Lawrence Berkeley National Laboratory

Recent Work

Title
A SUMMARY OF THE RATFOR LANGUAGE, AN EXTENDED PORTABLE DIALECT CALLED RFP, ITS STYLE AND FLAVOR, AND DETAILS OF ITS IMPLEMENTATION ON THE PDP-10

Permalink
https://escholarship.org/uc/item/6qx134bx

Author
Figen, J.

Publication Date
1981-09-01
A SUMMARY OF THE RATFOR LANGUAGE, AN EXTENDED PORTABLE DIALECT CALLED REP, ITS STYLE AND FLAVOR, AND DETAILS OF ITS IMPLEMENTATION ON THE PDP-10

Joel Figen

September 1981

TWO-WEEK LOAN COPY

This is a Library Circulating Copy which may be borrowed for two weeks. For a personal retention copy, call Tech. Info. Division, Ext. 6782
DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
A Summary of the Ratfor Language,
An Extended Portable Dialect Called REP,
Its Style and Flavor,
And Details of its Implementation on the PDP-10

by

Joel Figen

Information Methodology Research Project
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

September 10, 1981

This work was supported by the U. S. Department of Energy
A Summary of the Ratfor Language,
An Extended Portable Dialect Called REP,
Its Style and Flavor,
And Details of Its Implementation on the PDP-10

Author: Joel Figen
Date: September 10, 1981

Ratfor (RATional FORtran) is a dialect of Fortran that allows structured programming and the use of simple macros. It is the language of the Software Tools package, and is documented in the book Software Tools (1). It has proved significantly easier than Fortran to read, write, and understand. Although debugging is slightly harder in Ratfor than in Fortran, there is usually less of it to do, since Ratfor lends itself to better program design. Ratfor operates as a preprocessor to any existing Fortran system. It is relatively easy, using Ratfor, to write programs that are portable with little or no change to any environment that supports standard Fortran.

REP (Ratfor Extended and Portable) is an extended version of Ratfor. It is fully upward compatible with the Addison-Wesley translator, though there are a few divergences from certain Unix (2) and Software Tools User Group (3) dialects. The principal feature of REP is its fully developed macro facility, a language within a language, capable of doing such things as creating new data types, data structuring, recursive procedures, and much more, portably, and in the spirit of Ratfor, but there are many other lesser though nevertheless important extensions. Henceforth, in this document, the term "Ratfor" refers to the Addison-Wesley dialect, and the subset of REP that is compatible with it. Extended features of REP will be indicated by prefixing "REP:" at the left margin.

SOURCE FORMAT

The syntax of Ratfor makes it concise and visually attractive. It has only the minimum of punctuation. Input is free-form. Statements may appear anywhere on the input line. The end of a line generally marks the end of a statement, but certain lines that are obviously not finished automatically continue onto the next line. This includes parenthesized conditions in control statements (if, while, until, for), and lines ending with a comma.

Multiple statements may appear on one line if separated by semicolons. A semicolon alone is also a null statement. The comment convention, a sharp (#) anywhere, means that the rest of the line is commentary, so Ratfor ignores it. Quoted strings are converted to nH...'s; both single and double
quotes are acceptable, but in any string they have to match. Blanks (and tabs) are significant in that they must be used to separate alphanumeric words.

SPECIAL OPERATORS

Certain operators have new representations in Ratfor, although the regular Fortran forms are also acceptable:

<table>
<thead>
<tr>
<th>Ratfor</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>.GT.</td>
</tr>
<tr>
<td>&lt;</td>
<td>.LT.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>.GE.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>.LE.</td>
</tr>
<tr>
<td>==</td>
<td>.EQ.</td>
</tr>
<tr>
<td>~=</td>
<td>.NE.</td>
</tr>
<tr>
<td>&amp;</td>
<td>.AND.</td>
</tr>
<tr>
<td>:)</td>
<td>.OR.</td>
</tr>
<tr>
<td>~</td>
<td>.NOT.</td>
</tr>
</tbody>
</table>

REP: () (DUMARG) -- allowing functions without arguments

CONTROL STRUCTURES

Ratfor has the right control structures for goto-less programming. Fortran control statements are also available, with the exception of the three branched arithmetic if, and the old fashioned do statement. Ratfor, as you will see, has newer ideas. Braces {} are used to group multiple statements.

1. If statement: if (condition) statement or if (condition) statement else statement

This is the basic forward branching or selection structure. A dangling else groups with the latest previous un-else’ed if. You can override this with braces, but that’s seldom desirable.

Example:

```plaintext
if (x == 1.0)
  return
else if (y == x)
  j = 2
else {
  j = 3
  call ugotit
}
```

2. While statement: while (condition) statement
This is a simple loop, that repeated as long as condition is true. If condition is false, the loop is skipped. Example:

```c
while (getc(c) != 0)
    call putc(c)
```

3. Repeat statement: repeat statement or repeat statement until (condition)

Without an until-clause, this is an infinite loop. You get out of it with a break, goto, or return.

With the until-clause, it is equivalent to

```c
repeat {
    statement(s)
    if (condition)
        break
}
```

4. For statement: for (initial; condition; reinit) statement

This is equivalent to:

```c
initial
while (condition) {
    statement
    reinit
}
```

Example:

```c
for ( i = 1; i < 10; i = i+1 )
    array(i) = 0
```

In the for statement, the init, condition, or reinit can be omitted.

In all of the looping structures, the body of the loop can be omitted, e.g.

```c
while (getc(c) == BLANK) ;
```

This would skip past blanks on the standard input. The semicolon marks the omission.

REP: A for statement can have multiple init and reinit clauses. This allows a loop to have several indices, e.g.

```c
for (i=1, j=10, k=100; i < 20; i=i+1, j=j-2, k=k*3)
```

This raises the possibility of writing entire loops in a single statement. Restraint is indicated.
REP: The conditional expression in an if, while, until, or for has extended syntax. There are 3 additional operators:

1. !! (ORIF)
2. && (ANDIF)
3. , (AFTER)

ANDIF and ORIF work like AND and OR (& and !) except that left to right evaluation is guaranteed, thereby controlling side effects. Both operands of these operators must be Fortran expressions that evaluate to a truth value. REP actually breaks up conditionals containing ANDIF or ORIF into a series of Fortran IF statements.

AFTER is used to introduce computations that must be performed before a conditional test is made. The preliminary computations are called "preconditions". Each precondition must be a single Fortran statement, normally an assignment or a call, although a read would also be possible. A precondition may not be a control statement. (if, do, goto, etc.) Each precondition is followed by a comma. Preconditions may be used only at the beginning of a conditional expression, or after an ANDIF, an ORIF, or another precondition.

RESTRICTION: ANDIF and ORIF may not be mixed in a single conditional, and must be used only at the outermost parenthetical level of a conditional expression. If this sounds like too much of a restriction, reflect for a moment on the fact that conditionals without parentheses are easiest to read, especially if they have only and's or only or's in them, and that the regular AND and OR are available, and can be mixed with ANDIF and ORIF. The precedence of ANDIF and ORIF is lower than that of any Fortran operator, or, equivalently, these operators take Fortran expressions as operands.

examples:

if (c = getc(), c == BLANK ; c == TAB)

First getc is called,
then the result is tested

c = getc(c)
if (c == BLANK ; c == TAB)

The same thing is straight Ratfor

If (getc(c) == BLANK ; c == TAB)

Effectively the same thing.
if (j=1, k=2, x == y || j =55, k == z)
call hoohah

j is set to 1, k is set to 2, then x is compared with y. If this comparison succeeds, the call to hoohah is made without further ado. But if the comparison fails, j is set to 55 and k is set to 3. If this comparison succeeds, the call to hoohah is made, but if it fails, control passes to the next statement. One way to read a statement like this is "if AFTER j equal 1 and AFTER k equal 2, x is equal to y ORIF AFTER j equal 55, c is equal to z..." If it sounds right, chances are it is right. In this case, it sounds as bad as it looks.

6. Do statement

This is the same as the Fortran do statement, except that the statement number must be omitted. The range of the loop is one statement, or one bracketed group of statements. Example:

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

REP: REP also accepts the Fortran do statement (with a statement number).

7. Break and Next.

These can be used to control looping, in conjunction with the loop structures above. Next skips the rest of the loop, and gets on to the test for the next iteration. In the case of a for-loop, next branches to the reinit clause. The behavior of the next-statement constitutes an exception to the equivalences stated above, namely that a repeat-until is equivalent to a repeat and a for is equivalent to a while. Break exits from the loop immediately. Only one level of looping can be so controlled, so, if you need to break or next out of two loops, you need a goto.

REP: A break or next can control multiple levels of loops. This is done by putting a digit after the word break or next to indicate the number of levels involved, e.g.

```fortran
for (...) {
   repeat {
      next          # to "until"
      next 2        # to the reinit of the for
```

Rep Summary, Page 5
break # to "other statements"
break 2 # to "rest of program"
} until (...) other statements
) rest of program

SIMPLE MACROS AND THE DEFINE STATEMENT

The statement

    define(macroname,anything)

declares that from that point on in the program, wherever Ratfor sees "macroname", which must be an alphanumeric token, it shall replace it with "anything", except within a word or in a quoted string. The replacement text can consist of any printable characters including white space and NEWLINES, with the restriction that parentheses in it must be balanced. Macro names may be arbitrarily long, and case is significant in macro names.

Once you have defined a macro, you can redefine it as many times as you like.

Defines can be nested. Thus

    define(select,define(case,if define(case,else if))
    define(otherwise,else)

has the effect of causing the first "case" after a "select" to expand to an "if", and each subsequent "case" to expand to an "else if". This creates a new way of writing a multiple branch structure:

select
  case (x == 1) can
    statement1 be
    if (x == 1)
      statement1
  case (x == 2) written
    statement2 in
    else if (x == 2)
      statement2
  otherwise place
    statement3 of
    else
The select-case structure is sometimes easier to understand than the corresponding if-then structure.

REP: Rep has greatly expanded the macro facility. This is discussed later.

TEXT INCLUSION AND THE INCLUDE STATEMENT

The statement

Rep Summary, Page 6
include filename

causes the contents of the named file to replace the include statement and the rest of the line it's on. This is especially useful when the same declarations, such as a description of a common block, must be used in several subprograms.

Includes may be nested to a fairly deep level, usually at least five.

REP: REP allows the filename to be a quoted string. This allows the use of file names that are not legal Ratfor tokens.

RATFOR SYNTAX

The following is a partial BNF syntax of the Ratfor language, representing the part of the language recognized by Ratfor-to-Fortran translator. Ratfor itself can't understand Fortran, so Ratfor doesn't even try to parse those nonterminals (grammatical constructs) that Fortran knows about. These are eventually recognized by Fortran, so if they're wrong, Fortran will (hopefully) give a diagnostic. Included in this category are "condition", which should be a Fortran logical expression, "initialize" and "reinit", which should each be a single Fortran assignment or call statement, and "limits", which is the variable and limits part of a Fortran do statement.

The capitalized words in the grammar below are the syntactic terminals (Basic tokens) of the grammar, as are most of the punctuation marks other than ; and :. All the word tokens represent keywords, spelled the same way as the token name, except that the keywords do not have to be capitalized, except for INTEGER, which represents any string of digits, and OTHER, which represents any statement that doesn't begin with one of the other keywords. The punctuation marks represent themselves.

```
program : full-statement
         : program full-statement

full-statement : statement ; label statement
                 : INTEGER

statement : IF ( condition ) statement
           : IF ( condition ) statement ELSE statement
           : WHILE ( condition ) statement
           : FOR ( initialize; condition; reinit ) statement
           : REPEAT statement
           : REPEAT program UNTIL ( condition )
```

Rep Summary, Page 7
REP: SYNTAX OF REP

The following is a partial BNF syntax of the REP language. All the remarks made about the Ratfor syntax above apply here as well.

REP contains more pass-the-buck nonterminals than RATFOR. Included in this category are "fortcondition", which should be a Fortran logical expression, "fortimperative", which should each be a single Fortran assignment or call statement, and "limits", which is the variable and limits part of a Fortran do statement.

This syntax also omits all macros, even though some of them are built into the translator and look like statements. Included in this category are main_entry, normal_exit, define, and include. For documentation on these, see the section on built in macros.

```
program : full-statement
  program full-statement
full-statement : statement
  label statement
label : INTEGER
statement : IF ( condition ) statement
  IF ( condition ) statement ELSE statement
  WHILE ( condition ) statement
  FOR ( initialize; condition; reinit ) statement
  REPEAT statement
  REPEAT statement UNTIL ( condition )
DO limits statement
DO INTEGER limits
BREAK
BREAK INTEGER
NEXT
NEXT INTEGER
STRING ["] id [(INTEGER)] [at equiv] str-init
{ program }
OTHER
```
equiv : a Fortran subscripted array name
condition : subcondition
subcondition : fortran-imperative , condition
andif-expr : && for-conditional
orif-expr : :: fort-conditional
initialize : fortran-imperative
reinit : initialize , fortran-imperative
id : any fortran identifier
str-init : str-init-elt
str-init-elt : "string" ; 'string' ; INTEGER

REP: ADDITIONAL FEATURES OF REP

REP has several additional features worthy of mention here.

