NASA CONTRACTOR REPORT 166601

Ames RATFOR User's Guide

Leland C. Helmle

(NASA-CR-166601) RATFOR USER'S GUIDE VERSION 2.0 (Informatics General Corp.) 51 p HC A04/MF A01 CSCL 09B

N85-16490

Unclas
G3/61 13243

CONTRACT NAS2- 11555
January 1985

NASA

NASA CONTRACTOR REPORT 166601

Ames RATFOR User's Guide

Leland C. Helmle
Informatics General Corporation
1121 San Antonio Road
Palo Alto, CA 94303

Prepared for
Ames Research Center
under Contract NAS2-11555

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

Ames RATFOR User's Guide
Version 2.0

by

Leland C. Helmle

Informatics General Corporation

July 16, 1983

Prepared under Contract NAS2-11555, Task 101

Table of Contents

page

Appendix A

1. Introduction

This document describes an extension to FORTRAN† known as RATFOR [1], an acronym for **RAT**ional **FOR**TRAN. The RATFOR preprocessor translates source code written according to the syntax rules described here to FORTRAN code that can then be compiled by a FORTRAN compiler. RATFOR syntax gives the user access to the block-structured control flow features of modern programming languages while allowing the user to exploit the advantages to be gained from using FORTRAN in a scientific or engineering computational environment (popularity, portability, ...). RATFOR does not, however, make any attempt to implement the more sophisticated data structures of languages such as PASCAL; the user is limited to those of the underlying FORTRAN compiler.

This document is intended to serve two purposes: that of providing the potential RATFOR user (who need not be familiar with the language) with a quick reference to the basic language constructs and that of giving the user who already knows RATFOR (or a similar FORTRAN preprocessor) enough information to write and run RATFOR programs on large-scale batch computer systems. Some documents of this type (see, for example, [2] or [3]) are designed to give a brief overview of RATFOR, but in the context of a *specific implementation on a specific machine*. This User's Guide, on the other hand, first presents, as does Comer [4], a description of RATFOR as a language. Separate appendices describe implementations of RATFOR in use on specific systems.

This document assumes that the reader knows at least some FORTRAN and that she has gained (through formal training, brilliant insight or sad experience) an appreciation of the benefits of structured programming. Although our purpose is not specifically to sell the reader on structured programming, the RATFOR user will find that the control structures offered by RATFOR allow her to write well-structured programs in a natural manner. Thus, the use of structured programming concepts and the use of RATFOR are inextricably linked. The user wishing more information on the topic of structured programming should consult the book *Software Tools* [5].

There are now a large number of versions of the RATFOR preprocessor available. The implementations of the preprocessor described in the appendices of

† Although this document is produced with FORTRAN 66 in mind, FORTRAN 77 compilers will also accept the output from RATFOR.

this document are based on a RATFOR dialect called MOUSE4 [6]. This version of RATFOR was chosen for the work described here because the program was specifically designed to operate efficiently in a batch environment and because its design appeared to be more flexible than that of the program released in 1975 and described by Kernighan [1]. The Comer version has proven to be both efficient and easy to modify. MOUSE4 processes code five to ten times faster than an earlier release of RATFOR on the CDC 7600, even though enhanced listing capabilities were added. It should be noted that the name RATFOR was chosen for this release to avoid confusion in the user community; one extra "language" is enough for most people.

The reader not already familiar with RATFOR should continue with the sections described briefly here, while a user who wants to use RATFOR on a particular machine could probably skip immediately to the appropriate appendix. This introductory section is followed by a description of the basic features of the RATFOR language and its usage. These features distinguish the appearance of RATFOR source code from that of FORTRAN and other languages and preprocessors. A quick reference to the basic RATFOR control structures† with examples of their usage is then included. A final section describes features of RATFOR that allow the user added flexibility in designing programs.

