | Generalized-character-to-long-integer (32 bit); handles plus and minus signs; in addition to program-specified radix, accepts an optional user-specified radix from 2-16. |

In addition to the "character-to-something" routines, there are the "something-to-character" routines. Most of these routines require three arguments: the value to be converted, the destination string, and the maximum size allowable. They return the length of the string produced as the function value. An EOS is always placed in the position following the last character in the destination string, but the EOS is not included when the size of the returned string is calculated.

Since the functions will accept a sub-array reference for the output string, you may place several objects in the same string. For example, using the "integer-to-character" conversion routine 'itoc', you can place the four integers in the array 'k' into 'str' in character format:

```
character str (MAXLINE)
integer i, k(4), pos
integer itoc
...
pos = 1
do i = 1, 4; {
   pos = pos + itoc (k (i), str (pos), MAXLINE - pos)
   if (pos >= MAXLINE - 1) # there's no room for any more
      break
   str (pos) = BLANK
   pos = pos + 1
   }
str (pos) = EOS   # cover up the last blank
```

This code will place the four integers in 'str', separated by a

single blank. Although all conversion routines leave an EOS in the string, we have to replace it here because we clobber it with the blank.

It's worth noting that the maximum size parameter always includes the EOS -- the conversion routine will never touch any more characters than are specified by this parameter.

Here is a list of all available "something-to-character" conversion routines:

ctoc — Character-to-character; copies character strings and pays attention to the maximum length parameter.

dtoc — Double-precision-real-to-character; handles general floating point conversions in Basic or Fortran formats.

gitoc — Generalized-integer-to-character (16 bit); handles integer conversions; program-specified radix.

gltoc — Generalized-long-integer-to-character (32 bit); handles long integer conversion; program specified radix.

itoc — Integer-to-character (16 bit); handles integer conversion; decimal only.

ltoc — Long-integer-to-character (32 bit); handles long integer conversion; decimal only.

ptoc — Packed-string-to-character; accepts arbitrary delimiter character; will unpack fixed length strings if delimiter is set to EOS and maximum is set to (length + 1).

rtoc — Single-precision-real-to-character; handles general real conversion in Basic or Fortran formats.

vtoc — PL/I-character-varying-to-character; converts PL/I character varying format to character.

Decode. 'Decode' handles conversion from character strings to all other formats. It is written to be used in concert with 'getlin' and other such routines, and as such, has a rather odd calling sequence. It requires a minimum of five arguments: the usual string, and string index; a format string; a format string index and an argument string index. Following are receiving arguments, depending on the data types specified in the format string. In almost all cases, you should just supply variables with a values of 1 for the format index and the argument index. The string index behaves just as it does in all other character-to-something routine -- on successful conversion, it points to

the EOS in the string. The specifics of the format string and receiving fields are identical to 'input'. The only differences are that 'decode' returns with OK in the situations in which 'input' would read another line of input, and EOF otherwise, and that all characters in the format string that are not format codes are ignored.

Encode. 'Encode' is a companion routine to 'decode': it can access all of the something-to-character conversion routines in a consistent way. For arguments it takes a character string, maximum length of the string, a format string, and a varying number of source arguments, depending on the format string. 'Encode' behaves exactly like 'print', except that it puts the converted characters into the string, rather than putting them onto a file.

## Argument Access

Programs often find it necessary to access arguments specified on the command line. These arguments can be obtained as EOS-terminated strings, ready for processing or passing to a routine such as 'open'.

Getarg. 'Getarg' is the only routine that retrieves arguments from the shell's argument buffer. It is called with three arguments: an integer describing the position of the argument desired, a character array to receive the argument, and an integer describing the maximum size of the receiving array. 'Getarg' tries to retrieve the argument in the specified position; if it can, it returns the length of the string placed in the array; if it can't, it returns the constant EOF. 'Getarg' will never write farther in the character array than the size specified in the third argument.

Arguments are numbered 0 through the maximum specified on the command line. Argument 0 is the name of the command, argument 1 is the first argument specified, and so on. The number of arguments present on the command line can be determined by the point at which 'getarg' returns EOF.

As a short example, here is a program fragment that attempts to delete all files specified as arguments on its command line:

```
character file (MAXLINE)
integer i
integer remove, getarg

i = 1
while (getarg (i, file, MAXLINE ~= EOF)) {
   if (remove (file) == ERR)
      call print (ERROUT, "*s: cannot remove*n"s,
         file)
   i = i + 1
   }
```

Parscl. In many programs, argument syntax is quite complex. 'Parscl' exists for the benefit of both programmers and users: it makes coding argument parsing simple and it helps keep argument conventions uniform. Of course, to do this, it must automatically enforce certain argument conventions. 'Parscl' and its accompanying macros expect to recognize arguments of a single letter without regard to case. Rather than a lengthy explanation, let's look at an example: For its arguments, a program requires a page length (which should default to 66 if not present), a title (which may also not be present), a flag to tell whether to format for for a printer or a terminal, and a list of file names to process. In this case, a reasonable option syntax is

```
prog [-l <page length>] [-t [<title>]] [-p] {<file name>}
```

We have used single letter flags to avoid the need for always specifying arguments. Now, in terms of 'parscl', what we have is an "required integer", an "optional string", and a "flag". This means that "-l" cannot be specified without a <page length>, but "-t" can be specified without a <title> (in this case, of course, we would use an empty title). Be sure to note that a "required" argument means that if the letter is specified, it must be followed by a value. It does **not** mean that the letter argument must always be present. In other circumstances, we can also have "optional integer" and "required string" arguments.

To use 'parscl' in our program, we must first include the argument macros and declare the argument data area:

```
include ARGUMENT_DEFS
ARG_DECL
```

Then, near the beginning of the main program, we use a macro call to call 'parscl' that contains the syntax of the command line and a "usage" message to be displayed if the command line is incorrect. For our example, we can use

```
PARSE_COMMAND_LINE ("l<req int> t<opt str> p<flag>"s,
   "prog [-l <page len>] [-t [<title]] [-p] {<file}>"s)
```

For "optional integer" and "required string" arguments, the argument types are "<opt int>" and "<req str>", respectively.

If the command line is parsed successfully, 'parscl' returns and the program continues; otherwise, 'parscl' prints the "usage" message with a call to 'error'. Once 'parscl' has returned, we can set the default values, test for the presence or absence of arguments, and obtain values of arguments. First we usually set default values:

```
ARG_DEFAULT_INT (l, 66)
if (ARG_PRESENT (t))
   ARG_DEFAULT_STR (t, ""s)
else
   ARG_DEFAULT_STR (t, "Listing from prog"s)
```

Remember, default values are set **after** the call to 'parscl'!

In the preceding example, we set the value of the argument for "l" to 66. This is simple enough. But for the "t" argument, we really have three different cases: the argument was specified with a string, the argument was specified without a string (meaning that we must use an empty title), or the argument was not specified at all (meaning that we use some other default). In the first case, neither call to ARG_DEFAULT_STR will do anything, since the string was specified by the user; in the second case, ARG_PRESENT (t) will be ".true." setting the default to the empty string (since the "t" argument was specified, even though it was without a string); and in the third case ARG_PRESENT (t) will be ".false.", setting the default to "Listing from prog".