1. Statement reordering: REP allows type declarations, data statements and executable statements to mingle freely in any subprogram.

2. Name shortening: REP allows identifiers of any length, up to the maximum for a token, which is about 200 characters. If longer than 6 characters, identifiers are shortened automatically. The replacement names are guaranteed to begin with the same letter as the original, and to consist of a letter followed by 5 digits. To prevent a name from being shortened, enclose it in percents, which REP will remove. e.g.

   define very_long_name %very_long_name%

3. Continuation symbol: REP has a very general convention for line and word continuations. The continuation character is the Ascii underline or backarrow (_). You can use it in a variety of ways

   - To continue a statement or macro body
   - To punctuate any alphanumeric or quoted token
   - You may put a comment after it
Here's how it works: In the lexical scanner, i.e. before REP has even looked at the syntax of a statement, whenever an underline is encountered, REP discards it and looks ahead. If there's nothing on the same line but air (space or a comment), REP discards the rest of the line and all leading white space on the next line. If, on the other hand, there are other characters on the same line, REP proceeds as though nothing had happened, discarding only the underline. If the next character is an underline, REP accepts it as text. This happens even in a quoted string.

4. Transmitted comments: In REP, you can transmit comments into the Fortran intermediate code. A transmitted comment is indicated by beginning with two or more \'. The comment is transmitted as a regular Fortran comment, on a line by itself, with a C in column 1.

5. Octal constants: REP treats any alphanumeric token that begins with 0 (zero) as an octal integer, and converts it to decimal. BUG ALERT: Don't use real constants beginning with zero. REP will think they're octal.

6. Character code independence: REP programs can be made independent of the character codes used in the target environment. This is accomplished by redefining the names of the characters and recompiling. The string declaration is sensitive to the character names, and responds automatically to character redefinitions. This technique is reasonably robust, but not idiot proof. Collating sequences can be a problem, and library routines may have to be recompiled. The REP translator itself is character code DEPENDENT. The normal way of running it in a hostile environment is to translate its input and output.

7. Extended string declaration: The syntax can also be represented as

```
string [$] name [(length)]
    [at array(subscripts)] [initializer]
```

The []'s mean that the stuff inside them is optional. The line break shown here is false. It's just to fit the pattern on the page.

The $ means that the string doesn't end with an EOS.

The length, if specified, becomes the actual length of the string, regardless of the initializer. If you specify a length shorter than the initializer, the initializer is truncated, and you get a warning message. In the case of a non-EOS ($) string, if the length is longer than the initializer, the string is padded with blanks on the right.

The at-clause, if specified, generates an equivalence
statement to put the string into a larger structure.

The initializer can consist of one or more of the following, in series, without commas between them:

- strings quoted with single or double quotes.
- integers (unquoted, octal or decimal).
- simple macros defined as quoted strings or integers.

If there are multiple initializers, their values are concatenated. Initializers that are unquoted integers each become a single character. This is the way of introducing unprintable characters and special codes into a string.

The initializer can be omitted if a length is specified. The string is then initialized to a null string, i.e. a single EOS in column 1, or if $ is specified, all blanks.

There are 3 hidden features:

1. The type of the generated string is whatever type character is defined as at the time.

2. Codes generated from quoted strings are whatever values are in effect for the standard names of ascii characters at the time.

3. The length of the string is remembered and can be used in computation. This is done by defining it as a simple macro. The name of the macro is stringname.LENGTH, where the name of the string is spelled casewise exactly as declared, and "LENGTH" is always in caps. This feature must be enabled by a .mode macro before use.
REP PORTABILITY CONSIDERATIONS

Rep is provided in two forms, as was the old Addison-Wesley Ratfor translator. There is a portable version, in Fortran, called the bootstrap, and the full source in Ratfor. The bootstrap is originally created from the source, and can translate the source.

Unlike the Addison-wesley translator, the REP source file requires no includes, since all the common blocks and definitions are already in it. The common blocks are defined as macros and expanded where needed. All of the standard macros are likewise included in the source, and the standard macro library file is in fact generated as a byproduct of translating the source.

Also unlike the Addison-wesley translator, the REP source file contains conditional assembly statements to generate any of 3 separate versions of REP: the bootstrap, the dec10, or the dec20 version. It is recommended That you make your changes in such a way as to add machine specific versions to the repertoire, and to keep all versions current. Typically, machine dependencies will affect primarily the standard macros, at the beginning, and the main program, which just opens files and calls the parser.

The bootstrap translator contains quick and dirty portable versions of the i/o primitive routines needed to run REP. These will be EXTREMELY slow, because of the i/o, and your error messages will be garbled, because the routines remark and remark can't really be made portable in Fortran-66. Also, the include, saveframe, and use macros will only work if the filename is a fortran unit number that has previously been connected to the desired file. A little work here would go a long way. What is needed is for open and close to access files by name, rather than number. Other than that, the bootstrap is equivalent in most ways to the production translators.

Once you have the bootstrap running and able to compile itself, which should be supremely easy, it is recommended that you rewrite the i/o primitives and the main program to suit your environment and taste, preferably after groking (1). In most cases, you should write the i/o in some language other than Fortran. REP operates on Ascii characters internally. Please leave it that way. Programs written in REP may use any code you desire, by redefining the macros that name characters.

REP was developed on a DECsystem-20 (3) using Fortran-20 (4) under TOPS-20 version 3a. It has run successfully under TOPS-10 (5) on a DECSYSTEM-10 (3), which is not really a port. Fortran-20 Is a fairly forgiving environment, so there may be portability errors undetected, but the language is about as close to the 1966 standard as I could make it. There is one glaring exception: I made no effort to hold subscript
expressions to the 1966 sanctioned form, \((c*v + k)\). This restriction runs counter to all compiler hacking sense, even for 1966, and most implementors have allowed general integer expressions as subscripts. This relaxation is especially important for REP, in which macros involving subscripting are often used for data structuring. The restriction was removed in the 1977 standard. However, any departure from a standard is dangerous when portability is the goal. Already one potential target system has surfaced that restricts subscripts (8). Perhaps the answer would be an extension to REP that allows it to separate selected subscript calculations from the statements in which they occur.

REP requires nearly the full ASCII set for programming. No attempt was made to accommodate partial mappings. The characters can be represented externally in any form, but they had better all be there. All REP keywords and most Fortran keywords must be in lower case. All macro names must be spelled casewise exactly as declared. It would be possible to set up a macro library in which synonyms are provided, in upper case, for all lower case words, plus synonyms such as BEGIN and END for \{ and \}. It would be better to buy a new terminal.

REP has been partially ported from the Dec-20 to an IBM 4341 using Fortran H Extended (6) under VM-CMS (7). An EBCDIC tape was written on the 20, using a translation table derived from the REP macro set that defines EBCDIC equivalents to ASCII (ebcdic.rat), and read on the 4341. There were no problems. The bootstrap and the REP source both looked perfect when displayed on an ASCII terminal on the 4341, and the bootstrap compiled, ran, and regenerated itself from the source. The project was dropped at that point due to lack of interest, but there is every reason to believe that it could have been completed without unexpected problems.
RATFOR AND REP OBJECT CODE

The Addison-Wesley Ratfor translator generates Fortran code that is nearly unreadable. There are no spaces between tokens in a statement. There are lots of unlabeled continue statements. There are no comments and no blank lines. Statements that are longer than 72 characters are continued in column 6, quite normally and quite unbeautifully. There is nothing in column 73 or thereafter. Ratfor sometimes produces awkward code from control structures, full of extra goto's and continue's. The way Ratfor translates control structures is documented in (1), pp. 292-295.

However, the Fortran from a Ratfor translation is really just intermediate code, and not meant for human consumption, so who cares what it looks like? Even the awkwardness of the control structures is usually of no consequence, since at worst, a few microseconds are involved. Actually, many Fortrans do enough optimization of control statements that there is no effect whatever on the final object code. Perhaps the worst code is generated when an else follows a break, stop, return, or next. In this case, Ratfor generates an unreachable goto statement, which can elicit a diagnostic from some Fortrans. The only general answer is to avoid such constructs.

Many of the same shortcomings are also found in REP output, but

1. Enough white space is preserved to make the Fortran readable

2. REP generates no unlabeled continues. Whenever possible, REP avoids generating a continue statement.

3. In nearly no cases does REP generate unreachable code, so go ahead and use else after break, etc.

4. You can, if you wish, insert comments into the Fortran, by starting a regular ratfor comment with ## instead of #. such comments appear in Fortran on a separate line, preceding the ratfor line they're on, with the ## replaced by C.

5. In every output line, you will find a funny number such as 1-0002 in cols 72-77. This is derived from the line number of the ratfor source statement. The left digit is the include level, and the right 4 are the line number within the source file. This is mostly useful in debugging.

With these improvements, The Fortran intermediate code is almost civilized, though it hardly looks humanly written. It would certainly be feasible to write a portable program in REP, and distribute only the Fortran code to users on various machines.
STRING HANDLING IN RATFOR

There are several disciplines for handling string and character data in Ratfor:

1. The Fortran method.
2. The Software Tools method.
3. The PDP-10 method.
4. The hidden packing method.
5. The fixed field method.

The Fortran method is available simply because Ratfor includes all of Fortran. It is not recommended, because of its crudeness and machine dependency. It will not be discussed here. The methods found only in REP will be discussed in a separate section. That leaves only the Software Tools method to discuss in this section.

THE SOFTWARE TOOLS METHOD OF STRING HANDLING

This method has the advantages of elegance, understandability, and portability. Its disadvantage is primarily in efficiency of memory usage. There are two sources of inefficiency.

1. All strings are statically allocated. Memory is allocated at compile time, and stays allocated permanently.

2. On some machines, a full word must be allocated for each character. This is because many Fortrans have no smaller data type.

Both of these problems can be mitigated to a degree by careful programming. Therefore, this method is recommended for portable programs.

In essence, this method is the use of a single Fortran variable or array element, in the integer mode, to store a single character. For strings, we use an integral array. In those Fortrans that have a short integral type (such as BYTE or LOGICAL*1), that type may be used. Otherwise, a full integer variable is necessary. To cover these cases, while improving readability, we use an imaginary data type, type character, in declarations, e.g.

character function xyz(token)
character c, token(ARB)

We normally allow the Ratfor translator to decide what Fortran type to assign.

Please note that, although a character variable is an integer, it is best not to assume that it will hold anything bigger than a single character, i.e. 7 bits for ASCII or 8
bits for EBCDIC. Also, if you declare a formal parameter that is type character array, don't pass it an integer array. They may not always be the same size. It is not safe to make any assumptions about whether a character is signed or unsigned in its representation. It is reasonably safe to assume a character will hold a positive integer between 0 and 127.

Since we are dealing with integers, at the Fortran level, we can't use any Fortran methods of talking about characters. For instance, we can't use character constants ('aaa' or 3haaa), we can't use format statements, or formatted i/o, and we can't easily use data statements to initialize strings.

Instead of character constants, we use simple macros ("manifest constants") that are defined as having the numeric values of ascii (or whatever) characters. There is a standard list of these names, which will be appended to this document.

For string constants, we use the string declaration, e.g.

    string s "This is a string."

Once this declaration is made, the identifier 's' can be used wherever a software tools method string is legal (mainly as an actual parameter). What Ratfor actually does is to create a character array and initialize it with the value of the string followed by an end of string marker (EOS). The Addison-Wesley translator, as originally distributed, did not support the string declaration.

In connection with this method, there are also some special values that are used:

    EOS    -- A nonprintable character used to mark the logical end of a string
    NEWLINE -- A character that marks the end of a print line.
    EOF    -- An integer (n.b.) that marks the end of a file on input.

There are a few things to note here:

1. EOS is a value of type character, since it must be stored in a character array. Therefore, that value can't be used for actual data. Usually the ascii NUL (0) is used for EOS.

2. Whatever method your computer customarily uses to delimit lines of text, the Ratfor i/o system passes you a NEWLINE only when end of line is read, and it creates a line break when you write a NEWLINE. If your computer likes fixed length records, this may involve padding output with blanks, and stripping trailing blanks on input. Don't plan on receiving
trailing blanks. Also, random access to character files must be used with caution if portability is desired.

3. Since EOF is an integer, any function that may return EOF should be declared as type integer. This is not always observed, even in the book Software Tools.

For the most part, this is a convenient method of talking to a computer about strings. One thing that’s missing is formatted i/o, but that’s seldom really needed, and, in any case, quite inefficient. This method is very good in cases where field boundaries are not fixed, but are delimited by syntactical markers of some sort. It’s weak in cases where strings may be arbitrarily long (you’d rather use dynamic allocation), and in cases where field boundaries are delimited by a fixed count (you’d rather use a format). Both of these cases are addressed in REP, but in Ratfor you are more limited. There is nothing stopping you from designing a new set of primitives for handling strings, or any other kind of data if you choose to, however, but then you lose portability.