The appendices explain the enhancements and the machine-specific features for the Ames RATFOR implementations. The features which were added to make the preprocessor more useful in a batch environment include enhanced source listing capabilities, better error location and reporting, embedded preprocessor directives and options accessible through the operating system command which invokes RATFOR.

Unpublished installation instructions supplied by D. Comer of Purdue University are available in Ames Document 362 in the Computer Information Center at Ames Research Center.

---

† Still more extensive descriptions are available in *Software Tools* [5].

2. Stylistic Features of RATFOR

A major advantage of RATFOR over standard FORTRAN is that users write programs which look better and are easier to read and modify. Using RATFOR assists a programmer in writing well-structured and readable programs. However, program structure is obviously a function of program design and badly designed programs are not likely to be rescued by RATFOR.

A programmer can develop a formal style of displaying what a routine actually does if less effort is needed to avoid certain annoying FORTRAN "features". For example, even when FORTRAN code is properly indented and well-structured, statement labels are an integral part of the control structures. As a result, a person reading FORTRAN code will find his concentration broken frequently because he must constantly shift his eyes from the statement at hand to the statement label in the first six columns.

A more important drawback than statement labels becomes apparent when a section of code must be added or moved. To accomplish this, the programmer must check all new statement labels rather than just variable names for conflicts. What's more, if the logic of a routine is at all complex, the large number of labels needed may make the code incomprehensible.

Consider the example below. Suppose you have inherited a program containing the following code segment. The specifications say *in black and white* that the variable `A(I)` can *never* be equal to `0.0` and your predecessor assumed the specifications were correct. However, you have before you clear evidence that such is not the case: your program bombed because of a very strange value in one element of the array `F`. You have finally traced the problem to this section of code and you see that `F(I)` is not updated if `A(I) = 0.0`. You must add a block of code to recognize and correctly handle this case.

```
C     SHOW AN EXAMPLE WHICH DEMONSTRATES THE ABOVE.

 100          IF ( A(I) .GE. 0.0 )   GO TO 200
                   F(I) =   A(I) + B(I)
     *                    - ( D - E ) / A(I)
              GO TO 300
C
 200          IF ( A(I) .LE. 0.0 )   GO TO 300
                   F(I) =   A(I) + B(I)
     *                    + ( D - E ) / A(I)
C
 300      CALL PUTVAL ( F(I) )
```

Aren't you tempted just a little to rewrite the whole routine when you find that statement 200 can also be reached from three other locations in the routine? RATFOR avoids problems of this type as will be shown in Section 2.1.1.

The purpose of the following sections is to describe RATFOR's way of treating stylistic aspects of programming. Note that RATFOR may vary in certain details from one implementation to another, mostly in the area of the stylistic features discussed here. Also, and more important, an experienced FORTRAN programmer should treat RATFOR as a totally new language. Correct usage of RATFOR control structures will be attained more easily if the programmer avoids simply "translating" his or her FORTRAN code to RATFOR.

2.1. The RATFOR Statement

Even though much of a typical RATFOR program *looks* like FORTRAN, there are some fundamental differences, which are summarized here. The basic differences involve the nature of the RATFOR statement, the way statements are analyzed and the names of the control structures themselves.

2.1.1. Block Structure

An important part of RATFOR's attack on the problems mentioned in the introduction to this section is embodied in the concept of **block structure**. A **block**

of code is defined here as *a sequence of statements from which and into which control can never be passed.* Execution begins at the top of the block and proceeds directly to the bottom (with the exception of loops and excursions into external subprograms). Such a block can be treated as a unit with a single entry and a single exit and its execution is controlled by a single RATFOR control statement.

A **RATFOR statement** may be 1) *a single FORTRAN statement* or 2) *a RATFOR control statement and its associated block of code.* The various RATFOR control statements will be discussed in a later section. A block of code consists of zero or more RATFOR statements that may be executed conditionally or executed more than once as part of a RATFOR loop control structure.