Now that we have finished setting defaults, we can obtain the values of arguments with more macros: the call ARG_VALUE (l) will return the page length value: either the value specified by the user or the value 66 that we set as the default. ARG_TEXT (t) references an EOS-terminated string containing the title: either the value specified the user, an empty string, or "Listing from prog". Use of the values in our example might look like this:

```
page_len = ARG_VALUE (l)
call ctoc (ARG_TEXT (t), title, MAXTITLE)
if (ARG_PRESENT (p))
   ### do printer formatting
else
   ### do terminal formatting
```

And now, here's how all of the argument parsing will look:

```
include ARGUMENT_DEFS
ARG_DECL

PARSE_COMMAND_LINE ("l<req int> t<opt str> p<flag>"s,
   "prog [-l <page len>] [-t [<title]] [-p] {<file}>"s)

ARG_DEFAULT_INT (l, 66)
if (ARG_PRESENT (t))
   ARG_DEFAULT_STR (t, ""s)
else
   ARG_DEFAULT_STR (t, "Listing from prog"s)

page_len = ARG_VALUE (l)
call ctoc (ARG_TEXT (t), title, MAXTITLE)
if (ARG_PRESENT (p))
   ### do printer formatting
else
   ### do terminal formatting
```

Now, what about the file name arguments we were supposed to parse. Where did they go? 'Parscl' deletes arguments that it processes; it also ignores any arguments not starting with a

hyphen (that do not appear after an letter-argument looking for a string). So the file name arguments are still there, ready to be fetched by 'getarg', with none of the "-t <title>" stuff left to confuse the logic of the rest of the program.

Now, how about some example commands to call this program:

```
prog -p
      (page_len = 66, title = "Listing from prog",
         formatted for printer)

prog -l34 -t new title
      (page_len = 34, title = "new",
         file name = "title",
         formatted for terminal)

prog file1 file2 -p -t -l70
      (page_len = 70, title = "",
         file names = file1 file2,
         formatted for printer)

prog filea -t"my new title" -l 60
      (page_len = 60, title = "my new title",
         file name = filea, formatted for printer)

prog -x filea
      (the "usage" message is printed)

prog fileb -l
      (the "usage" message is printed)
```

As you can see, 'parscl' allows you to specify arguments in many different ways. For more information on 'parscl', see its entry in the Reference Manual.

## Dynamic Storage Management

Dynamic storage subroutines reserve and free variable size blocks from an area of memory. In this implementation, the area of memory is a one-dimensional array. Each block consists of consecutive words of that array.

The dynamic storage routines assume that you have included the following declaration in your main program and in any sub-programs that reference dynamic storage:

```
DS_DECL (mem, MEMSIZE)
```

where 'mem' is an array name that can be used to reference the dynamic storage area. You must also define MEMSIZE to an integer value between 6 and 32767 inclusive. This number is the maximum amount of space available for use by the dynamic storage routines. In estimating for the amount of dynamic storage required, you must allow for two extra 'overhead' words for each block allocated. Three other overhead words are required for a

pointer to the first available block of memory and to store the value of MEMSIZE.

Dsinit. The call

```
call dsinit (MEMSIZE)
```

initializes the storage structure's pointers and sets up the list of free blocks. This call must be made before any other references to the dynamic storage area are made.

Dsget. 'Dsget' allocates a block of words in the storage area and returns a pointer (array index) to the first useable word of the block. It takes one argument -- the size of the block to be allocated (in words).

After a call to 'dsget', you may then fill consecutive words in the 'mem' array beginning at the pointer returned by 'dsget' (up to the number of words you requested in the block) with whatever information called for by your application. If you should write more words to the block than you allocated, the next block will be overwritten. Needless to say, if this happens you may as well give up and start over.

If 'dsget' finds that there is not enough contiguous storage space to satisfy your request, it prints an error message, and if you desire, calls 'dsdump' to give you a dump of the contents of the dynamic storage array.

Dsfree. A call to 'dsfree' with a pointer to a block of storage (obtained from a call to 'dsget') deallocates the block and makes it available for later use by 'dsget'. 'Dsfree' will warn you if it detects an attempt to free an unallocated block and give you the option of terminating or continuing the program.

Dsdump. The dynamic storage routines cannot check for correct usage of dynamic storage. Because block sizes and pointers are also stored in 'mem' it is very easy for a mistake in your program to destroy this information. 'Dsdump' is a subroutine that can print the dynamic storage area in a semi-readable format to assist in debugging. It takes one argument: the constant LETTER for an alphanumeric dump, or the constant DIGIT for a numeric dump.

The following example shows the use of the dynamic storage routines and uses 'dsdump' to show the changes in storage that result from each call.

```
define (MEMSIZE, 35)

pointer pos1, pos2   # pointer is a subsystem defined type
pointer dsget
DS_DECL (mem, MEMSIZE)

call dsinit (MEMSIZE)
call dsdump (LETTER) # first call

pos1 = dsget (4)
call scopy ("aaa"s, 1, mem, pos1)
call dsdump (LETTER) # second call

pos2 = dsget (3)
call scopy ("bb"s, 1, mem, pos2)
call dsdump (LETTER) # third call

call dsfree (pos2)
call dsdump (LETTER) # fourth call

stop
end
```

The first call to 'dsdump' (after 'init') produces the following dump:

```
* DYNAMIC STORAGE DUMP *
    1   3 words in use
    4   32 words available
* END DUMP *
```

The first three words are used for overhead, and 32 (MEMSIZE - 3) words are available starting at word four in 'mem'.

The second call to 'dsdump' (after the first write to dynamic storage) produces the following:

```
* DYNAMIC STORAGE DUMP *
    1   3 words in use
    4   26 words available
   30   6 words in use
          aaa
* END DUMP *
```

Note that only four characters were written, three a's and an EOS (an EOS is a nonprinting character), but two extra control words are required for each block. That block is comprised of words 30 - 35 in the array 'mem'.

The third call to 'dsdump' (after the second 'scopy') produces the following:

```
* DYNAMIC STORAGE DUMP *
    1   3 words in use
    4   21 words available
   25   5 words in use
          bb
   30   6 words in use
          aaa
* END DUMP *
```

The final call to 'dsdump' produces:

```
* DYNAMIC STORAGE DUMP *
    1   3 words in use
    4   26 words available
   30   6 words in use
          aaa
* END DUMP *
```

As you can see, the second block of storage that began at word 25 has been returned to the list of available space.

## Symbol Table Manipulation

Symbol table routines allow you to index tabular data with a character string rather than an integer subscript. For instance, in the following table, the information contained in "field1", "field2", and "field3" can obtained by specifying a certain key value (e.g. "firstentry").

| key | field1 | field2 | field3 |
|---|---|---|---|
| firstentry | 10268 | data | u |
| secondentry | 27043 | moredata | a |

All Subsystem symbol table routines use dynamic storage. Therefore, the declarations and initialization required for dynamic storage are also required for the symbol table routines; namely:

```
DS_DECL (mem, MEMSIZE)
...
call dsinit (MEMSIZE)
```

where 'mem' is an array name that can be used to reference the dynamic storage area, and MEMSIZE is a user-defined identifier describing how many words are to be reserved for items in dynamic storage. MEMSIZE must be a integer value between 6 and 32767 inclusive. For a discussion on how to estimate the amount of dynamic storage space needed in a program, you can refer back to the section on the dynamic storage routines.