There are a few library routines available for string operations in this method. They are:

\[ i = \text{length}(s) \]

Length of string (NOT counting the EOS).

\[ i = \text{index}(s,c) \]

Index of \( c \) in \( s \). Returns 0 (zero) if string \( s \) does not contain character \( c \), else the index (i.e. subscript value) of the leftmost match.

\[ i = \text{equal}(s_1,s_2) \]

Compares \( s_1 \) with \( s_2 \), stopping at first EOS. Returns YES if \( s_1 == s_2 \), else NO. (YES and NO are the standard Software Tools way of talking about truth values.)

\[ \text{call scopy}(s_1,i_1,s_2,i_2) \]

Copies the string at \( s_1(i_1) \) to \( s_2(i_2) \). Movement is rightward and stops after moving the first EOS in \( s_1(i_1) \).

\[ n = \text{itoc}(\text{int},str,size) \]

Converts int to decimal string in array str with size limit size. If int is negative, a minus sign is placed at the left. An EOS is placed after the last
digit. The returned value is the length of the converted string, not counting the EOS.

\[ n = \text{ctoi}(s,i) \]

Converts the UNSIGNED digit string at \( s(i) \) from decimal to binary integer and returns it as the function value. Stops at the first nondigit encountered. Upon return, \( i \) points to the nondigit. Therefore, \( i \) must be a variable.

\[ \text{REP: } n = \text{sctoi}(s,i) \]

Same as \( \text{ctoi} \), except that \( s \) may have a leading sign.

\[ \text{REP: } i = \text{indexs}(s1,s2) \]

Same as \( \text{index} \), except that \( s2 \) is a string rather than a character.

\[ \text{REP: } \text{call scopyn}(s1,i1,s2,i2,n) \]

Same as \( \text{scopy} \), except that at most \( n \) characters are moved. Very possibly there will be no EOS.

\[ \text{REP: } i = \text{equaln}(s1,s2,n) \]

Same as \( \text{equal} \), except that at most \( n \) characters are compared.

\[ \text{REP: } i = \text{iscomp}(s1,s2) \]

Same as \( \text{equal} \), except that a high-low-equal comparison is done. Returns 0 if strings are equal, a negative number if \( s1 < s2 \), and a positive number if \( s1 > s2 \). Collating sequence is straight binary, and therefore not entirely portable across character sets.

These are not predeclared, therefore you must include an integer declaration for equal, \( \text{ctoi} \), \( \text{sctoi} \), and \( \text{equaln} \). The routines with the REP: prefix could be used with Ratfor, but are not standard primitives.

Certain common string operations are not included in this list, notably substringing and concatenation. These can be done quite simply using two or more of the above routines. For instance:

\[ \text{call scopy } (s1,1+1,s2,\text{length}(s2)+1) \] # concats \( s2 \) to \( s1 \)

\[ \text{call scopyn}(s1,5,s2,1,4) \] # \( s1(5..8) \rightarrow s2 \)

\[ s2(5) = \text{EOS} \]

If you find it easier to think in terms of substring and concatenate, you could write little routines of your own to
do them.

There are also standard routines for transferring characters and strings between memory and files. These are covered in the section on I/O.
INPUT/OUTPUT IN RATFOR

Since Ratfor includes all of Fortran, all Fortran I/O facilities are available. We will not discuss them here, except to say that you can use them if you want, but Ratfor provides and recommends another method, especially designed for convenient portable handling of character data. WARNING: although some implementations may allow it, you can't mix Ratfor and Fortran I/O calls on the same file at the same time.

This method treats each file as a stream of characters. Most files are divided into lines of text. Externally, the computer can represent this division in whatever way is most convenient for the operating system. Internally, the character NEWLINE is used. When reading a file, one character at a time, you get the data characters interspersed with NEWLINES, and you get an EOF at the end. You never get an EOS. When writing a file, you write data characters and NEWLINES. You don't write EOS's or EOF's.

When you open a file, you are given a number called the fd (or file descriptor). It's value is unimportant to you. It simply refers to the file while it is open.

In addition to any files you may open, there are 3 standard streams (4 in REP), with permanently assigned fd's. These are known by the names:

```
STDIN -- The standard input
STDOUT -- The standard output
ERROUT -- The error output
REP: CTLIN -- The control input
```

These names (actually manifest constants) may be used wherever an fd is legal. There are also a few simple i/o functions that use one of the standard streams implicitly.

By default, the standard streams are directed to the user's terminal. In a batch environment, they might default to the job deck or control file for input, and to the log file for output. As a minimum, there is a means for redirecting STDIN and STDOUT to a file (at run time). Some systems have more elaborate schemes, allowing ERROUT to be redirected also, and allowing STDOUT of one program to be piped into the STDIN of another. This is extremely useful, but even the simplest case mentioned above is very powerful, often allowing a single program to do the work of two or three.

LIBRARY ROUTINES FOR THE SOFTWARE TOOLS METHOD OF I/O

SINGLE CHARACTER I/O

```
integer getc, getch
```
\[ i = \text{getc}\ (c) \] Read 1 char from STDIN
\[ i = \text{getch}\ (c, \, \text{fd}) \] Read 1 char from any fd

\text{call putc}\ (c) \quad \text{Write 1 char to STDOUT}
\text{call putch}\ (c, \, \text{fd}) \quad \text{Write 1 char to any fd}

The functions getc and getch return an integer value, either the character read or EOF. The formal parameter c also receives the character read, and should be a character variable or array element. When getc or getch returns EOF, the value of c is undefined. If you continue to read after EOF, the results are undefined.

The "c" parameter to putc or putch should be a character variable or array element, or a symbolic constant naming a character.

**OPENING, CLOSING, ETC.**

\begin{verbatim}
integer open, copen, create
string fn "whatever"

fd = open (fn, mode) \quad \text{open a file}
fd = copen (fn, mode) \quad \text{open if possible, else abort}
fd = create (fn, mode) \quad \text{create a file}

\text{call close}\ (fd) \quad \text{close an open file}
\text{call remove}\ (fn) \quad \text{delete a file}

REP: call rename\ (fn1, fn2) \quad \text{rename a file}
\end{verbatim}

Open opens an existing file (for input or output).

Create creates a new file, erasing an existing one if necessary, and opens it.

Open and create return an fd if successful. If unsuccessful they return a symbolic constant ERR. Copen never return ERR, instead, it aborts with the message

\text{<filename> : can't open}

The modes for open, copen, and create are, minimally, READ and WRITE. Additional modes such as READWRITE, APPEND, and RANDOM are often provided.

As a rule it is not necessary to close files. The system does it automatically at the end of the run.

**STRING I/O**
integer getlin
line(MAXLINE)
n = getlin(line, fd)

Read a line of text from fd into character array line. The line will be terminated by NEWLINE EOS. The function value is the length of the line (exclusive of EOS), or EOF if there are no more lines to read. The maximum possible line length is set by the system constant MAXLINE. If the external representation is fixed length records, trailing blanks are removed. If the external representation is variable length, then a line may be longer than MAXLINE. In such a case, at most MAXLINE-1 actual characters are returned in each call to getlin, and only the last reading of the line will have NEWLINE. Each result will end with EOS. At EOF, the line contents are undefined.

call putlin(line,fd)

Write the string at line on fd. The name is putlin, but it might as well have been putstr. The name stems from the fact that putlin is the inverse of getlin in that a line read with getlin can be written with putlin.

REP: call setlin(n, fd)

Sets the maximum line length for future getlin and putlin calls on fd. Initially, the value is MAXLINE.

DECIMAL I/O

call putdec(n,w)

Convert n to decimal and write on STDOUT with a leading minus sign if negative. Use at least w columns, putting blanks on the left, if necessary.

REP: call putdf(n,w,fd)

Same as putdec, except that output is to any file.

REP: call putdfz(n,w,fd)

Same as putdf, except that the number is zero filled, rather than blank filled. If negative, the zeroes come between the sign and the first significant digit.

Note that putdf and putdfz provide all the primitives needed for writing floating or fixed point numbers as well as integers. To write a number with a decimal point, write the integral part with putdf, then the absolute value of the fractional part with putdfz.

Rep Summary, Page 22
No primitive is (yet) provided for reading decimal numbers. To accomplish this, read it as characters with getxxx, and convert it with ctoi or sctoi.

ERROR AND INFORMATORV MESSAGES

call remark ("String.")

Write string and NEWLINE on ERROUT

call error ("String.")

Write string and NEWLINE on ERROUT and abort.

REP: call remar ("String.")

Write string on ERROUT (no newline)

In all of these, the string must be a quoted string, not a declared string. Actually, it is a Fortran Hollerith constant. It MUST end with a period, which is not printed. The call to error is the standard way of voluntarily aborting a program. What happens to open files is not defined, but should be implemented in a favorable way.

REP: In REP, (A) the string may contain a real period, which must be coded as two adjacent periods (..). (B) The effect of a call to error on files is as follows: Files being read are unaffected. Files being created are aborted completely. If a file is being updated in place, all requested updates are made, and the file is closed normally.
This section includes such topics as entering and exiting a main program, receiving arguments in the main program, and controlling the standard I/O streams.

The prototype of Ratfor implementations is the Unix operating system. Many features of Ratfor are borrowed directly from it. So, let us look at some typical Unix command lines and see how they translate into portable Ratfor.

In Unix, a command line contains a program name, some optional arguments, and, optionally, some syntax for directing the standard I/O to a device, a file, or another program. In case the source or destination is another program, that program and its arguments and redirects are also part of the command line. In the following examples, assume that "cat" is a program that takes a file name as an argument and copies that file to its standard output, and that "raise" is a program that copies its standard input to standard output and changes any lower case to upper. A program that just reads standard input and writes standard output is called a filter. Now for the examples.

```
cat food
Since no redirect is present, the file "food" is listed on your terminal

cat food >dish
Standard output is redirected to a file called dish, so this copies food into dish.

raise <dish
Standard input is redirected, but not standard output, so the contents of dish are displayed in in upper case.

cat food | raise
The standard output of cat is fed to raise as standard input. Both programs run simultaneously. This is called a pipeline.
```

Now, how can we extend this functionality to Ratfor programs running on non-Unix systems? Any computer that has disk and terminal can redirect standard I/O, provided we get to write the getc/putc routines in any language that's suitable. Not all computers can run two or more programs simultaneously, so we may have to fudge the pipeline feature by using a temporary file. The operating system's command syntax may not be congenial to arguments and redirections, but there is nothing stopping us from reading another line, or reading the necessary information from a file.

Depending on the environment, there may be certain auxiliary statements or conventions needed to make a Ratfor program
feel at home. For Ratfor, this is entirely dependent on the operating system, and the whims of the implementor. Some Ratfor implementors, when working in a non-Unix environment, have chosen to have a main program in assembly language program act like the shell. All of their ratfor programs must be subroutines. Others have chosen to call the shell surrogate directly from a ratfor main program. In either case, a couple of statements have to be changed, slightly compromising portability.

REP: For REP, some conventions have been established, involving the use of standard macros to enforce portability.

1. Each main program should begin with a program statement:

```
program name
```

Some REP implementations will ignore it, others will use it wisely.

2. Each main program should have the macro

```
main_entry
```

as its first executable statement. Again, some installations will generate code from it, and some won’t.

3. To exit from the program normally, use

```
normal_exit
```

This has the effect of closing all ratfor files and getting on with any possible next program in a pipeline or sequence. This does not necessarily close fortran files.

4. To exit abnormally, use

```
call error ("message.")
or
stop
```

The stop statement is a macro, though is looks just like the fortran stop statement. Its action should be to close files that are open for update in place, abort any files being created, and abort any pending programs in a pipeline or sequence.

COMMAND LINE ARGUMENTS
Ratfor does not specify the external syntax of command line arguments, but suggests that they be separated by blanks, not commas or parentheses, and that there be an option of quoting them, in case they must contain white space, or begin with a redirection character.

Inside the program, the function getarg is used to pick up the command line arguments:

\[ i = \text{getarg}(n, \text{array}, \text{maxsize}) \]

where

- \( i \) will receive the actual length of the argument, or EOF if there are fewer than \( n \) arguments
- \( n \) specifies which argument to read, starting with 1 for the first.
- \( \text{array} \) tells where to read them. The book specifies that the array should be of type integer, but character seems more appropriate.
- \( \text{maxsize} \) tells how many positions are available in \( \text{array} \). The argument will be truncated if necessary at \( \text{maxsize}-1 \), to leave room for the EOS.
- You must declare \( \text{getarg} \) as type integer. It is not predeclared.
For those whose background is in Fortran, or Cobol, or pascal or in many other languages, the notion of a macro in a higher level language may be foreign. Many people still think of a macro as an assembly language statement that generates several instructions. To others, familiar with C or Lisp, macros may be second nature. What a macro actually is, in the simplest case, is a named string. The process of substituting strings (called macro bodies) for their names is called macro expansion. Macros must be defined before use. The process of associating names with bodies is called macro definition. A program that reads definitions and expands macros is called a macro generator.