The astute reader may be wondering how these blocks of code are recognized by the preprocessor. There are two conventions: a RATFOR control statement will control 1) the next single statement or 2) a sequence of RATFOR statements bracketed by a pair of *brace* characters, "{" and "}". Where the *brace* characters are not available or not preferred, a number of synonyms may be used. The *square bracket* characters "[" and "]" or the *diphthong* combination "$(" and "$)" are also normally available and can serve the same purpose. Ames RATFOR has been extended to include pairs of *angle bracket* characters "<<" and ">>" as still another alternative.

The RATFOR statement is designed to be executed from top to bottom except when jumping back to the top of a loop or to an external subprogram. Thus, no statement labels are needed within the structure, offering the programmer no temptation to use a `GOTO` statement to branch into a structure. When a programmer finds a situation in which a `GOTO` appears necessary, it is recommended that the segment of code containing the `GOTO`, and possibly the whole routine, be rewritten.

Let's return now to the example mentioned in the introduction to this section. Suppose now that you inherited the same problem as before except that the segment was coded in RATFOR as below. Note that, even though you may never have seen any RATFOR code before, the logic used is *much clearer* and, if a correction can be specified for `A(I) = 0.0`, it can be made by replacing two lines of code (and it is almost certain to work as expected).

```
# THIS CODE DEMONSTRATES A BETTER SOLUTION.

      IF ( A(I) < 0.0 )
        {
          F(I) =   A(I) + B(I)                -
                 - ( D - E ) / A(I)
        }
      ELSE IF ( A(I) > 0.0 )
        {
          F(I) =   A(I) + B(I)                -
                 + ( D - E ) / A(I)
        }
      ELSE # A = 0.0, UNSPECIFIED CASE.
        {
          CALL MESSAGE ( "ERROR:  A = 0.0" )
          STOP
        }

    CALL PUTVAL ( F(I) )
```

The reader may think this is an unfair example. Still, wouldn't you admit that you've seen things like the FORTRAN example in *other* programmers' work and that you have seen so-called "impossible" cases come back to haunt even you a few times?

The RATFOR example above shows that this programmer realized that if A(I) ever happened to be 0.0, the program should do something dramatic to flag the case that was not covered in the specifications. He or she programmed "defensively," covering the illegal case *should it ever arise*, not leaving it to execute incorrectly until the bug happened to cause the program to crash or to give obviously bad results. Notice that there is no need even to check for other possible paths to any point in the above code — there can be *only one*.

Suppose that you want to reorder the cases considered in the above example or in a more complex one. In RATFOR this is essentially a mechanical problem: the blocks of code are simply moved (with some care to be sure that the syntax is left valid). Similar changes in FORTRAN are sometimes simply avoided as "dangerous" unless absolutely necessary.

2.1.2. Tokens

A RATFOR source statement is analyzed by the preprocessor in terms of **tokens**. For this discussion, a token is defined as an alphanumeric string, a string of digits, a quoted string or a single non-blank special character. Tokens of more than one character are delimited by any special character or a blank. Most blanks are removed from the code, since they are not significant to FORTRAN. For example, the statement

```
1000    ERROR = "TRUE"
```

would be separated into the following four tokens, each of which would be treated separately: the label 1000, the alphanumeric string ERROR, the special character = and the quoted string "TRUE". The important thing to remember here is that if you type "I F (A)", RATFOR will *not* recognize this as the beginning of an IF statement. That is, *embedded blanks are not allowed in RATFOR keywords*, in contrast to the case of FORTRAN. In the example above, if ERROR had been typed ER ROR, RATFOR would have processed two separate tokens, but would have generated the same FORTRAN code.