A symbol table entry consists of two parts: an identifier and its associated data. The identifier is a variable length character string; it is dynamically created when the symbol is entered into a symbol table. The data associated with the symbol is treated as a fixed-length array of words to be stored or modified when the associated symbol is entered in the table and returned when the symbol is looked up. The size of the data is fixed for each symbol table -- each entry in a table must have associated data of the same size, but different symbol tables may have different lengths of data.

Mktabl. A symbol table is created by a call to the pointer function 'mktabl' with a single integer argument giving the size of the associated data array or the "node size". 'Mktabl' returns a pointer to the symbol table in dynamic storage. This returned pointer identifies the symbol table -- you must pass it to the other symbol table routines to identify which table you want to reference. A symbol table is relatively small (each table requires about 50 words, not counting the symbols stored in it), so you may create as many of them as you like (as long as you have room for them).

In the table above, if "field1" and "field3" require one word each, and "field2" requires no more than 9 words, then you can create the symbol table with the following call:

```
pointer extable
...
extable = mktabl (11)
```

The argument to 'mktabl' is 11 -- the total length of the data to be associated with each symbol.

Enter. To enter a symbol in a symbol table, you must provide two items: an EOS-terminated string containing the identifier to be placed in the table, and an array containing the data to be associated with the symbol. Of course this array must be at least as large as the "nodesize" declared when the particular symbol table was created. A call to the subroutine 'enter' with the identifier, the data array, and the symbol table pointer will make an entry in the symbol table. However, if the identifier is already in the table, its associated data will be overwritten by that you've just supplied. It is not possible to have the same identifier in the same symbol table twice.

Now, continuing our example, to enter the first row of information in the table, you can use the following statements:

```
info (1) = 10268
call scopy ("data"s, 1, info, 2)
info (11) = 'u'c
call enter ("firstentry"s, info, extable)
```

Lookup. Once you've made an entry in the symbol table, you can retrieve it by supplying the identifier in an EOS-terminated

string, an empty data array, and the symbol table pointer to the function 'lookup'. If 'lookup' can find the identifier in the table, it will fill in your data array with the data it has stored with the symbol and return with YES for its function value. Otherwise, it will just return with NO as its function value.

In our example, to access the data associated with the "firstentry" we can make the following call:

```
foundit = lookup ("firstentry"s, info, extable)
```

After this call (assuming that "firstentry" was in the table), "foundit" would have the value YES, "info (1)" would have the value for "field1", "info (2)" through "info (10)" would have the value for "field2", and "info (11)" would have the value for "field3".

Delete. If you should want to get rid of an entry in a symbol table, you can make a call to the subroutine 'delete' with identifier you want to delete in an EOS-terminated string and the symbol table pointer. If the identifier you pass is in the table, 'delete' will delete it and free its space for later use. If the identifier is not in the table, then 'delete' won't do anything.

Using our example again, if you want to delete 'firstentry' from the table, you can just make the call

```
call delete ("firstentry"s, extable)
```

and "firstentry" will be removed from the table.

Rmtabl. When you are through with a table and want to reclaim all of its storage space, you pass the table pointer to 'rmtabl'. 'Rmtabl' will delete all of the symbols in the table and release the storage space for the table itself. Of course, after you remove a table, you can never reference it again.

To complete our example, we can get rid of our symbol table by just calling 'rmtabl':

```
call rmtabl (extable)
```

Sctabl. So far, the routines we've talked about have been sufficient for dealing with symbol tables. It turns out that there is one missing operation: getting entries from the table without knowing the identifiers. The need for this operation arises under many circumstances. Perhaps the most common is when we want to print out the contents of a symbol table for debugging.

To use 'sctabl' to return the contents of a symbol table, you first need to initialize a pointer with the value zero. We'll call this the position pointer from now on. Then you call

'sctable' repeatedly, passing it the symbol table pointer, a character array for the name, a data array for the associated data, and the position pointer. Each time you call it, 'sctabl' will return another entry in the table: it will fill in the character string with the entry's identifier, fill in your data array with the entry's data, and update position in the position pointer. When there are no more entries to return in the table, 'sctabl' returns EOF as its function value.

There are two things you have to watch when using 'sctabl'. First, if you don't keep calling 'sctabl' until it returns EOF, you must call 'dsfree' with the position pointer to release the space. Second, you may call 'enter' to modify the value of a symbol while scanning a table, but you cannot use 'enter' to add a new symbol or use 'delete' to remove a symbol. If you do, 'sctabl' may lose its place and return garbage, or it may not return at all!

Here is a subroutine that will dump the contents of our example symbol table:

```
# stdump --- print the contents of a symbol table
   subroutine stdump (table)
   pointer table

   integer posn
   integer sctabl
   character symbol (MAXSTR)
   untyped info (11)

   call print (ERROUT, "*4xSymbol*12xInfo*n"s)

   posn = 0
   while (sctabl (table, symbol, info, posn) ~= EOF)
      call print (ERROUT, "*15s|*6i|*9s|*c*n"s,
               symbol, info (1), info (2), info (9))

   return
   end
```

If make a call to 'stdump' after made the entry for "firstentry", it would print the following:

```
     Symbol            Info
firstentry     | 10268|data     |u
```

## Other Routines

There are a number of miscellaneous routines that provide often needed assistance. The following table gives their names and a brief description. For full information on their use, see the Subsystem reference manual:

date  Obtain date, time, process id, login name

| | |
|---|---|
| error | Print an error message and terminate |
| follow | Follow a path and set the current and/or home directories |
| remark | Print a string followed by a newline |
| tquit$ | Check if the break key was hit |
| wkday | Determine the day of the week of any date |

# Appendixes

## Appendix A -- Implementation of Control Statements

This appendix contains flowcharts of the code produced by the Ratfor control statements along with actual examples of the code Ratfor produces.

In different contexts, a given sequence of Ratfor control statements can generate slightly different code. First, where possible, statement labels are not produced when they are not referenced. For instance, a **repeat** loop containing no **break** statements will have no "exit" label generated, since one is not needed. Second, **continue** statements are generated only when two statement numbers must reference the same statement. Finally, internally generated **goto** statements are omitted when control can never pass to them; e.g. a **when** clause ending with a **return** statement.

These code generation techniques make no fundamental difference in the control-flow of a program, but can make the code generated by very similar instances of a control statement appear quite different. Please keep in mind that the examples of Fortran code generated by 'rp' are included for completeness, and are not necessarily character-for-character descriptions of the code that would be obtained from preprocessing. Rather, they are intended to illustrate the manner in which the Ratfor statements are implemented in Fortran.

## Break

Syntax:

```
break [<levels>]
```

Function:

Causes an immediate exit from the referenced loop.