In some cases, a macro body may have "holes" in it, into which other strings (called arguments) may be placed at macro expansion time. Macros without arguments are called simple macros. Although much of the power of macros is available only through argument macros, it's quite feasible to program using simple macros only.

The most fundamental use for simple macros is in the naming of constants. Such animals are variously referred to as "program parameters" or "manifest constants." Some examples:

```plaintext
define DIGO 48
define EPSILON .0055
define MAXLINE 82
define MSG "Too much sodium."
```

And their uses:

```plaintext
character c, line(MAXLINE)
data eps/EPSILON/
if (c == DIGO) call error(MSG)
100 format (MAXLINE a1)
```

The macro body may begin on the line after the define statement. If the macro body extends over several lines, it must be bracketed with {}'s. The outermost {}'s are stripped off. So, if you want the body bracketed, use an extra pair of {}'s. If you want a null body, code it as {}. Now for some slightly hairier examples:

```plaintext
define scan t = gettok(token)
  # get a token

define scannb
  repeat scan until (t = BLANK & t = TAB)
  # get a nonblank token

define nothing {}  
  # leave it out
```
define push
  {{sp = sp+1; stack(sp) = i}}
  # put it on a stack
  # note the double {{}}'s

define some_common
{
  common /some/ v1,v2,v3
  integer v1,v2,v3
}
  # a good way to handle common

Once you have defined a macro, you can redefine it as often as you like. Thus, macros can be used as translation-time variables. Macros are handled in a single pass, so no forward references to macros are possible.

In all examples, keep in mind that macro substitution is basically a mindless process. The macro generator doesn't know any Ratfor, even though it's built in to 'the REP translator. The body doesn't have to make sense in terms of Ratfor syntax until after all substitutions have been made.

Once a macro has been expanded, the macro generator isn't necessarily finished with it. It will, in fact, keep going over it until there are no more macros in it. This allows nested macros. For advanced use, it allows recursion and looping in macros. It also allows infinite loops. If a macro body contains its name, it may cause the macro generator to loop forever, so watch it. (One exception to this is that if the name of a simple macro is identical to the body, it won't loop. This is a way to cancel a previous macro definition.)

RULES FOR THE FORMATION AND RECOGNITION OF MACRO NAMES

1. Macro names may consist of letters, digits, and periods.

2. Macro names may be as long as you like.

3. Macro names are recognized as whole tokens only. Part of a word will not be expanded. Macro names inside quoted strings will not be expanded.

4. Upper and lower case are distinct in macro names, but not in fortran identifiers

5. The underline (or backarrow) can be used as punctuation in any alphanumeric token (identifiers, macro names, numbers) it is ignored, and is not part of the word.

6. By convention, certain macro names are uppercased, and variable names are always in lower case. It is difficult to give a rule for this, but in generally manifest constants should be in caps, and other macros are up to
RULES FOR THE FORMATION AND RECOGNITION OF MACRO BODIES
(Using the new format define statement)

1. The macro body may not contain unpaired \{\}'s or \[\]'s.

2. If the macro body begins with a \{, it ends with the matching \}. In this case, the outer \{\}'s are removed.

3. The outermost \[\]'s are removed in all cases.

4. The macro body can't begin with any of the following characters: _ = , \ / ? unless special measures are taken.

5. The macro body can't begin with white space or a newline unless special measures are taken. If there is no body after the name on the same line, the body is taken to begin on the next line.

6. The body ends at the first newline (or comment). For a multi-line body, use \{\}'s.

7. The underline (\_) can be used to continue any line of a macro body. This does not introduce a NEWLINE into the definition.

8. Certain nonessentials are stripped from the body at definition time. They are
   a. Unescaped leading and trailing whitespace on every line.
   b. All comments.
   c. Continuation markers (\_) plus the newlines they cover.

THE OLD FORMAT FOR DEFINES

There is another format for definitions:

    define(name, body)

It's included primarily for compatibility with other versions of Ratfor. The open format described above is preferred wherever possible, for readability. Some versions of Ratfor don't allow spaces between define and \(. The same versions typically don't like spaces after the comma, either. There's a bug in REP that prevents the body, in this format, from containing any commas (unless they're bracketed [\]).
The book (1, pp. 264-282) has a good deal to say about macros with arguments. Nearly all of it applies to REP, but with modifications. The macro environment of REP is much richer. Nevertheless, the information in the book is useful as background.

A programmer-defined macro may have 4 kinds of arguments in the following order:

1. Word arguments

These each consist of a single alphanumeric word, quoted string, or anything in []'s.

2. The () arguments

This is almost the same as the argument list in the book. The arguments are separated by commas and may contain balanced ()'s. Commas within ()'s in an argument don't count as argument terminators. If an argument has to contain a comma or unbalanced ()'s, you can bracket the argument with []'s, or escape the offending graphic with \ . One difference from the book is that REP can know how many arguments to expect, since a number or a list of names can be included in the define statement. Commas after the declared number of arguments have been parsed don't act as separators, but simply as part of the last argument.

3. The = argument

This argument, if defined may consist of exactly one =. The purpose of this argument is to tell the macro expander whether to look for further arguments, and to make it look to the programmer like an assignment statement.

4. The right argument

This argument, if declared, consists one string after the previous arguments. A right argument is not parsed if the macro expects but doesn't find the = argument. There are 4 syntactic variations on the right argument. The variations concern only how the macro generator finds the end of the right argument.

a. The right arg may consist of the rest of the statement, i.e. up to newline, semicolon, ), or until. This is good for assignment-like macros.

b. The right arg may consist of the rest of the
line, i.e. up to newline. This is good for macros that parse arbitrary strings.

c. The right arg may consist of the rest of the line, the next line, or a group of lines in {}'s. This is good for conditional assembly macros. The outer {}'s are removed.

d. The right arg may be an expression, that is, anything with balanced parenthesis, up to a comma, a super-balancing right parenthesis, or the end of the statement.

The ordering of the four argument types is invariable, and is as presented above. Each macro has its own syntax. Any or all of the above arguments can be omitted. The first two types can have a variable number of occurrences. All arguments can be given names, or referenced positionally. Even if an argument has a name, it can still be referenced positionally. It is important to note that the syntax of macro arguments is significant only in the act of finding them in the source. Once they have been located, the syntax is forgotten, and all arguments are treated alike.

ARGUMENT SUBSTITUTION AND THE SHRINKING DOLLAR

Once the arguments to a macro are found, and possibly evaluated, they are simply substituted into the text of the macro body, or in the case of built in macros, passed to a subroutine. There are a maximum of 9 arguments. The arguments may be referred to anywhere in the body by position, as $1 through $9. If the arguments have names, they may be referred to by name wherever a name can be recognized, i.e. not within a word or a quoted string.

If you want a real $ to be in the body, you must code it as $$$. This shrinks when the macro is expanded to a single $.

Normally, the arguments are simply numbered from left to right, but a special glitch occurs when there are an unspecified number of () args and also an = or right argument. In this case, the = arg is $8 and the right arg is $9. Otherwise, the first argument is $1, and the second $2, etc. This is not a bug, but intended behavior, inherited from older implementations and retained for compatibility. It is recommended that you always specify how many () args a macro can have. It's better for documentation, you don't run into the numbering glitch, and you don't get burned when the last argument contains a comma.

All missing arguments are substituted as null strings.
Hypothetical example using macros to implement data structuring:

Suppose \texttt{xrecord} is a macro that declares a new variable of some complex form, i.e. a data structure. Then a macro call such as

\begin{verbatim}
xrecord.sss(3) = <something>
\end{verbatim}

could generate declaration and initialization for an array of 3 \texttt{xrecords} initialized to \texttt{<something>}, and also generate macros that cause what look like references to \texttt{sss} or its members to expand into the appropriate function or subroutine calls (or macro calls). Subsequently, references to members of \texttt{sss}, could be made in assignment statements on either side of the =, in function calls, and in conditionals. References to the whole structure could be made in parameter lists, and in statements of the form \( x = y \). These following examples illustrate the kinds of transformations that can be made:

\begin{verbatim}
sss(zzz) = 1       ->  call setxr(xsss,zzz,1)
vbl = sss(zzz)    ->  vbl = getxr(xsss,zzz)
call subr(sss)   ->  call subr(xsss)
if (sss(zzz) == 1) ->  if (getxr(xsss,zzz) == 1)
sss = ttt          ->  call movxr(ttt,sss)
\end{verbatim}

The possible variations are endless.

THE ARGUMENT TEMPLATE - CRUDE BUT PAINFUL

In order to define a macro with arguments, you must code a template after the name to tell the format of the argument list.

If there are only parenthesized arguments, () is all you need, but /()/ is also legal. For anything more ambitious, the slashes are mandatory.

If you want the arguments list to be active, i.e. to be evaluated before substitution, put a + after the first slash or paren in the template.

If there are word arguments, the number of them goes after the left slash. If you want to give names to them, put the names separated by whitespace instead of a number after the slash. A word argument may not have any of the following names (unquoted): "rest", "stmt", "line", or "expr".

If there are paren args, () goes next. If you want to specify the number of paren args, put a digit in the parens. If you want to give them names, put the names, separated by commas or spaces in the parens. Don't try to use both names and a number.

Rep Summary, Page 32
If there is an = arg, = goes next. If you want to give it a name, put the name after a colon after the =.

If there is a right arg, the type of the right arg goes next, just before the right slash. The types are (unquoted) "rest", "stmt", "line", and "expr". If you want to give it a name, put a colon followed by the name after the type. By default, the right arg is given the same name as the type, so you don't have to code "rest:rest/". If you want an unnamed right argument, use "rest:/"

Some examples of templates:

```plaintext
define aaa()  
    # up to 9 paren args

define aaa ()
    # same

define aaa /()/
    # same

define aaa (3)
    # up to 3 paren args

define aaa (+3)
    # up to 3 paren args, active

define aaa (xx, yy, zz)
    # up to 3 paren args with names

define aaa /xx yy zz/
    # up to 3 word args with names

define aaa /+xx yy zz/
    # up to 3 word args with names, active

define nt /name(subs)line:val/
    # a type declaration, with initializer,  
    # like this:
    #
    #    nt var(1,33,55) "string",44, 55, .....  

define v /(s1,s2)=rest/
    # a pseudo-array, like this  
    #
    #    v(2,3) = 5
    
    #    j = v(aaa,bbb)
    #    v(aaa,5) = v(aaa,6)

define .if /(1) stmt/
    # conditional assembly, like this:
    #
    #    .if (a=3) {
    #        include this
    #        and this
    #    }
```
ACTIVE AND PASSIVE BODIES

Sometimes you want the body of a macro to be evaluated before the definition is made. This is especially important when macros are used for translation time computation. To do this, put an equal sign before the body, between after the template, or, if there is no template, after the name.

```
define xx = arith(y+1)
    # Make xx be what y was, plus 1.

define set/var = rest/
    define var = arith(rest)
    # Define a "set" macro to do just that.

set xx = y+1
    # And now you can write it like this.
```

BLOCK STRUCTURED MACRO TABLES

Where the Ratfor has a single unstructured symbol table for macros, REP has a stack of symbol tables. A binary search lookup algorithm was chosen over the faster hash search for simplicity and because it takes less memory space for the table. The hash search would require a separate hash table for each local frame. Even the present binary search is ENORMOUSLY faster than the linear. The binary search also retains its speed for huge tables, whereas the hash algorithm degrades to linear beyond a certain size.

Programmers may ignore the local frame facility if they choose. To use a local frame, the programmer needs to know the builtin macros "beginframe" and "endframe". They take no arguments and can be thought of as statements. Any macros defined after a beginframe are forgotten at the matching endframe. Any macros redefined within the frame revert to their previous definitions at endframe. Within a local frame, all definitions made in encompassing frames are still in effect. It is in fact impossible to change them in any way. It is only possible to hide them with local definitions of their names.

There is a trick, however, for passing data from a local frame to it's parent. It involves the use of the argument mechanism to bridge the gap. For example:

```
define retval(arg) endframe define fun arg
    retval ("Hello up there!")
```

This would cause the local frame to end and the definition of fun as "Hello up there!" to take place in the parent frame.
It is possible to force each subprogram to have its own local frame. This is desirable when macros are used to extend the range of data types, for otherwise macros defining local variables would have scope greater than the variables themselves. You could accomplish this by simply putting beginframe and endframe around the subprogram, but it is better to make it automatic. Here are the definitions that do it:

```plaintext
define subroutine beginframe 0 %subroutine%
define function /rest/
  beginframe ifelse($0,,0) %function% rest
define end endframe END
```

That's a mouthful. If you can understand that, you know REP pretty well. So, in order to save a lot of questions, I'll explain it in detail now. The %'s around "subroutine" and "function" serve two purposes. Firstly, they prevent self reference, which, in this case, would loop forever (RIP). More importantly, they prevent the automatic name shortening feature from butchering the keywords; the percent's will be removed automatically. The "O" before subroutine makes the statement go into the rising class, meaning that without the "O", the subroutine statement would migrate down among the executable statements. The situation is basically the same for "function", but with an added twist. "function" isn't always the first word in a statement. It might be preceded by a type name. The argument $0 can be used to determine whether a macro is called from the first word of a statement. That's what the ifelse does, generating the 0 label appropriately. But in order to test $0, the macro must have arguments. A simple macro wouldn't do, so we use a right arg to pick up and deliver the function name and formal parameter list. The case of the "end" macro is much simpler. It is short, so shortening isn't a problem, and it belongs at the bottom of the subprogram, so we don't want it to rise. (Are we talking about baking bread?) Therefore, it is sufficient to put it in caps, in order to prevent self reference.