2.1.3. Keywords

The term **keyword**, mentioned in the previous paragraph, can be defined as *those symbols which are recognized as meaningful by RATFOR*. That is, RATFOR keywords are a special set of tokens for each of which the preprocessor must take a very specific action, such as generating a certain type of FORTRAN statement. In RATFOR the keywords are also **reserved** words: they must not be used except as keywords. Thus, variables cannot be named DO or NEXT without hopelessly confusing the preprocessor. The following symbols are designated RATFOR keywords:

STANDARD RATFOR KEYWORDS

Section 3 (Control Structures)	Section 4 (Additional Statements)
IF	DEFINE
ELSE	INCLUDE
WHILE	STRING
FOR	MACRO
DO	
REPEAT	
UNTIL	
BREAK	
NEXT	

2.2. Source Code Format

RATFOR, unlike FORTRAN, does not impose on the user any statement formatting requirements. RATFOR source code is free-format: statements may begin anywhere on a line without regard to column number. Columns one to five are not reserved for statement labels nor is column six reserved for statement continuation characters as in FORTRAN. Thus, the user is able to format his code to demonstrate its logical structure in a natural way without visual interference from statement labels and continuation characters. No attempt is made by RATFOR to indent the user's code, under the assumption that this is a personal style consideration.

RATFOR does not prevent a programmer from using statement labels, but they are necessary only with FORMAT statements. Statement labels may appear anywhere on a line of RATFOR source. How the need for statement labels is eliminated is discussed further in later sections. For now it is worth noting that, partly because of the lack of statement labels in RATFOR code, well-written RATFOR programs are much easier to read and modify than equally well-written FORTRAN programs.

Line continuation is normally indicated in RATFOR by an "_" (*underscore*) character at the end of the line to be continued. This character is not passed to the FORTRAN code being generated by RATFOR. Lines ending in a normally

occurring *comma* (e.g., separating CALL arguments) or *semicolon* (e.g., in a FOR loop, as described in the next section) will also be continued.

RATFOR line continuation conventions have been troublesome in several cases. Ames RATFOR has been modified to *allow* the continuation character at the end of any line containing an incompiete statement. In other implementations the continuation character may be required in some cases, not allowed in others and allowed but not required in still others.

RATFOR also does not, by design, restrict the user to seventy-two columns of source code as does FORTRAN. However, there is always a practical limit to the length of input records on any given system. In some cases, the source may contain card sequencing information beginning at some point in the line. This part of the input to RATFOR must not be processed as source code. The way in which this is handled may vary from site to site. For example, the RATFOR preprocessors described in the appendices allow the user to invoke RATFOR with a parameter specifying the length of the RATFOR source code lines.

The comment character in RATFOR is the "#" (*sharp* or *pound sign*), denoting information which is not to be processed by RATFOR. The characters between "#" and the end of the line are ignored by the processor. Comments may be inserted at any point in a source line, a feature of many computing languages other than FORTRAN. A "C" in column 1 is *not* recognized as a comment character and will generally confuse RATFOR.

A blank line is ignored by the processor and thus serves as a useful formatting device for breaking up sections of source code. A blank line between continued lines of a statement will terminate the statement, cancelling the continuation. The user may, however, use a line containing only a continuation character for such formatting purposes.

A final note concerns the use of lower case letters. RATFOR is designed to accept any ASCII character, including lower case letters, as input. RATFOR then translates all lower case letters to upper case to be recognized by FORTRAN as standard characters. Although this document shows examples in upper case only, the Ames RATFOR preprocessor accepts lower case as well. Note, however, that some equipment operates on a restricted (64 character) character set, converting all letters to upper case, regardless of what the user types. In such an environment, maintaining mixed cases is very inconvenient.

2.3. Additional Stylistic Features

Several other miscellaneous features of RATFOR must still be described. Quoted strings are delimited by either the "`"`" (*double quote*), or the "`'`" (*single quote* or *apostrophe*). RATFOR translates the strings into `H` format Hollerith strings, treating blanks as significant characters. String delimiters must be matched (i.e., a string begun with "`'`" must be ended with "`'`"). This provides a (not quite fool-proof) mechanism for using quote characters in strings: simply bracket the string with the other quote character.