Example:

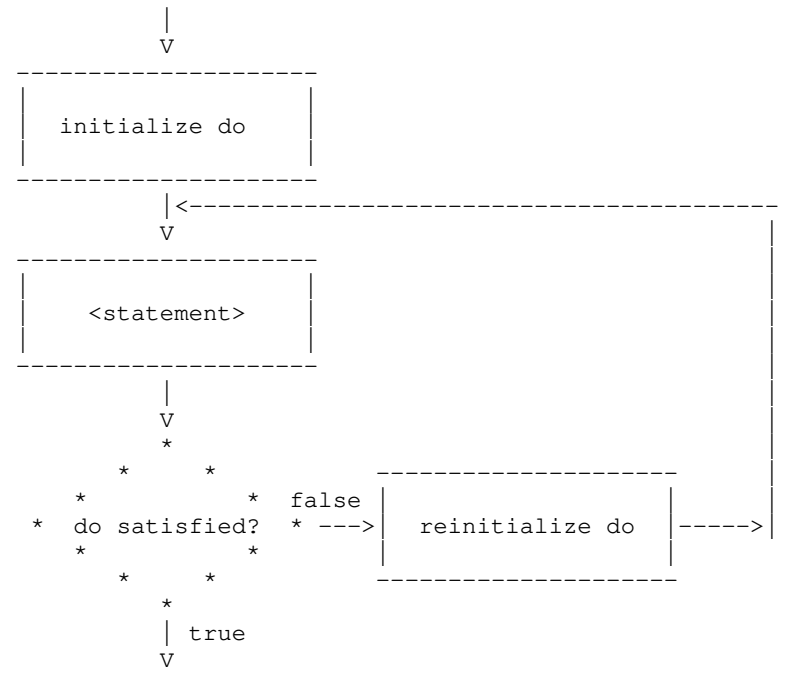
```
for (i = length (str); i > 0; i = i - 1)
   if (str (i) ~= ' 'c)
      break


      i=length(str)
      goto 10002
10000 i=i-1
10002 if((i.le.0))goto 10001
        if((str(i).eq.160))goto 10003
          goto 10001
10003 goto 10000
10001 continue
```

## Do

Syntax:

```
do <limits>
   <statement>
```

Function:




Example:

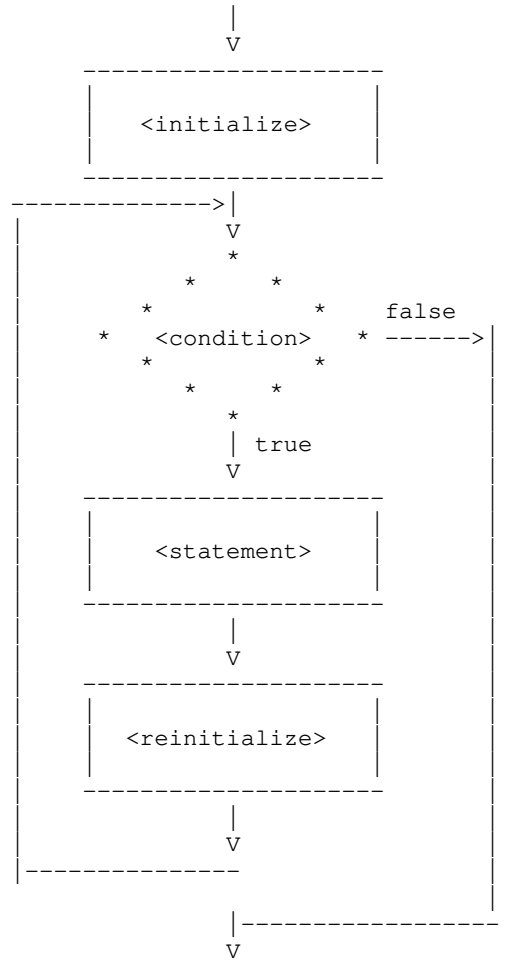
```
do i = 1, 10
   array (i) = 0


      do 10000 i=1,10
10000 array(i)=0
```

**For**

Syntax:

```
for ([<initialize>]; [<condition>]; [<reinitialize>])
   <statement>
```

Function:



Example:

```
for (i = limit - 1; i > 0; i = i - 1) {
   array_1 (i) = array_1 (i + 1)
   array_2 (i) = array_2 (i + 1)
   }
```

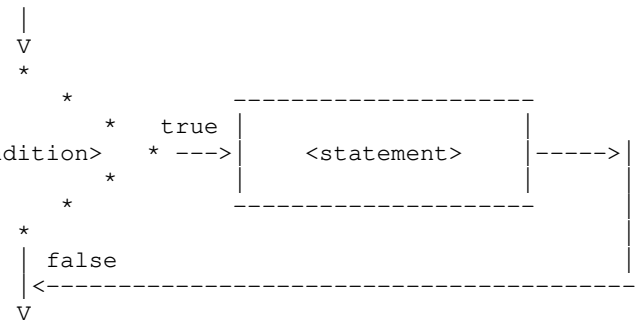
```
      i=limit-1
      goto 10002
10000 i=i-1
10002 if((i.le.0))goto 10001
        array1(i)=array1(i+1)
        array2(i)=array2(i+1)
      goto 10000
10001 continue
```

## If

Syntax:

```
if (<condition>)
   <statement>
```

Function:




Example:

```
if (a == b) {
   c = 1
   d = 1
   }
```

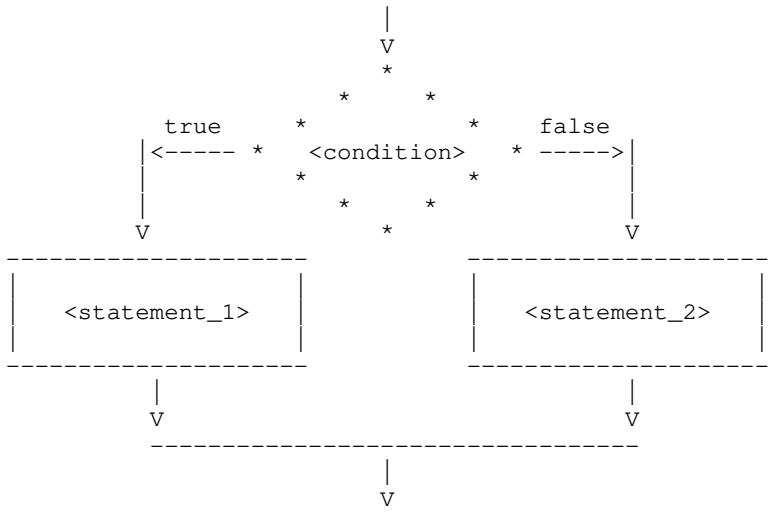
```
      if((a.ne.b))goto 10000
        c=1
        d=1
10000 continue
```

## If - Else

Syntax:

```
if (<condition>)
   <statement_1>
else
   <statement_2>
```

Function:




Example:

```
if (i >= MAXLINE)
   i = 1
else
   i = i + 1
```

```
      if((i.lt.102))goto 10000
        i=1
        goto 10001
10000   i=i+1
10001 continue
```

## Next

Syntax:

```
next [<levels>]
```

Function:

All loops nested within the loop specified by <levels> are terminated. Execution resumes with the next iteration of the loop specified by <levels>.