**SAVING COMPILED MACRO LIBRARIES**

Compiled macro tables may also be stored on files, with the "saveframe" built in macro. This simply causes the current frame to be written out. A saved table can be reused with the builtin macro "use filename". It's generally faster to do it this way than to use an include statement to read the source of the definitions, although the results are the same.

When you bring in a macro table with "use", it retains its status as a separate frame, and is opened for new definitions (in core, not on the file). The program remembers how many such tables are in use, and endframe always cuts back to the state before the previous beginframe -- you don't need an endframe to match each "use filename".

The set of frames consisting of those started by a beginframe
and those created by each subsequent use up to the next beginframe or the top of the stack, is called a region of the symbol table. If you do a saveframe after a use, only the frame created by the use is written out. You could use the following sequence to update a macro library:

```
beginframe
  use libname
  define
  ...
  define
  saveframe libname
endframe
```

EVERY MACRO IS A POTENTIAL MINISTACK

You can treat any macro name as a stack. The key to doing this is the define.save built in macro. It works just like define, except that the previous definition of the name, if there is one in the current frame, is pushed onto a stack of inactive definitions. You may think of the current definition as the top of the stack. When you undefine the name (with an undefine built in macro) the most recently saved inactive definition of the name is reactivated. The stack depth is limited only by the amount of memory available for macro definitions. Each ministack is local to one frame. Therefore, a local frame can’t affect a ministack in a parent frame, and a ministack in a local frame disappears along with its frame at endframe time.
The macro features in rep are sophisticated enough for almost any purpose. Many of the advanced features were introduced for the needs of software engineering and especially for the purpose of data structuring and data type creation. The goal in all cases was to provide facilities for extending the underlying (ratfor) language in the spirit of the language without losing portability. This goal has largely been reached, and the system is powerful. There are, however, a few design glitches that must be well understood before the more advanced features can be adequately exploited.

We have seen simple macros, and macros with arguments. This section will discuss the full syntax of macros with arguments, and the various issues of controlling the order of evaluation.

**THE DEFINE STATEMENT (BNF In Toto)**

```
define_stmt ::= define <macname> [=] <body>
define_stmt ::= define <macname> <template> [=] <body>
template ::= (actpas xarg)
template ::= /actpas larg parg eqarg rarg/
actpas ::= + ; - ; empty
xarg ::= empty ; nargs ; argnames
nargs ::= 0 ; 1 ; ... ; 9
argnames ::= argname ; argnames argname
argnames ::= argnames , argname
argname ::= any alpha token except an rtype
rtype ::= rest ; stmt ; line ; body
larg ::= empty ; xarg
parg ::= empty ; (xarg)
eqarg ::= empty ; = ; = ; argname
rarg ::= empty ; rtype ; rtype ; argname
```

**LEXICAL UNDERPINNINGS — SOME TOKENS OF MY ESTEEM**

In Ratfor, a token is one of the following: a quoted string, a sequence of letters and digits, one of the alternative operators == >= <= "~, a single printable nonalphanumeric character, or a newline character. Spaces and tabs don't count at all. They are needed only to delimit alphanumeric tokens. In REP there are a few differences in the way tokens are scanned:

1. There is one additional token, the blank. Any sequence of blanks and tabs is reduced to a single blank. Blanks and tabs are totally ignored at the
beginning and end of a line.

2. % and . are treated as alphabetic.

3. _ is the continuation character. Whenever a _ is read, the scanner looks forward as far as the first nonblank on the next line, if necessary. If nothing is seen other than "air": (blanks, a newline, possibly a comment), the air is discarded, so it looks to ratfor like one line without any continuation. If there's anything on the same line other than air, scanning resumes there. In this way, the continuation character can also be used for readability in names, numbers, and even strings. If you need a real _ in your program, anywhere, you should code _

ACTIVE AND PASSIVE ARGUMENTS

An argument to a macro may be active, in which case it is evaluated before it is substituted, or passive, in which case it is substituted without evaluation. A passive argument is usually substituted verbatim but not always, however, since the ? escape operator still operates. A programmer defined macro can have passive or active arguments, but not both. Some built in macros have active and some have passive arguments. A few have both. It's no tragedy not to be able to mix active and passive in programmer-defined macros, since you can evaluate an argument in the macro body before use (with a define statement). And that's almost never needed.

The following are the escape characters or "magic markers" in macroland:

[] - Text within square brackets is never evaluated. In macro of arguments, the argument can't end until the []'s balance, so this means can be used to include the normal end delimiter in the argument. The outermost []'s are removed, in all cases. One use is to protect a macro name from evaluation. But watch out. If you want to have a protected macro name inside a macro body, it must be within (at least) 2 levels of []'s. This is because the outermost brackets are stripped off when the macro is defined, and the next level goes when it's expanded.

? - Causes the next token, if it's alphanumeric, to be evaluated even in []'s or in a passive argument. For now, please limit the use of this to simple macros. A bug prevents it from working right on argument macros or builtins. If the next token is not alphanumeric, ? is the same as \.
\ - Causes the next token, if it's alphanumeric, to remain unevaluated. exactly as if it were in []'s. if the next character is not alphanumeric, it becomes the next token, without its usual meaning. (Remember that .% are alphabetic.) There are numerous bugs in the operation of this feature, and roughly the same bugs occur when ? is used before a nonalphanumeric character.

The escape characters ? and \ are removed when they operate, analogously with []'s. Therefore, if you want a ? or \ in a macro body, you need to code ??? or ``` in the define statement. Escape characters are effective in the mainline text, as well as in macros. Escape characters are not effective in quoted strings.

Since the macro body in a define statement is in fact a macro argument, no matter what it may look like, the normal macro escapes (\()?) also apply. It is therefore possible to bracket the body with []'s as well as {}'s, if it extends over one line.

Sometimes it is necessary to evaluate a macro body before the defining takes place. This is done by putting an equal sign (=) before the body, after the name or the template.

ORIGINS OF THE REP MACRO GENERATOR

The REP macro generator is loosely based on the "macro" program in the book (1, chapter 9). As a preprocessor to Ratfor, "macro" had several faults:

1. When you got a diagnostic from Ratfor or FORTRAN, it was really hard to track it down, because your source code was 1 or 2 files upstream.

2. Expansions tended to happen in awkward places, like within comments. This was because macro didn't know the lexical structure of ratfor.

3. Macro definitions were unreadable and error prone -- chock full of brackets and parentheses, without spacing or line breaks.

4. All macros had the same syntax, which resembled fortran function calls or subscripted variables. There was no way to write a macro that resembled an assignment or a declarative statement.

5. The argument list, if present, had to follow the macro name without any space in between, or it would not be recognized. This was not a bug, but necessary to distinguish between
macro(args) -- a macro with arguments
and
macro (args) -- a redefined function
or array name.

The first two problems were corrected by incorporating the macro generator into the ratfor translator proper. No more would you hunt for the error. REP always tries to tell you where it happened. Problems 3, 4, and 5 were ironed out by changing the syntax of the macro generator. The current setup has two types of programmer-defined macros, with and without arguments, and numerous builtin macros. The argumentless macros are identical to the macros of Ratfor. Argument macros are defined and used in a new way. Among argument macros, there is a healthy variety of syntax options that the programmer can declare along with the macro body. The builtins each have syntax suitable to their functions.

The resulting macro system is complex, even baroque, but that complexity pays off in readable programs. The macro language of REP has sufficient recognizing power to define very natural seeming extensions to the underlying Ratfor language, and sufficient macro-time computing power to make the extensions meaningful and easy to use, once the defining macros have been written.
BUILT-IN MACROS IN REP

In the following list, arguments are given names reflecting their functions, and the name is preceded by a + to indicate active, or a - for passive. The + or - should NOT be coded as part of the argument. In those macros that have a filename argument, the filename may be quoted or not. If not quoted, it must be alphanumeric.

.err (-message)

Give an error message looking just like a message from ratfor.

.index (s1, s2) check

Search s1 left to right for a substring equal to s2. Generate 0 if not found. Generate position of leftmost matching char in s1 if found. Positions are numbered from 1 on the left to n on the right.

.ini

Clear the macro tables and reinstall the builtins.

.mode -modename [ ... -modename ]

Set internal operating modes for the REP translator. Legal mode names are:

reorder
noreorder

Reorder or don't reorder statements. When reordering is in effect, all declarative statements with keywords (except DATA) bubble to the top of a subprogram, and all executable statements, data statements, end statements, and statement function declarations sink to the bottom. Statements are not reordered within the rising and sinking classes. This is not exactly perfect for statement functions (Who uses them?), but works well for everything else. You can override the sinking designation of statement functions (or anything else) by putting a label of 0 on the statement. The label is removed, but the statement rises. You can then put all your statement functions at the end of the subprogram (or in the next county.) WARNING: The string declaration will not work correctly if reordering is off. The main use for turning reordering off is in interactive debugging of macros. CAUTION: Don't try to toggle reordering within a
program. The results may be disappointing. It was not intended to be used that way.

shorten
noshorten

Shorten or don’t shorten identifiers. When shortening is in effect, every Fortran identifier longer than 6 characters is replaced by a unique 6 character identifier starting with the same letter as the original. Fortran keywords are exempt, but subprogram names are eligible for shortening. The effect is the same as if an identifier declaration (see below) were used before the first occurrence of the identifier. If a name that has been shortened should become undefined, by an undefine, endframe, or .ini macro, it cannot be referenced below that point in the source. If used, a different identifier will be assigned to it. Names shortened in one translation have no relationship to names shortened in another, so keep subprogram and commonblock names short unless you intend them to be local to one file. If, for any reason, you need to exempt a long name from being shortened, you can do so by putting a % before and after it. You can even define it as a macro, e.g.

define flamingo O %flamingo%

The O makes it rise, This is in fact how all fortran keywords are handled, except for those that need to rise and aren’t necessarily the first word in a statement, like "function."

number
nonumber

Put or don’t put the include level and line number in cols. 73-80 of every Fortran statement generated. Useful if your Fortran compiler looks at those columns, and when interactively debugging macros at a slow terminal.

stringlength
nostringlength

Remember or don’t remember the lengths of declared strings.

.msg (-tag, -message)
Give an arbitrary message at the terminal, similar to the message from at" are replaced by the tag argument, and the message argument appears on the next line. If the tag is null, the first line is not displayed. If the message is null, the second is not displayed.

.nolf (+stuff)

Evaluate argument and convert all NEWLINES in it to blanks. Generate the result.

.nosp (+stuff)

Evaluate the argument and squeeze out all spaces, tabs, and newlines. Generate the result.

.skipto -"string"

Skip forward in source file to a comment that begins with string. Blanks and tabs between and string are ignored.

@ (+expression)
arith (+expression)

Evaluate infix expression using integer arithmetic and masking. Operators in order of priority (lowest to highest) are

& (and), | (or)
\^ (not)
> (greater than), < (less than), = (equal)
+ (add), - (subtract)
* (multiply), / (divide) % (remainder)
- (unary minus)

Grouping can be controlled by parentheses, as in Ratfor. Since the argument is active, macros in it are evaluated before the arithmetic is done. Therefore simple macros serve as variables, argument macros as functions. Truth values for & | ^ are 0 (false) and 1 (true). Arithmetic is done in fortran integer arithmetic. TOTALLY unlike the arith described in the book.

beginframe

Start a new frame and region in the macro table

deblank (+stuff)

remove any leading or trailing blanks or tabs from the (evaluated) argument, and generate the result.
define (-name,-body)
define -name -template -body
define -name -template = +body
define.save (-name,-body)
define.save -name -template -body
define.save -name -template = +body

define a macro, described above. The difference between define and define.save is that define replaces any previous definition of the same name in the current frame with the new one, forgetting the old one, while define.save pushes the previous definition (in the same frame) onto a stack of definitions of that name in this frame. The difference is unimportant until you undefine the name. Then, if you originally define.save'd the name you're undefining now, the previous value will reappear like magic. Macro table space freed by define or undefine is reclaimed, but naturally define.save frees no space.

deparen (+stuff)

First deblank the (evaluated) argument, then, if it is parenthesized, remove one level of parens and generate it.

endframe

End frame and region of macro table. Undefines all definitions made since the last beginframe.

identifier -name

Create a unique identifier starting with the same letter as name and define name as identifier. Mostly useful in macro bodies. If not the first word in a statement, also generate the created identifier. Thus

a = identifier b

is equivalent to

identifier b
a = b

The definition is made with the equivalent of a define.save, so you can revert to a previous definition with an undefine.

ifelse (+str1, +str2, -truestuff, -falsestuff)
ifelse (+str1, +str2, -truestuff)
Compare str1 with str2, and if equal, generate truestuff, else falsestuff.

include +filename

Include the named file in the source stream at this point. Include is line oriented, i.e. the included file must consist of one or more complete lines.

incr (+integer)

Generate integer + 1. Integer can be a variable.

label -name

generate a unique label number, in the 26000 range, and define the name as the number. If not the first word in a statement, also generate the created label number. Thus

go to label blazes

is equivalent to

label blazes
go to blazes

The definition is made with the equivalent of a define.save, so you can revert to a previous definition with an undefine.

len (+string)

length of string

macpeek -macroname

Display the body of a macro unprocessed on the terminal, intended for interactive use in debugging macros. The body may look peculiar, since it is copied right out of the macro table. It will lack all comments and indentation, and will be preceded by 1 to 6 characters of internal codes.

quote (+stuff)

Put quotes around something

saveframe +filename

Save the top frame of the macro table in the named file

substr (+string,+first)
substr (+string,+first,+length)

Find the substring of string from the first’th character, of length length, or if length argument not given, the rest of the string to the right. If the length is zero or negative, return a null string. If the length is greater than the number of characters left in the string, return the remaining characters. Strings index from 1 on the left rightward and upward.