Hollerith strings defined by `H` or such nonstandard FORTRAN delimiters as `L`, `R`, or "`*`" (asterisk) will not be recognized by RATFOR as string declarations. The use of such strings may result in RATFOR preprocessor errors or FORTRAN errors because embedded blanks will be compressed out of the string.

In case a nonstandard RATFOR statement is really needed, the line can be passed through RATFOR without change to the FORTRAN compiler by using a "`%`" (*percent sign*) in column 1. Such a statement is processed by removing the "`%`" and shifting each remaining character on the line one place to the left. (The user must be sure to allow for this if the line is to contain a FORTRAN continuation character in column 6, for example.) This mechanism is useful for the nonstandard string delimiters such as `L` and `R` and other cases which arise from time to time. The user wishing to pass large sections of code or whole routines through RATFOR is directed to the Appendices for the discussion of the FORTRAN pass-through preprocessor directive implemented in those versions of RATFOR.

The relational operators of FORTRAN, such as `.LE.` and `.EQ.` and the logical operators such as `.NOT.` and `.OR.` can all be represented in RATFOR by a convenient shorthand as shown in the table on the following page.

RELATIONAL OPERATORS

.EQ.	==
.NE.	¬=
.LT.	<
.LE.	<=
.GT.	>
.GE.	>=

LOGICAL OPERATORS

.OR.	|
.AND.	&
.NOT.	¬

The original operators are also valid if the user is more comfortable with them. The "==" is used for logical equality, as distinct from "=" which represents replacement. The "¬" (*logical not*), and the "|" (*vertical bar*) are nonstandard characters on many systems and keyboards and may be unavailable (e.g., the CDC 7600 uses a restricted character set of 64 characters which contains neither of these). The "¬" is not even part of most standard ASCII tables, "~" or *tilde*, being the nearest equivalent.

WARNING

Although most implementations of RATFOR have been modified to include various "common" equivalents to the non-standard ¬ and |, using them will very likely make your code less portable.

3. Basic Control Structures

RATFOR control flow structures are described in this section, giving the user enough information to understand the general nature of each structure. Examples, drawn from scientific applications environments, are given for each structure. More detail can be found in Section 1.8 of [5].

The examples which follow assume **strong typing**: each variable in a routine must appear explicitly in a `TYPE` statement. If a variable is named `BLANK` and it is used as an `INTEGER`, the reader can assume that it has been typed `INTEGER` previously. The syntax of each structure is demonstrated by means of a "model" in which upper case letters must be supplied by the user in exactly the form given and lower case letters represent generic information which will be replaced by the user. The following definitions will also be assumed for the discussion which follows:

statement	*any legal FORTRAN or RATFOR statement,* or *one or more of these statements enclosed in compound statement delimiters.*
condition	*any legal RATFOR conditional phrase.*

A RATFOR conditional phrase is a valid FORTRAN conditional phrase using any combination of the standard FORTRAN operators and the operators recognized by RATFOR as described in the previous section.

3.1. The IF-ELSE Statement

```
IF (condition)
     statement 1
ELSE
     statement 2
```

This is performed in the following way: if `condition` is true, only `statement 1` is executed; otherwise only `statement 2` is executed. The `ELSE` part is optional. The `IF-ELSE` construct is frequently implemented in FORTRAN with the following code:

```
      IF (condition)  GO TO 10
      statement 2
      GO TO 20
10    statement 1
20    CONTINUE
```

RATFOR, however, translates the construct in a way which maintains the original structure slightly better:

```
      IF ( .NOT. (condition) )  GO TO 10
      statement 1
      GO TO 20
10    statement 2
20    CONTINUE
```