Example:

```
  # output only strings containing no blanks
for (i = 1; i <= LIMIT; i = i + 1) {
   for (j = 1; str (j, i) ~= EOS; j = j + 1)
      if (str (j, i) == ' 'c)
         next 2
   call putlin (str (1, i), STDOUT)
   }
```

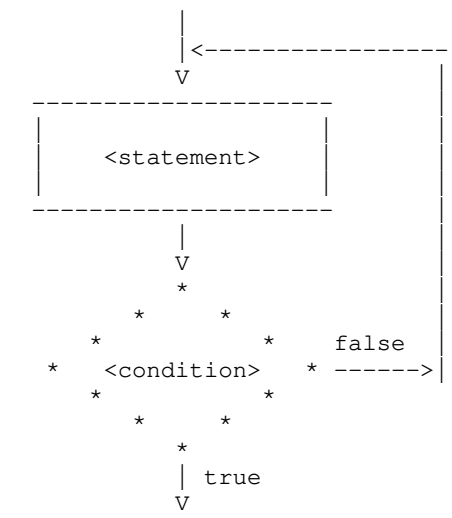
```
      i=1
      goto 10002
10000 i=i+1
10002 if((i.gt.50))goto 10001
        j=1
        goto 10005
10003   j=j+1
10005   if((str(j,i).eq.-2))goto 10004
          if((str(j,i).ne.160))goto 10006
            goto 10000
10006   goto 10003
10004   call putlin(str(1,i),-11)
      goto 10000
10001 continue
```

## Repeat

Syntax:

```
repeat
   <statement>
   [until (<condition>)]
```

Function:




Example:

```
repeat {
   i = i + 1
   j = j + 1
   } until (str (i) ~= ' 'c)


10000   i=i+1
        j=j+1
      if((str(i).eq.160))goto 10000
```

## Return

Syntax:

```
return [ '(' <expression ')' ]
```

Function:

Causes <expression> (if specified) to be assigned to the function name, and then causes a return from the subprogram.

Example:

```
integer function fcn (x)
...
return (a + 12)
```

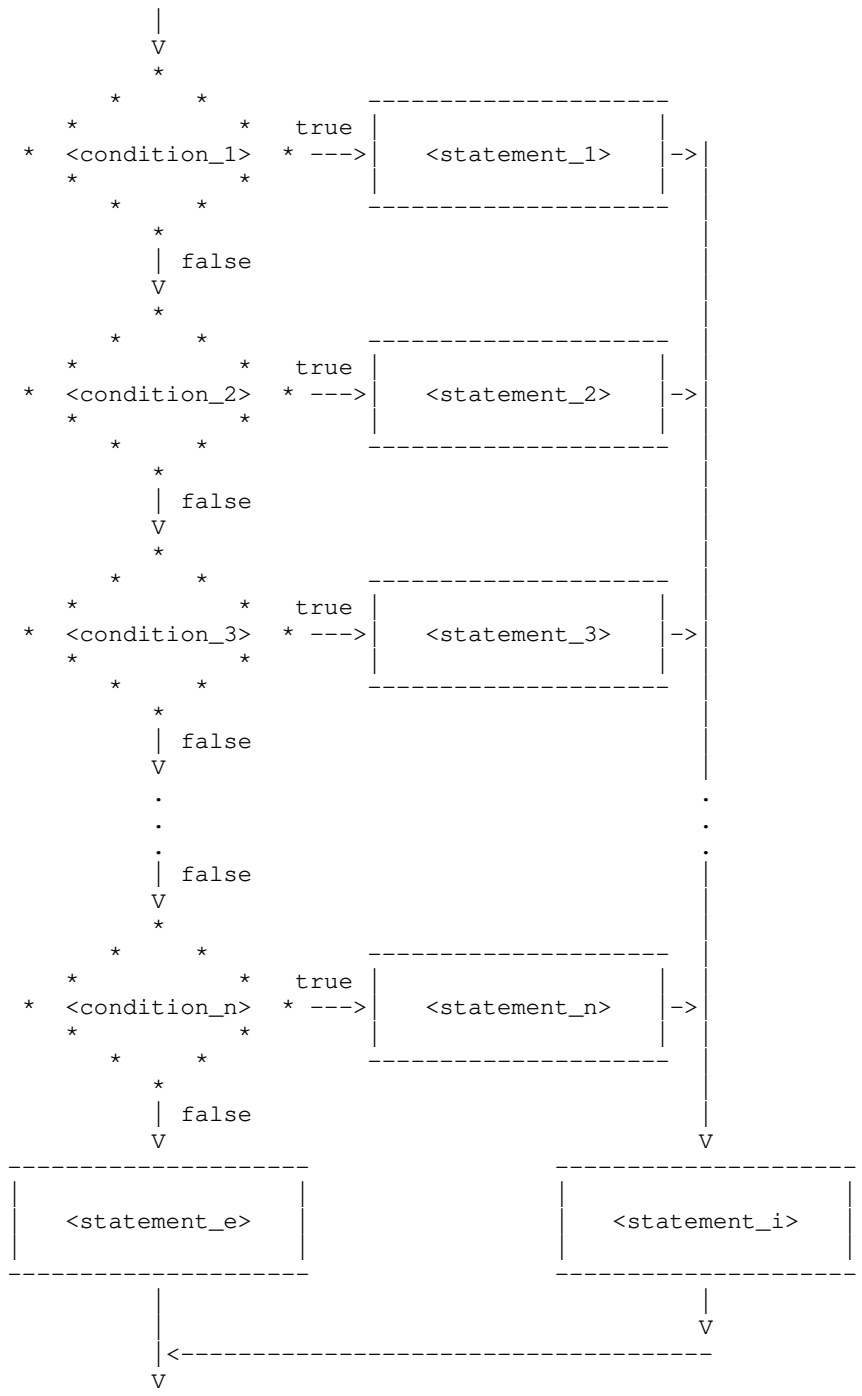
```
integer function fcn (x)
...
fcn=a+12
return
```

## Select

Syntax:

```
          select
             when (<condition_1>)
                <statement_1>
             when (<condition_2>)
                <statement_2>
             when (<condition_3>)
                <statement_3>
                  .
                  .
                  .
             when (<condition_n>)
                <statement_n>
*         [ifany
             <statement_i>]
          [else
             <statement_e>]
```

Function:


true
<condition_1>
<statement_1>
false
true
<condition_2>
<statement_2>
false
true
<condition_3>
<statement_3>
false
true
<condition_n>
<statement_n>
false
<statement_e>
<statement_i>

Example:

```
          select
             when (i == 1)
                call add_record
             when (i == 2)
                call delete_record
          else
             call code_error
```