Either or both of the first and length arguments may be a nonnumeric string. These are converted, as it were, to numbers by matching (as in the .index macro). If first is a string, and there is a match, the substring begins with the first character AFTER the match. If first is a string and there is no match for it, return a null string. If length is a string and it matches, the substring ends with the first character BEFORE the match. If the length is a string and there is no match for it, the substring continues to the end of the string.

undefine -macroname

Remove the most recent definition of macroname in the current frame. No action if not defined in the current frame, even if defined in an .outer frame. If a previous definition of macroname in the current frame was made with a define.save, identifier, or label macro, revert to the latest such definition.

unquote (+stuff)

Remove the first and last characters of the evaluated argument, whatever they are, and generate the result

use +filename

Load the named file into a new frame of the current region of the macro table, leaving that frame open for new definitions. The file must have been written by a saveframe macro.
This set of macros contains everything you need to write application programs using Ratfor I/O and string handling in REP. It is automatically loaded at the beginning of every REP translation. The only way to get rid of it is with an \ini built in macro. Also included are macros that implement the select-case control structure, structured conditional assembly, and recursive procedures. Actually, there is more in the set than is documented here, but all of the undocumented macros are primarily for systems programming.

**STRUCTURED CONDITIONAL ASSEMBLY MACROS:** These provide a conditional assembly facility that roughly approximates the if-else structure in Ratfor proper. The macros are:

```
.if (condition) stmt
.elseif (condition) stmt
.else stmt
```

Here, the "condition" is any expression that the built in macro arith will accept, and the "stmt" is a right an argument of the statement variety, i.e. the rest of the line, up to a semicolon or an until clause. If the rest of the line is empty, it begins on the next line. If it must contain several statements, it can be bracketed with {}'s.

These don't group like the if-else of ratfor proper. Specifically, an else groups with the earliest if, not the latest. It is suggested, however, that good programming practice would avoid the problem, since conditional assembly should be kept simple. In any case, the default grouping can be overridden by bracketing with []'s, and by the use of .elseif - which does what "else if" does in ratfor proper.

The truth values are 0 and 1. The size of the conditional text is limited by the amount of storage available for macro arguments, so if it has to be over, say, 10-20 lines, use the .skipto builtin macro.

**CHARACTER NAMES FOR USE IN STRING PROCESSING:** The standard macro library contains suitable definitions of all the character names in the standard Ascii set. Here are the standard names, in Ascii order. These names should all be defined. If your character set lacks equivalents for some of these, define them as something ridiculous, to cause a diagnostic if they should be used. If your set contains more characters, and you intend to use them (tsk tsk), add them to the list.

```
NUL  SOH  STX  ETX  EOT  ENQ  ACK  BEL
BS   HT   LF   VT   FF   CR   SO   SI
DLE  DC1  DC2  DC3  DC4  NAK  SYN  ETB
CAN  EM   SUB  ESC  FS   GS   RS   US
```
In addition to the above, the following auxiliary macros are defined. Together with the ascii names, they constitute all REP has to know about a character set. They are:

- **NEWLINE** - what code to use for line breaks
- **EOS** - what code to use for EOS
- **CHARBITS** - how many bits in a character
- **CHARCODE** - name of the code (e.g. Ascii)

The standard library also contains some synonyms for these names. It is not necessary to redefine the synonyms. Their values are the names they are equivalent to, not the values. In fact, you don't even have to know which are main names and which are synonyms, unless you want to change the character set. The synonyms are:

- **AND**
- **AMPER**
- **EXCLAM**
- **BANG**
- **BLANK**
- **SPACE**
- **CARET**
- **UPARROW**
- **MINUS**
- **HYPHEN**
- **SHARP**
- **POUND**
- **UNDERLINE**
- **BACKARROW**
- **ASTERISK**
- **STAR**
- **TILDE**
- **TWIDDLE**
- **NULL**
- **NUL**
- **ctl_A**
- **SOH**
- **ctl_B**
- **STX**
- **ctl_C**
- **ETX**
- **ctl_D**
- **EOT**
- **ctl_E**
- **ENO**
ctl_F ACK
ctl_G BEL
ctl_H BS
ctl_I HT,TAB
ctl_J LF
ctl_K VT
ctl_L FF
ctl_M CR
ctl_N SO
ctl_O SI
ctl_P DLE
ctl_Q DC1
ctl_R DC2
ctl_S DC3
ctl_T DC4
ctl_U NAK
ctl_V SYN
ctl_W ETB
ctl_X CAN
ctl_Y EM
ctl_Z SUB

Please note that NEWLINE is not necessarily synonymous with LF or ctl_J, and EOS is not necessarily synonymous with NUL or NULL. NEWLINE and EOS are main names, defined separately for each character set.

**GENERAL I/O AND SYSTEM-RELATED MACROS**

System identification:

<table>
<thead>
<tr>
<th>SYSTEM</th>
<th># One of the machine types known to REP currently DEC10, DEC20, or BOOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERSION</td>
<td># The change level of REP. If it’s not 300 (for 3.00), this document doesn’t apply.</td>
</tr>
<tr>
<td>STDOUTTABLE</td>
<td># the file name this macro library resides on</td>
</tr>
</tbody>
</table>

Standard fd's:

<table>
<thead>
<tr>
<th>CTLIN</th>
<th># Control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROUT</td>
<td># Error output</td>
</tr>
<tr>
<td>STDIN</td>
<td># Standard input</td>
</tr>
<tr>
<td>STDOUT</td>
<td># Standard output</td>
</tr>
<tr>
<td>TERMINAL</td>
<td># User’s terminal</td>
</tr>
</tbody>
</table>

Standard dimensions:

<table>
<thead>
<tr>
<th>MAXCARD</th>
<th># Maximum line size for getlin</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXLINE</td>
<td># must be 2 more than MAXCARD</td>
</tr>
<tr>
<td>MAXNAME</td>
<td># Max chars in a file name</td>
</tr>
<tr>
<td>ARB</td>
<td># Arbitrary dimension for actual</td>
</tr>
</tbody>
</table>
parameters

Modes for opening files:

READ
READONLY
WRITE
WRITEONLY
APPEND

miscellaneous constants

EOF # End of file code returned
YES, NO # Logical values
OK, ERR # General error/nonerror codes
WORDBITS # integer wordsize in bits
WORDCHARS # Packed Characters per word

Type names

canacter # Type character
char # Abbreviation
yesno # Takes values YES and NO

Portability hooks

program /name/ # beginning of main program
main_entry # Entry point of main program
normal_exit # Normal exit of main program
stop # abort program
abs # Absolute value function
max # Maximum function
min # Minimum function

Minor league syntax extensions

andif # Keyword synonym for "if"
orif # Keyword synonym for "else if"
otherwise # Keyword synonym for "else"
select # the select-case structure
case

Nonstandard stuff for 10's and 20's

all_integers # implicitit integer(a-z)
DEFAULT # Set file name defaults
default # Lower case preferred
DEVICE # Defaultable field of file name
DIRECTORY # Ditto
NAME # Ditto
EXTENSION # Ditto
GETBYTE # Character fetch
getbyte # Lower case preferred
SETBYTE # Character store
setbyte # Lower case preferred
REDIRECT # Redirect standard i/o
redirect  # Lower case preferred
SPAREUNIT  # Unit number for remove, rename

Odds and ends

COMPARE(s1,s2)  # Makes string comparison into
EQUAL  # Statement. Useful for some
HIGH  # algorithms. use:
LOW  # COMPARE(s1, s2)
UNEQUAL  # if (EQUAL)......
         # or
         # if (COMPARE(s1,s2),EQUAL)....

ISDIGIT(c)  # test value for being a digit
# becomes (DIG0 <= c & c <= DIG9)
ISASCII(c)  # test a value for being ascii
# becomes (0 <= c & c <=127)

SIMPLE RECURSION MACROS: This set of macros allows the coding
of mutually recursive procedures. There is no provision for
_call-parameters or local variables, but in many cases these
are not needed. The clarity achieved by this simple form of
recursion is more than adequate for parsers and macro
generators. Each set of mutually recursive procedures must be
contained within a single subprogram, which is a weakness.
Procedure names are macro names.

The user macros are

begin_recursion(n) - declare recursion in subprogram
  (n = stack size)
rpush(int)  - push integer on return stack
rpop(int)  - pop integer from return stack
procdef(name)  - predeclare a recursive procedure
proc(name)  - declare a procedure entry point
procend  - Return from procedure
name  - call named procedure

Procdef and proc are both required for any procedure. Procdef
must precede the first call, and proc is the entry point.
Procedures behave much like subroutines at machine level,
where a stack is used, as in the pdp-11, pdp-10, 8080, 6800,
etc. Implementation is via computed goto's, and should be
100% 1966 standard.

The following names are reserved to the recursion macros:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ijret</td>
<td>integer</td>
<td>Index for returns (computed goto)</td>
</tr>
<tr>
<td>plabel</td>
<td>macro</td>
<td>Used in generating entry labels</td>
</tr>
<tr>
<td>retlab</td>
<td>macro</td>
<td>Used in generating return labels</td>
</tr>
<tr>
<td>firstret</td>
<td>macro</td>
<td>Used in generating return labels</td>
</tr>
<tr>
<td>retlist</td>
<td>macro</td>
<td>Used in generating return labels</td>
</tr>
<tr>
<td>1000+++</td>
<td>label</td>
<td>Entry point labels</td>
</tr>
<tr>
<td>2000+++</td>
<td>label</td>
<td>Return point labels</td>
</tr>
</tbody>
</table>
rcall macro. Do a recursible call
rstk array The return stack
rp integer Return stack pointer
trace integer Call-return trace switch (YES/NO)
name.L macro Entry label for each proc
name macro Call to each proc
endreursion macro Invoked by end
end macro Use like regular end statement

For a good example of recursive procedures in REP, get a copy of the source code of REP itself, and examine the subroutine called "dorth". It contains a recursive descent parser for arithmetic/logical/comparison expressions written using these macros.

LIST PROCESSING MACROS: These macros do a kind of list processing at macro time, reminiscent of some operations in LISP. The similarity is superficial. These lists are represented as strings, with the elements separated by commas. Any element of the list may be a parenthesized sublist. The list itself is not parenthesized.

In all of the list macros, the list is an active parameter. This means that simple macros can be used as list variables, just as they can be used as numeric variables for arithmetic.

.car (list)
Get the first element of the list. Remove any whitespace that may be around it, and, if it's a sublist, remove the parentheses, and remove whitespace again.

.cdr (list)
Get the part of the list that .car would have left behind.

.isnil (list)
Test a list for emptiness. A list is empty if it contains nothing or just whitespace.

.nilcar (list)
Test the car part of a list for emptiness.

.nilcdr (list)
Test the cdr part of a list for emptiness.

EXTENDED CONDITIONAL ASSEMBLY MACROS

Rep Summary, Page 52
.while (condition) stmt

Analogous to the while loop of Ratfor proper. The statement is generated repeatedly as long as the condition is true. The condition and the statement are the same as those in the .if macro.

.for (init, condit, reinit) stmt

Analogous to the for statement of Ratfor proper. The init and reinit look something like assignments. Actually, they are define statements without the word "define". They should usually be of the form

\[
\text{name} = \text{arith(expression)}
\]

The conditional and the statement are as in .while or .if.