In the following example of an IF-ELSE construction, subroutine INVERT is called if N is greater than 0; otherwise, the error exit is taken. In either case, processing resumes at the next statement after the right brace which signals the end of the ELSE clause. Note that braces are not needed for the single-statement IF branch. The two-line block of code in the ELSE branch, however, must be surrounded by compound statement delimiters; without them only the call to MESSAGE would be controlled by the ELSE while the call to ABEND would be executed regardless of the value of N.

```
IF ( N > 0 )     CALL INVERT ( X, Y, A, N )
ELSE
 {
   CALL MESSAGE ( 'ERROR:  INVERT REQUIRES N > 0' )
   CALL ABEND
 }
```

3.2. The ELSE IF Construction

```
IF (condition 1)
     statement 1
ELSE IF (condition 2)
     statement 2
     .
     .
     .
ELSE IF (condition n-1)
     statement n-1
ELSE
     statement n
```

This construction is nothing more than a nested IF-ELSE statement, in which each ELSE branch is another IF-ELSE statement. This construction is designed for situations in which *one and only one* of several blocks of code is to be executed, in the manner of the CASE statement of languages such as PASCAL.

The final ELSE, if present, serves as a perfect trap for that "impossible" case as demonstrated in the example of Section 2. This author has *never* regretted expending the effort of adding an extra three or four lines of code needed to print a message and exit "just in case".

3.3. The WHILE Statement

```
WHILE (condition)
     statement
```

This statement is performed in the following manner: Test "condition". If "condition" is true, perform "statement" once and test again. If "condition" is ever false (including the first time), control is passed to the first statement after the body of the WHILE.

This control statement corrects one of the major flaws of the FORTRAN DO statement: the WHILE condition is tested before performing the loop body for the first time. Thus, the loop behaves properly when the loop body may or may not be performed on the first pass. A FORTRAN (before FORTRAN 77) DO loop would require a separate explicit test and branch around the loop to work the same way.

Note the following:

```
WHILE ( INLINE(I) == BLANK )      I = I + 1
```

The above statement can be used to skip any elements of the array INLINE which are equal to BLANK. That is, the "pointer" I will be advanced from its current position to point to the next non-blank character of the array INLINE. If I is pointing to such an element initially, the loop body is not executed and the value of I is not changed. Note that the free format feature of RATFOR allows the one-line statement seen here.

3.4. The FOR statement

```
FOR ( initialize ; condition ; reinitialize )
      statement
```

The "initialize" and "reinitialize" parts represent single, executable FORTRAN statements. A preprocessor does not have all the syntax-checking capabilities of a compiler; using statements other than simple assignments in FOR statements may cause RATFOR to produce incorrect results or even abort. Note the ";" (semicolon) separating the parts of the FOR statements. A common error has been to substitute commas for the semicolons. The Ames RATFOR preprocessor has been improved to recognize and report this error. Other versions may not do so.

Except in one situation (see the section on the NEXT statement), the FOR loop

is performed in the following manner:

```
initialize
WHILE (condition)
  {
     statement
     reinitialize
  }
```

Note that the above expansion can still be carried out even if any one or more of the four parts are missing. A ";" (semicolon) may be used to indicate a missing **statement** section (see the example below). A missing "condition" section is treated as *always true* and an infinite loop results. For a way out of the infinite loop, read the section on the BREAK statement below.

The following example of a FOR statement is equivalent to one which every FORTRAN programmer has used: the simple, standard DO loop.

```
FOR ( I = 1 ; I <= N ; I = I + 1 )
 {
    F(I) =   SIN( X(I) )
    G(I) = - COS( X(I) )
 }
```

The programmer can read this as: "for each I from 1 to N, set ...." Note that there are N iterations of the loop.