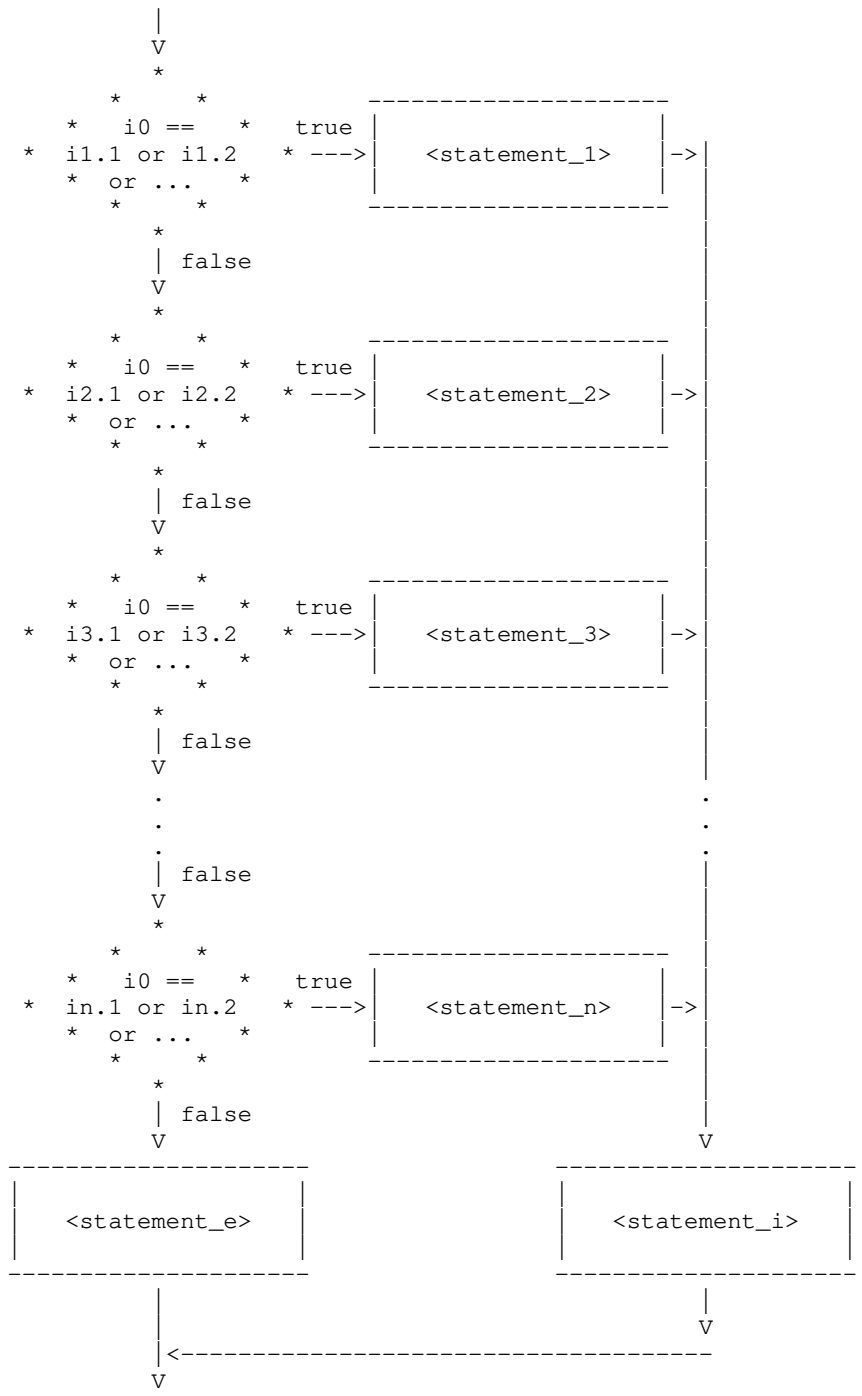
```
                goto 10001
          10002   call addre0
                goto 10000
          10003   call delet0
                goto 10000
          10001 if((i.eq.1))goto 10002
                if((i.eq.2))goto 10003
                  call codee0
          10000 continue
```

## Select (<integer expression>)

Syntax:

```
          select (<i0>)
             when (<i1.1>, <i1.2>, ...)
                <statement_1>
             when (<i2.1>, <i2.2>, ...)
                <statement_2>
             when (<i3.1>, <i3.2>, ...)
                <statement_3>
                  .
                  .
                  .
             when (<in.1>, <in.2>, ...)
                <statement_n>
*            [ifany
                <statement_i>]
             [else
                <statement_e>]
```

Function:


i0 == i1.1 or i1.2 or ...
true
<statement_1>
false
i0 == i2.1 or i2.2 or ...
true
<statement_2>
false
i0 == i3.1 or i3.2 or ...
true
<statement_3>
false
i0 == in.1 or in.2 or ...
true
<statement_n>
false
<statement_e>
<statement_i>

Example:

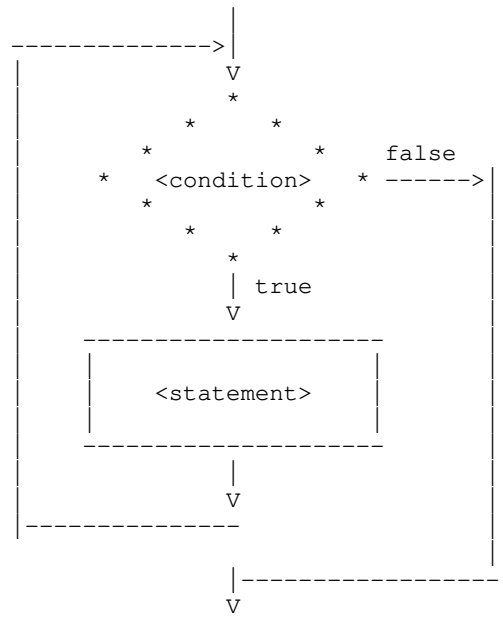
```
          select (i)
             when (4, 6, 3003)
                call add_record
             when (2, 12, 5000)
                call delete_record
          else
             call code_error


                integer aaaaa0,aaaab0
                ...
                aaaaa0=i
                goto 10001
          10002   call addre0
                goto 10000
          10003   call delet0
                goto 10000
          10001 aaaab0=aaaaa0-1
                goto(10003,10004,10002,10004,10002,
               *     10004,10004,10004,10004,10004,
               *     10003),aaaab0
                if(aaaaa0.eq.3003)goto 10002
                if(aaaaa0.eq.5000)goto 10003
          10004 continue
          10000 continue
```

## While

Syntax:

```
while (<condition>)
   <statement>
```

Function:




Example:

```
while (str (i) ~= EOS)
   i = i + 1
```

```
10000 if((str(i).eq.-2))goto 10001
        i=i+1
      goto 10000
10001 continue
```

## Appendix B -- Linking Programs With Initialized Common

The Subsystem link procedure makes the assumption that all common areas are uninitialized to allow programs to access up to 27 64K word segments of data space. A program which uses initialized common areas must be linked with one of two slightly different procedures: If the object file can be a segment directory (this is usually not a problem), you can have the object file placed in a segment directory. Just add the "-d" option to the 'ld' command line. Assuming your binary file is named "prog.b", you can use the command

```
ld -d prog.b
```

If you would rather the object program be stored in a regular file, you can use a slightly different procedure. With this procedure, the program is restricted to one segment (64K words) for both code and data space. If this limit is exceeded, no warning will be given, and unpredictable results will occur during execution. If more than 64K words of space is required, the common areas must be initialized at run time, or the program must be placed in a segment directory.