.foreach temp in (list) stmt

Generate the statement once for each element in the list. On each iteration, a macro name, referred to as "temp" above, is defined as the current list element, so any mention of the temporary name in the statement is replaced by the current list element. The temporary name must not appear in the list, or the macro generator will have a nervous breakdown. The argument referred to as "in" above should be exactly the word "in".

\[
\text{example:}
\]

.foreach a in (q,w,e,r,t,y)
{
  call read (a)
  call print (a)
}

Although this macro has a passive parameter list, the "list" argument is evaluated in the body before use. This means that the list can be a macro name.
USING REP ON THE PDP-10

To start REP, use

.r rep

or the equivalent for your system. It would vary depending on whether REP is located on sys: or on some private area, and on what name you are calling it.

For Dr. Glaser's PDP-10, on the University of California's Berkeley Campus, you would use one of

 .run repv1 ; for REP v1.99
 .run rep202 ; for REP v2.02
 .run rep300 ; for REP v3.00

after establishing a path to the Software Tools directory by

.path swt:=[3302,301,swt]/search

In any case, REP responds with an asterisk prompt, and you reply with a program name or a unix-like redirection string.

If you just want to translate a program with extension .rat, use

.r rep
 *programe

This will translate the program dsk:programe.rat to Fortran and place the output on dsk:programe.for. If you wish, you may specify a device other than dsk:, or an extension other than .rat. As elsewhere on TOPS-10, program names are limited to 6 characters, and extensions and device names to 3.

An alternative way to call REP is by redirecting STDIN and STDOUT.

.r rep
 *<input >output

You might use this if the input file has a funny extension, when the output file is to have a nonstandard name, or when the output is to be directed to the terminal, for example:

.r rep
 *<input.rat

translates input.rat and writes the output on your terminal. The ability to do this has proved lifesaving when bugs in REP made it bomb off without a diagnostic. Who knows whether such a bug might still exist, and you may get bitten. The idea is to let REP translate the program until it bombs, and you get to see what happened on the terminal. (On the 10, if a

Rep Summary, Page 54
program bombs off, any files being created tend to get lost. There is a way to save them, but no reasonable way to get at data still buffered in core.)

One final way is useful for debugging macros and familiarizing yourself with REP.

```
.r rep
* 
```

Just type a return at the asterisk. This puts REP in fully interactive mode. Now you can type statements and see immediately what REP does with them. When using this mode, the statement reordering feature works against you, so you probably want to turn it off with

```
.mode noreorder 
```

If you are debugging macros, the macpeek builtin macro is very useful. It allows you to look directly at the body of a macro, which may have been defined by another macro.

COMPILING AND LOADING

The output of a REP translation is a standard Fortran program, and is compiled and loaded just like any other. You need to search the REP library in loading. Actually, the library needs to be searched 4 times. The loader doesn’t have the ability to search the library repeatedly until it finds no progress, (Smooth move, DEC!) so you would have to tell it 4 times. But there is a solution. There is a tiny program, called lib.mac, that contains no instructions, only loader directives. It will do the searching for you. If you have a copy of it in your area, you can load and execute a REP program like this:

```
.ex progrname,lib
```

On Dr. Glaser’s machine, that would have to be

```
.ex progrname.for,lib
```

because if you leave out the extension, the machine will find the source program (extension .rat) and run Ratfor (not REP) on it.

DEBUGGING

TOPS-10 provides two debugging tools for Fortran programs, ddt and forddt. Ddt, the assembly language debugger, is recommended only for those who are experienced hackers. Forddt is much better for those experienced in Fortran but not comfortable in the bowels of the 10. Forddt is documented in the Fortran-10 manual available from DEC.
One of the best features of Forddt is that it lets you set breakpoints on any subprogram entry, and at any label. Since REP programs don't have goto's, they don't have many labels, either. REP does generate labels, but you don't know what they are. One way out of this is to put in labels for debugging purposes. A good way to do it is to use small integers and type them at the left margin. This doesn't disfigure the program very much, and you just might leave them in permanently.

If Fortran gives you a diagnostic, it usually reflects a similar error in the REP source program. In nearly all cases, Fortran displays the offending statement, all 80 columns. The identification that REP put in columns 73-80 will be visible too, and can direct you back to the source error. REP's idea of statement numbering may not exactly agree with yours, but it will usually be close. One exception is when the code was generated by a macro. REP will tell you where the macro call was, but not where in the body the offending statement was.

USING REP ON TOPS-20

REP was developed on a 20, and actually runs a little better on the 20 than on the 10. You could use the 10 version on TOPS-20, but there is a native mode version of the i/o library that runs like greased lightning and permits access to long file names. Also available on the 20 is complete jsys support, directly from REP, with the entire MONSYM symbol set defined as macros. (current only through version 3a of TOPS-20). The last version of REP to run on the 20 was 1.99, but is should be trivial to take 3.00 back to it's roots.
PDP-10 PRIMITIVES

The following primitives are very useful on 10’s and 20’s. Many of them could be used with caution even in programs destined to run on other architectures.

DYNAMIC MEMORY ALLOCATION: These operations can be implemented on most systems, but a thorough knowledge of the Fortran runtime environment is needed to implement them. You might have to write a memory allocator, or get memory from the operating system without confusing the Fortran runtime routines. On the other hand, they might serendipitously interface directly with a preexisting facility.

\[ a = \text{alcoa}(i) \]
\[ a = \text{alcor}(i) \]

Allocate \( i \) words of contiguous memory. Return address of leftmost word. \text{alcor} returns ERR if memory not available. \text{alcoa} aborts if memory not available.

\[ \text{call decor}(a) \]

Deallocate an area allocated with \text{alcor}.

BYTE POINTER MANIPULATION: These routines manipulate pdp/10 byte pointers. Although they are very close to the hardware, they could be implemented on other architectures. For a full discussion of pdp/10 byte pointers, see DEC’s hardware documentation. It is recommended that these be hidden away in string handling routines written in Ratfor, or in macros. They go well with dynamic memory allocation.

\[ b = \text{adjbp}(b, i) \]

Perform the adjbp instruction on a copy of \( b \) using \( i \). Return the adjusted pointer.

\[ \text{call dpb}(b, i) \]

Deposit (low order bits of) \( i \) in byte addressed by \( b \).

\[ i = \text{ibp}(p) \]

Increment byte pointer \( p \) with the ibp instruction.

\[ \text{call idbp}(b, i) \]

Increment byte pointer \( b \) and deposit byte \( i \) where it now points.

\[ i = \text{ildb}(b) \]

Increment byte pointer \( b \) and return the byte it now points to.
\( i = \text{ldb} \ (b) \)

Return byte pointed at by byte pointer \( b \).

\( b = \text{point/pointa} \ (\text{object/address}) \)
\( b = \text{point/pointa} \ (\text{object/address, bytesize}) \)
\( b = \text{point/pointa} \ (\text{object/address, bitplace, bytesize}) \)

Make a byte pointer.

Point expects its first argument to be a Fortran data object (variable, array, or constant), and creates a pointer to that object.

Pointa expects its first argument to be an address or an integer expression that evaluates to an address, and creates a pointer to that address.

Otherwise, point and pointa are identical.

Bitplace refers to the position of the rightmost bit of the addressed byte, counting in opposition to the DEC standard from 0 on the right to 35 on the left. Thus bitplace is identical to the \( p \) field of a hardware byte pointer. This choice was motivated by the need to use predefined field definitions from the file monsym.rat, and may be changed in the future. Therefore, \textbf{ALWAYS DEFINE FIELD BOUNDARIES WITH A MACRO.}

If bitplace is omitted, it defaults to 36. Such a pointer addresses an imaginary byte left of the addressed word, and must be incremented before use by an ildp, idbp, ibp, or adjbp. Such pointers are useful as pointers to byte strings, rather than individual bytes.

If bytesize is omitted, it defaults to 7 bits. It is not possible to omit bytesize without also omitting bitplace. Kudos to Fortran.

These formats do not resemble the point pseudo-op in macro, as closely as might be desired. They differ in the order of operands, and the interpretation of the bitplace argument. This is to facilitate field definition by preexisting macros.

No checking is done on the range or size of any argument.

\textbf{BIT FIELD MANIPULATION:} These operations extract contiguous bit fields from a word. They are related to the point/pointa primitives above, and programs using them can be considered
somewhat portable, depending on how they are used.

\[ i = \text{field}(\text{word}:i, p:i, s:i) \]

Extracts a field of bits from a word and returns it right justified.

The bit addressing conventions are: \(<p>\) is the number of bits to the right of the rightmost bit of the field in a word, and \(<s>\) is the size of the field in bits. These are exactly what would appear in a byte pointer to access the field, in the p and s parts of the pointer.

The above convention differs slightly from that used in the pdp-10 assembler point pseudo-op. The reason for this is that the field definitions in monsym.rat follow the field/sfield convention, and the primary motivation for these routines is to access partial words used in conjunction with jsys's. In the future, we hope to correct this situation, so ALWAYS DEFINE THE FIELD WITH A MACRO.

\[ i = \text{sfield}(\text{word}:i, p:i, s:i, \text{value}) \]

This is the opposite of field, which see. Sfield substitutes (the low order bits of) \(\text{value}\) in the field of \(\text{word}\) addressed by \(p\) and \(s\), with bit addressing conventions as in field, point, and pointa. This is a pure function, i.e. the substitution is made only in the function value, not in \(\text{word}\) itself, so \(\text{word}\) can be any expression, possibly involving indirect addressing via iaddr/inhalt.

HALFWORD OPERATIONS: These operations manipulate the 18 bit halves of 36 bit words. They could probably be implemented on any architecture, as long as there are enough bits for the data. In evaluating the portability of existing programs, though, one had better be cautious. These operations are often used in manipulating vary machine dependent data; for instance, they could be used to extract or replace the address part of an instruction. But if they are merely used to pack 2 small integers in the space normally occupied by a larger one, that might be workable. The funny names (HLLE, etc.) are exactly the names of pdp-10 instructions, so look in a pdp-10 manual for further info.

\[ i = \text{halves}(\text{left}:i, \text{right}:i) \]

Compose a word of the right halves of the two
arguments.

i = hle (i)
i = hlo (i)
i = hllz (i)
i = hlre (i)
i = hiro (i)
i = hlrz (i)
i = hrle (i)
i = hrlo (i)
i = hrlz (i)
i = hrre (i)
i = hrro (i)
i = hrrz (i)

These are equivalent to the machine instructions of the same names: Get a Halfword, from the Left or Right of the argument, return it in the Left or Right of the result, with the other half of the result set to Zeroes, Ones, or the Extension of the sign.

i = left (i)

Return left half of argument in right half of result with zero fill.

i = right (i)

Return the right half of the argument, with the left half zeroed out. Same as hrrz.

INDIRECT ADDRESSING: These could probably be implemented on any Fortran, and can be used with care in portable programs. These go especially well with dynamic memory allocation.

i = haunt (a)

Return the word at address a.

a = iaddr (anything)

Returns the address of the argument, which should be a Fortran variable, array, array element, or an external name.

SHIFTING: A simple shift operation. Probably portable to most systems.

i = ls (i, n;i)

Logical shift of i left by n. (right if n negative) end off, no sign extension.
EXPLICIT PACKING AND UNPACKING OF STRINGS: These depend heavily on the representation of character data used in the 10. Better to avoid them.

\[
i = \text{packd} (\text{from:sp, i, to:p, j, delim:c})
\]

\[
i = \text{packdn} (\text{from:sp, i, to:p, j, delim:c, n})
\]

\[
i = \text{packn} (\text{from:sp, i, to:p, j, n})
\]

Copy the from string into the to area, starting with the i'th word of from and the j'th word of to. If i or j is zero, do no copying. The result string is guaranteed to be packed and terminated with a null. The source string may be packed or unpacked, and delimited by an explicit count (n), an explicit character (delim), or a standard EOS, depending on which of the three entries is called, and which is seen first in the source. Delim, if found is not moved. Returns the number of characters moved.

\[
i = \text{unpkd} (\text{from:ps, i, to:s j:i, delic:c})
\]

\[
i = \text{unpkn} (\text{from:ps, i, to:s j:i, count:i})
\]

Same as packd, packdn, packn, except that the result is unpacked, rather than packed.

DIRECT ACCESS TO CHARACTERS IN PACKED STRINGS: These load and store characters in packed character arrays. It is probably better style to use the regular Ratfor type character declarations, and if necessary, later use macros such as ps to pack them if necessary.

\[
c = \text{igb7 (p, i)}
\]

Return i'th character in packed character array p.

\[
\text{call stb7 (p, i, exp:c)}
\]

Store character exp in i'th byte of packed character array p.

MISCELLANEOUS OPERATIONS

\[
\text{call DEFAULT (code:i, val:ps)}
\]

Set default DEVICE or EXTENSION for subsequent opens, removes, renames This is characteristic of the file system on the 10, and may have no meaning on other systems if you must use them, best to arrange the program so that they are compiled only in the 10 or 20 version.