This modification to the link procedure is as follows: the option string "-s 'co ab 4000'" must appear on the 'ld' command line before the first binary file. For instance, if the file "prog.b" contained a program with **block data** statements, an 'ld' command to link it might appear as follows:

```
ld -s 'co ab 4000' prog.b
```

The executable program would be placed in the file "prog.o".

## Appendix C -- Requirements for Subsystem Programs

This appendix gives the technical specifications of requirements for programs that run under the Subsystem. It is included to allow non-Ratfor programs to run under the Subsystem.

32S and 16S addressing modes

- There is no support for the execution of these addressing modes.

64R & 32R addressing modes

- The 64R mode library routines cannot access the Subsystem common areas, so 32R and 64R mode programs cannot execute under the Subsystem.

64V addressing mode

- Segments '4040 and '4041 may not be disturbed.
- When a Subsystem program is executed, the stack is already constructed in segment '4041. However, the executing program may rebuild it if desired.
- Programs that use native i/o routines must inform their native i/o routines of the Subsystem (if they wish to take advantage of Subsystem i/o) by calling the proper initialization routines, i.e. 'init$f' for Fortran 66 and Fortran 77, 'init$p' for Pascal and 'init$plg' for PL/I G.
- The program must terminate with a call to the Subsystem routine 'swt' at the end of its execution or its main program must return to its caller. A **stop** statement in Ratfor will be transformed into a call to 'swt'.
- The program must not tamper with any file units already open by the Subsystem. It should always use a Subsystem or Primos call to obtain an unused file unit.
- The program must be in a P300 format runfile or a SEG-compatible segment directory.
- If the program is in a P300 format runfile, it must have been loaded by the modified version of the segmented loader, 'swtseg', or the entry control block for the main program must be at location '1000 in segment '4000.
- The runfile must not expect any segment other than '4000 to be initialized before execution, unless it is loaded from a SEG-compatible segment directory.
- The default load sequence produced by 'ld' will correctly

link programs requiring up to 64K words of procedure (code) and linkage (initialized local data) frames. Up to 27 64K word segments may be used for uninitialized common blocks. Up to 64K words of local data may be allocated on the stack. Programs loaded from SEG-compatible segment directories may be as large as the operating system permits, as long as they do not modify segments '4040 and '4041.

32I addressing mode

- Programs in 32I mode may be executed under the Subsystem subject to the same constraints as 64V mode programs.

# Appendix D -- The Subsystem Definitions

The file "=incl=/swt_def.r.i" contains Ratfor **define** statements for all the symbolic constants required to use the routines in the Subsystem support library. This appendix describes the more frequently used constants and the constraints placed on them.

## Characters

ASCII Mnemonics. Character definitions for the ASCII control characters NUL, SOH, STX, ..., GS, RS, US, as well as SP and DEL.

Control characters. Character definitions for the ASCII control characters CTRL_AT, CTRL_A, CTRL_B, ..., CTRL_LBRACK, CTRL_BACKSLASH, CTRL_RBRACK, CTRL_CARET, and CTRL_UNDERLINE.

BACKSPACE Synonym for ASCII BS.

TAB Synonym for ASCII HT.

BELL Synonym for ASCII BEL.

RHT Relative horizontal tab character (used for blank compression in Primos text files).

RUBOUT Synonym for ASCII DEL.

## Data Types

bits Bit strings (16 bit items).

bool Boolean (logical) values: .true. and .false. (16 bit items).

character Single right-justified zero-filled character (scalar), or a string of these characters terminated by an EOS (array).

file_des File descriptor returned 'open', 'create', etc.

file_mark File position returned by 'seekf'.

longint Double precision (32 bit) integer.

longreal Double precision (64 bit) floating point.

pointer Pointer for use with dynamic storage and symbol table routines.

## Macro Subroutines

fpchar (<packed array>, <index>, <character>) Fetches <character> from <packed array> at character position <index> and increments <index>. The first character in the array is position zero.

spchar (<packed array>, <index>, <character>) Stores <character> in <packed array> at character position <index> and increments <index>. The first character in the array is position zero.

getc (<char>) Behaves exactly like 'getch', except the character is always obtained from STDIN.

putc (<char)) Behaves exactly like 'putch', except the

character is always placed on STDOUT.
SKIPBL (<character array>, <index>) Increments <index> until the corresponding position in the character array is non-blank.
DS_DECL (<ds array name>, <ds array size>) Declares the dynamic storage array with the name <ds array name> with size <ds array size>.

## Language Extensions

ARB Used when dimensioning array parameters in subprograms (since their length is determined by the calling program, not the subprogram).
FALSE Represents the Fortran logical constant .false.
IS_DIGIT (<char>) Logical expression yielding TRUE if <char> is a digit.
IS_LETTER (<char>) Logical expression yielding TRUE if <char> is an upper or lower case letter.
IS_UPPER (<char>) Logical expression yielding TRUE if <char> is an upper case letter.
IS_LOWER (<char>) Logical expression yielding TRUE if <char> is a lower case letter.
SET_OF_UPPER_CASE Sequence of 26 character constants representing the upper case letters for use in the **when** parts of **select** statements.
SET_OF_LOWER_CASE Sequence of 26 character constants representing the lower case letters for use in **when** parts of **select** statements.
SET_OF_LETTERS Sequence of 52 character constants representing the upper and lower case letters for use in **when** parts of **select** statements.
SET_OF_DIGITS Sequence of 10 character constants representing the digits for use in **when** parts of **select** statements.
SET_OF_CONTROL_CHAR Sequence of 32 character constants representing the first 32 ASCII control characters for use in **when** parts of **select** statements.
TRUE Represents the Fortran logical constant .true.

## Limits

CHARS_PER_WORD Maximum number of packed characters per machine word.
MAXINT Largest 16-bit integer.
MAXARG Maximum length of a command line argument (EOS-terminated character string).
MAXCARD Maximum input line length (excluding the EOS).
MAXDECODE Maximum size of string processed by 'decode'.
MAXLINE Maximum input line length.
MAXPAT Maximum size of a pattern array.
MAXPATH Maximum size of a Subsystem pathname.
MAXPRINT Maximum number of character that can be output by a single call to 'print'.
MAXTREE Maximum number of characters in a Primos tree

| | |
|---|---|
| | name. |
| MAXFNAME | Maximum number of characters in a simple file name. |

## Standard Ports

| | |
|---|---|
| STDIN | Standard input 1. |
| STDIN1 | Standard input 1. |
| STDIN2 | Standard input 2. |
| ERRIN | Standard input 3. |
| STDIN3 | Standard input 3. |
| STDOUT | Standard output 1. |
| STDOUT1 | Standard output 1. |
| STDOUT2 | Standard output 2. |
| ERROUT | Standard output 3. |
| STDOUT3 | Standard output 3. |