True if user typed control-o since last icochk call. Used only by the user_signal macro locally added to
the Software tools editor. An equivalent type of signal might be available on some interactive systems.

call redir (std:fd, file:fd)

Redirect STDIN, STDOUT, ERROUT to file Used by REDIRECT macro Really a part of the i/o system, and could be implemented on any system.
A DYNAMIC STRING HANDLING PACKAGE ON THE PDP-10

This set of routines permits handling strings of any length, up to the limits of available address space. This package was not designed for portability, but could be modified into a portable package, contingent upon the porting of the memory allocation and byte pointer primitives. These routines are all written in Ratfor. A few of them dissect byte pointers, and are therefore have to be considered nonportable primitives.

All of the routines in this package use pdp/10 byte pointers to address strings. Those that operate on existing strings (except delstr) can operate on any string, wherever the pointer points, not just in dynamically allocated memory. All string arguments are passed as byte pointers, pointing one character to the left of the first character in the string. These routines are oriented towards Ascii data, and use the same EOS convention used in handling regular Ratfor strings. String arguments can have any byte size from 1 to 36 bits, although only 7 and 36 are likely to be used for ascii data. This package does not contain routines for accessing single characters. For that, use the byte pointer primitives. Use the point primitive if you need a pointer to a string contained in static (i.e. Fortran) storage.

Please note the distinction between a string pointer and a byte pointer. A string pointer is the byte pointer returned by newstr, strsav, strsev, etc., when a string is created in dynamic storage. It points one character to the left of the first character, and can be used in all string and byte pointer operations, but it has a special bit pattern that cannot be recreated by any of the byte pointer primitives once it has been modified. Delstr checks for this bit pattern, so you must preserve an exact copy of the initially created pointer if you intend to delete the string from dynamic storage. It is for this reason that most of these routines preserve the exact value of any pointers passed to them.

STRING CREATION: These primitives create new strings in dynamic storage. The pointer they return must not be modified in any way, or else the string can never be deleted from dynamic storage.

\[ sp = \text{newstr}(\text{length},\text{size}) \]

Allocate a null string of specific length and byte size. The string will hold length characters plus an EOS.

\[ sp = \text{strsev}(ss) \]

Save a string, forcing byte size to 7 bits. Return the pointer.
\texttt{sp = strsav (ss)}

Save an exact copy of string \texttt{ss} in dynamic storage return a string pointer to the copy.

\texttt{sp = plit ("literal")}

Save a modified copy of a Hollerith literal and return a pointer to the copy. The modifications to the original in the copy are

1. Backslash sequences (as in C) are converted to their actual values
2. Then any trailing blanks are removed.
3. Then up to one trailing period is removed.

These strange conventions allow the use of certain special characters in literal strings, and are also compatible with the trailing period convention used in error and remark.

The removal of trailing blanks is necessary because fortran-10 pads literal strings with blanks out to a full word. The trailing period need only be used to protect real blanks at the end of the string. If you need a real trailing period, however, you must use two of them. If you have any hopes of portability, you must always use the trailing period.

The removal of trailing characters is done after the backslash conversion, so backslashes can't protect trailing periods or blanks.

The backslash sequences are

| \texttt{\n or \N} | \texttt{NEWLINE} |
| \texttt{\\} | \texttt{backslash} |
| \texttt{\t or \T} | \texttt{tab} |
| \texttt{\b or \B} | \texttt{backspace} |
| \texttt{\r or \r} | \texttt{carriage return} |
| \texttt{\f or \F} | \texttt{form feed} |
| \texttt{\q or \Q} | \texttt{single quote} |
| \texttt{\d or \D} | \texttt{double quote} |

\textbf{STRING DESTRUCTION}

\texttt{call delstr(sp)}

Delete a dynamic string saved with \texttt{strsav} or \texttt{strsev} or created with \texttt{newstr}. The pointer passed must be an exact copy of the pointer originally returned when the string was created. It is not possible to recover that pointer once it has been modified with \texttt{ibp}, \texttt{idpb}, \texttt{adjbp}, etc.
STRING MANIPULATION: Unless otherwise noted, arguments are byte pointers, and are not modified. The pointer, in all cases, must point 1 character to the left of the first character in the string, since the characters are retrieved with ildb's.

p = strcpy (to,from)
Copy string from to string to, stopping after moving EOS. The pointers are incremented BEFORE the move. Returns a pointer to the EOS in the receiving string

p = strcat (to,from)
Concatenate string t to string s, stopping after moving EOS. The pointers are incremented initially. Returns a pointer to the EOS in the receiving string

i = strcmp (ss,tt)
Compare two strings byte by byte in binary from left to right, stopping on EOS in either string. The pointers are incremented before the compare. The value returned is 0 if the strings are equal, negative if string tt > string ss, else positive.

i = strlen (ss)
Count the characters in string ss, incrementing first

i = strndx (str,pat)
Find the index of pat in str; return 0 if no match.

sp = substr (str,from,count)
Extract a substring from string str, of length count, starting with the from'th character. If from is negative, index from the right of the string. If from out of range to the left, start with the first character. If from is out of range to the right, return a null string. If count is negative, count leftward, but don't reverse the characters. Returns a pointer to a new dynamic string containing the substring. The byte size of the new string is always 7. CAUTION: the name is the same as the built in macro "substr", so to use this routine you must bracket or percent the name.
c = strint(p, i)

Convert a string to integer, ADVANCING THE POINTER. return first character after the number as value of strint. return ERR if first nonblank, nontab character is not a digit or minus sign, or if the first character after the minus sign is not a digit. Ignore leading blanks and tabs.

sp = intstr(arg)

Convert integer to decimal string with possible leading minus sign. Save the result as a 7 bit dynamic string, and return the pointer.

INPUT/OUTPUT

call strput(s, fd)

Write string from pointer on file fd.
This section describes a method of optimizing the amount of memory required for string data, perhaps at the expense of processing time, in environments that support no character sized elementary data type. It is intended for use on developed programs, after they are well debugged. The goal here is to make the modifications to the program as minimal as possible.

What is needed is a set of macros that create a new data type for packed strings. The new strings must look just like character arrays to the programmer. This can be achieved using the macro facilities of REP. The macros to do it are quite complex, and translating time may go up dramatically if they are used extensively, but typically it is only worthwhile to pack a few large tables, letting garden variety strings run wild. Part of the complexity of this method is due to the need to emulate Fortran as closely as possible, including the way parameters are passed to subprograms. If you are just creating a new data type from scratch, you can take more liberties.

This method has not been proven in the heat of battle yet, that is, it hasn’t been used in any programs. But there is a working model of a macro package that uses this technique. It is called "ps", for packed string, after the primary data type it defines. The rest of this section describes ps.

When you use ps, you change some of the type declarations in the program from character to ps, for example:

```
character table(80,100)
```

would become

```
ps table(80,100)
```

Arrays of type ps can be in local storage or common. The ps macro declaration starts the considerable machinery into movement. Here’s what it does:

1. First, it records the type and dimensions of table, by defining 2 simple macros: table.TYPE and table.DIMS. Table.TYPE is defined as "ps" (naturally), and table.DIMS is the dimension list, without the parens.

2. Next, it defines the name "table" as a macro with a parenthesized argument (the actual subscript list), an equal-sign argument, and a right argument of the expression variety. This will allow "table" to be used subscripted in any expression or on the left of an assignment.

3. Next is declares 2 genuine data objects: table.SPACE,
large enough to hold the declared dimensions, and table.POINTER, which contains a string pointer to table.SPACE. This pointer is needed for passing the object to subprograms, since ps must mimic Fortran call-by-reference exactly. The automatic name shortening feature converts these funny names to legal identifiers.

4. Finally, ps generates an assignment statement, initializing table.POINTER to the correct string pointer. This is necessary because Fortran can’t create string pointers via data statements. The statement reordering power of REP sorts out the melange.

This takes care of assignments and conditionals. Now, we have to implement call-by-reference for objects of type ps. This is not too bad on the receiving end. All that is needed here is to declare actual parameters as type ps. Well, not quite. They really have to be declared with a stripped down version of the ps macro, called psparm. psparm is the same as ps, except that it omits declaring the space for the object and initializing the pointer.

On the sending end, however, there must be a way of saying, "This use of 'table' is a call by reference." This is done with the macro psref(table) or psref(table(subscripts)). Now, this wouldn’t be too bad for a new program, but we hardly want to go through that hundred page listing, finding and changing all the uses of table in parameter lists.

This is further complicated by the fact that table, or subarrays of it, will surely be used in calls to library routines. So, all the library routines involved have to be modified and recompiled with the program, too. Sounds heavy, and it is. But it only has to be done once per routine. Then they go into a special library for type ps. The modifications to any one routine are minimal, and all the affected routines are in regular Ratfor, not XYZ assembly and carburetor tuning language.

Now arises the problem that some of these routines will inevitably also be used for regular unpacked character strings too, so they have to have separate names. Back to the 100 page listing? No! Macros to the rescue! Throw away the 100 page listing! If you already used it to put in the psref’s, use it again to change them back before you throw it away. The solution is to define all the names of subprograms that handle arrays of type character, library or user-defined, as macros. These macros can interrogate the type of any appropriate arguments to see if they are ps, put in the psrefs, and choose the appropriate alternate entry points. Your precious code escapes unscathed, and only the systems programmer gets ulcers.

The final ps package, then, will contain the macros necessary
to handle all the library routines that take string arguments, and variants of those routines as needed. Not really too bad. The package is even portable. The macros may have to be rewritten for porting, but not the applications. Type ps will not be allowed in equivalence statements, which should be avoided like the kludge anyway, or in Fortran i/o lists, which will also be rare if you use Ratfor I/O, unless you want to modify the compiler and runtime system, which probably ARE written in Carburetor Tuning Language (CT/L).

One final blow: Programs using ps had better have a local frame for every subprogram, as described in the section on the macro generator. Otherwise, the scope of such things as table.TYPE will be greater than the scope of the objects they describe. Funny how bugs creep into programs.
This section describes a package for handling data structures composed of possibly overlapping character fields of fixed size. Field references can have a single subscript to allow tables, blocked records, and such. The package includes a few macros and a set of subroutines. This package has been used in real programs, performs admirably on the pdp-10, and is portable. It should be considered as an alternative to formatted i/o for applications, say of a statistical nature, that read a lot of records with fixed field boundaries.

To use this package, bring in the macros with a "use" builtin macro, and link the subprograms at load time. The macros are created from the same file that generates the subprograms.

**FIXED FIELD MACROS**

field name (base, index, width, offset)

Declares a field

Base is the base address, an array or subarray of type character.

Index is the index of the leftmost byte relative to base.

Width is the width of the field in characters.

Offset (optional) is the distance between the first character of the field, and the first character of its next occurrence. If omitted, it defaults to the length of the field. This parameter is significant only when a field name is subscripted. Its motivation is to allow a specific field to be addressed in one of a set of blocked records, or the like. When defaulted to the length of the field, subscripting addresses groups of adjacent fields, table fashion.

Width, index, and offset may be any integer expressions.

Field names may optionally have a single subscript. The subscript calculation is: index + (subscript-1) * offset. Thus a subscripted reference can be used to pick out one of several adjacent fields, or, when offset is greater than width, nonadjacent fields, in a limited fashion.

Defining a field creates simple macros name.B, name.I, name.L, and name.O defined as the base, index, width and offset, respectively. The field name itself is defined as a macro, that can take one
argument (the subscript). This macro merely expands
into 3 elements separated by commas, the base, the
effective index, and the length. The effective index
is actually an expression that does the subscript
calculation. Knowing this, you can override part of
the field name, if desired, in a macro or subprogram
call. It is correct to use parameters (e.g. name.L)
from previously defined fields in the declaration of
later fields.

move (field1, field2)

Move characters left to right from field1 to field2,
left justified with blank fill.

compare (field1, field2)

Compare the fields in ascii. Sets condition code
HIGH, LOW, or EQUAL. The condition code is the same as
that used by the standard library macro COMPARE,
which compares strings. This operation is not
implemented yet. The macro exists, but not the
comparison function.

Conversion macros

ftos (field, string) - Move and append EOS
stof (string, field) - Move and pad with blanks
itof (integer, field) - To decimal (leading blanks)
itoflz (integer, field) - To decimal (leading zeroes)

SUBPROGRAMS

i = ftoi (field)

Convert a numeric field to an integer. May not be
signed.

call puthol ('string', n, fd)

Write a Hollerith constant (or array containing
packed characters) on fd. n is the number of
characters to write. This really has nothing to do
with packed fields. It just happens to be in the
package.

YES/NO = num(field)

Test a field for numeric, as in Cobol. Returns YES if
the field contains only digits.

i = getrec(fd, field)

Read a record from open file fd into field. The
record must end with a NEWLINE. If the record is
shorter than the field, it is truncated, and if
longer, padded with blanks.

call putrec(field, fd)

The opposite of getrec. Write the field, removing trailing blanks, and append a NEWLINE.

call putfld(field, fd)

Write a field verbatim on a file, and don't add a NEWLINE.
REFERENCES


2. Unix is a trademark of Ma Bell.


5. TOPS-10 Operating System Commands Manual, Digital Equipment Corporation publication AA-0916D-TB.


This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.