## Argument and Return Values

| | |
|---|---|
| ABS | Request absolute positioning ('seekf'). |
| REL | Request relative positioning ('seekf'). |
| DIGIT | Character is a digit ('type'). |
| LETTER | Character is a letter ('type'). |
| UPPER | Map to upper case ('mapstr'). |
| LOWER | Map to lower case ('mapstr'). |
| READ | Open file for reading. |
| WRITE | Open file for writing. |
| READWRITE | Open file for reading and writing. |
| EOF | End of file (guaranteed distinct from all characters and from OK and ERR). |
| OK | No error (guaranteed distinct from all characters and from EOF and ERR). |
| ERR | Error occurred (guaranteed distinct from all characters and from EOF and OK). |
| EOS | End of string (guaranteed distinct from all characters). |
| LAMBDA | Null pointer (guaranteed distinct from all pointer values). |
| PG_END | Make 'page' return after the last page of input. |
| PG_VTH | Make 'page' use the VTH routines when writing to the terminal. |
| YES | Affirmative response (guaranteed distinct from NO). |
| NO | Negative response (guaranteed distinct from YES). |

## Appendix E -- 'Rp' Reserved Words

The following identifiers are reserved keywords in Ratfor and cannot be used as identifiers. 'Rp' will not diagnose the use of reserved keywords as identifiers; results of misuse will be unreasonable behavior such as misleading error messages and mis-ordered Fortran code.

```
blockdata
break
call
case
common
complex
continue
data
define
dimension
do
doubleprecision
else
end
equivalence
external
for
forward
function
goto
if
ifany
implicit
include
integer
linkage
local
logical
next
parameter
procedure
real
recursive
repeat
return
save
select
shortcall
stackheader
stmtfunc
stop
string
stringtable
subroutine
trace
undefine
until
when
while
```

## Appendix F -- Command Line Syntax

'Rp' provides a rich set of processing options to allow the user much flexibility and control over the code which is produced. The command line syntax is as follows:

```
rp [-{a | b | c | d | f | g | h | l | m | p | s | t | v | y}]
   [-o <output_file>] {<input_file>} [-x <translation file>]
```

The following is a full description of each option:

a Abort all active shell programs if any errors were encountered during preprocessing. This option is useful in shell programs like 'rfl' that wish to inhibit compilation and loading if preprocessing failed. By default, this option is not selected; that is, errors in preprocessing do not terminate active shell programs.

b Do not map long indentifiers or identifiers containing upper case letters into unique six character Fortran identifiers. This option is useful if your Fortran compiler will accept names longer than six characters.

c Include statement-count profiling code in the generated Fortran. When this option is selected, calls to the library routines 'c$init', 'c$incr', and 'c$end' will be placed (unobtrusively) in the output code. When the preprocessed program is run, it will generate a file named "_st_count" containing execution frequencies for each line of source code. The utility program 'st_profile' may then be used to combine source code and statement counts to form a readable report.

d Inhibit generation of the long-name dictionary. Normally, a dictionary listing all long names used in the Ratfor program along with their equivalent short forms is placed at the end of the generated Fortran as a series of comment statements. This option prevents its generation.

f Suppress automatic inclusion of standard definitions file. Macro definitions for the manifest constants used throughout the Subsystem reside in the file "=incl=/swt_def.r.i". 'Rp' will process these definitions automatically, unless the "-f" option is specified.

g Make a second pass over the code and remove GOTOs to GOTOs generated in Ratfor control structures. Use of this option lengthens preprocessing time significantly, but can result (sometimes) in a 2-5% speedup of the object program.

h Produce Hollerith-format string constants rather than

quoted string constants. This option useful in producing character strings in the proper format needed by your Fortran compiler.

l Include Ratfor line numbers in the sequence number field of the Fortran output. This may be useful in tracking down the Ratfor statement that caused a Fortran syntax error. By default, no sequence field is generated.

m Map all identifiers to lower case. When this option is selected, 'rp' considers the upper case letters equivalent to the corresponding lower case letters, except inside quoted strings.

p Emit subroutine profiling code. When this option is selected, 'rp' places calls to the library routines 't$entr', 't$exit', and 't$clup' in the Fortran output, and creates a text file named "timer_dictionary" containing the names of all subprograms seen by the preprocessor. When the profiled program is run, a file named "_profile" is created that contains timing measurements for each subprogram. The utility program 'profile' may then be used to print a report summarizing the number of times each subprogram was called and the total time spent in each.

s Short-circuit all logical conditions. The order of evaluation of logical operands in Fortran is unspecified; that is, in the expression "a&b" there is no guarantee that "a" will be evaluated before "b". Occasionally this creates inconveniences; one would like to say something like "if(i>1&array(i)~=0)...". 'Rp' supplies the short-circuit logical operators "&&" and "||" (read "andif" and "orif") for these occasions. Both operators evaluate their left operands; if the value of the logical expression is predictable solely on the basis of the value of the left operand, then the right operand remains unevaluated and the correct expression value is yielded. Otherwise the right operand is evaluated and the proper expression value is determined. The "-s" option may be used to automatically convert all "logical and" operators in a program to "andifs," and all "logical or" operators to "orifs." In addition to improving program portability, this option may also reduce execution time. By default, however, this option is not in effect.

t Trace subprograms. When a program preprocessed with the "-t" option is run, an indented trace of the subprograms encountered will be printed on ERROUT. This trace output is generated by calls to the library routine 't$trac' that are inserted automatically by 'rp'.

v Output "standard" Fortran. This option causes 'rp' to

generate only standard Fortran constructs (as far as we know). This option does not detect non-standard Fortran usage in Ratfor source code; it only prevents 'rp' from generating non-standard constructs in implementing its data and control structures. Programs preprocessed with this option are slightly larger and slower; the intermediate Fortran and binary files are approximately 10% larger.

x Translate character codes. 'Rp' uses the character correspondences in the <translation file> to convert characters into integers when it builds Fortran DATA statements containing EOS-terminated or PL/I strings. If the option is not specified, 'rp' converts the characters using the native Prime character set. The format of the translation file is documented below.

y Do not output "call swt". This option keeps 'rp' from generating "call swt" in place of all "stop" statements.

The remainder of the command line is used to specify the names of the Ratfor input file(s) and the Fortran output file. If the "-o" option, followed by a filename, is selected, then the named file is used for Fortran output. Any remaining filenames are considered Ratfor source files. If no other file names are specified, standard input is read. If the "-o" option is not specified, then the output filename is constructed from the first input filename by changing a ".r" suffix (if present) to ".f". If the ".r" suffix is not present, the output filename is the input filename followed by the suffix ".f".

The format of the translation file used with the "-x" option is as follows. Each line contains descriptions of two characters: the Prime native character to be replaced, and the character value to replace it. These descriptions may be any one of the following: a single non-blank Prime ASCII character, a number in a format acceptable to 'gctoi' (must be more than one digit), or an ASCII mnemonic acceptable to 'mntoc'. In addition, the character to be replaced may also be the mnemonic "EOS" to indicate that the value of the end-of-string indicator is to be changed. For example, here is a portion of the table for converting the EBCDIC character set:

```
A 16rc1
B 16rc2
...
Z 16re9
0 16rf0
...
9 16rf9
SP 16r40
```