The FOR loop, however, is much more powerful. Let us suppose the function GETLIN picks up a record from the file FD, saving the input record in the buffer LINE and returning the length of the record in characters as the function value. The function value is set to -1 on an "end-of-file" condition. The FOR loop in the example below could be used to count the lines and characters in a file, starting

from the current position.

```
NLINE = 0
NCHAR = 0
FOR ( J = GETLIN( FD, LINE ) ; J >= 0 ;
      J = GETLIN( FD, LINE )           )
 {
    NLINE = NLINE + 1
    NCHAR = NCHAR + J
 }
```

Another exotic FOR loop (which performs the same operation as the WHILE loop example in the previous section) is the following:

```
FOR (  ; INLINE(I) == BLANK ; I = I + 1 )     ;
```

In this case I is incremented while INLINE(I) has the value BLANK, leaving I pointing at the first non-BLANK value (we are assuming, obviously, that I has been defined previously). The body of the loop is a "null" statement; all the work is done by the test and the reinitialization statement which increments I.

3.5. The DO Statement

```
DO limits
     statement
```

This structure is translated into a standard FORTRAN DO loop. The "limits" are any legal DO loop limits written in the standard form and subject to the limitations of the local FORTRAN compiler (e.g., I = 1, N). The "statement" is performed as a block, so no statement labels are needed. Any FORTRAN programmer will

be comfortable with this example from the section above on the FOR statement:

```
DO   I = 1, N
 {
    F(I) =   SIN( X(I) )
    G(I) = - COS( X(I) )
 }
```

To the reader who is thinking "This is much easier to read and shorter to type than the FOR loop," consider the following: what happens if N = 0? Experienced FORTRAN programmers may remember that this question is left up to the compiler writers who, before FORTRAN 77, usually specified that the body of the loop would be executed at least once, *no matter what the upper limit might be.* That is, the test for exiting the loop is performed at the end of the loop. The N = 0 case is another of those "impossible" situations that pop up all too often in programming. Where there are not *compelling* reasons to the contrary, the FOR loop is preferred over the DO loop.

Having made the above point, there are still some situations in which the DO loop is appropriate. Most FORTRAN compilers are constructed to optimize a standard DO loop much better than the IF loops generated by RATFOR for the other loop control structures. In particular, the Cray cannot recognize an IF loop for vectorization purposes. Since the test in FORTRAN 77 DO loops is done before the body of the loop is executed, some users will no longer have to worry about "zero-pass" loops. Also, but with more caution, it is recognized that in many computational programs the DO construction is *exactly* what is needed. Calculations may be done a set number of times, there may be no need for unusual index control and there may be no question about whether or not the loop will be done the first time.

The following approach can be recommended. First, design your program carefully. The most dramatic gains in efficiency are almost always found at the algorithmic level. Second, write your code using a language like RATFOR which encourages a well-structured and modular program, parts of which can later be changed if they turn out to be inefficient. Third, once the program works, then (and *only then*) use a timing facility to find out where, in fact, the program is spending its time. Usually about twenty percent of the code is consuming a large part of the execution time. Fourth, work hard at improving those sections found to be inefficient, doing such things as replacing FOR loops with DO loops and replacing modules for which a more efficient procedure can be obtained.

3.6. The REPEAT - UNTIL Loop

```
REPEAT
     statement
     UNTIL (condition)
```

In this case "statement" is simply repeated until "condition" is satisfied. The test is performed *after* each completion of "statement" much as in the case of the DO loop. The UNTIL may be omitted to make an "infinite" loop which must be terminated by some other method (see the section on the BREAK statement, below).

One situation in which the REPEAT-UNTIL loop structure is the most natural way of expressing what is being done arises when an unknown number of records must be read from a file. The example below skips to the "end-of-file" (EOF), starting at the current record.

```
REPEAT
  {
     CALL READER ( FILE, EOFVAL, BUFFER )
  } UNTIL ( EOFVAL == "TRUE" )
```

Note that in the above example, at least one record must be read to enable the system to recognize the EOF condition. We can now treat the problem of exiting a loop early as must be done in the case of an infinite REPEAT loop.

3.7. The BREAK Statement

```
BREAK
```

This statement causes control to pass to the next statement after the body of the current loop (which may be any of the WHILE, FOR, DO or REPEAT loops). Only one

level of loop structure is "broken" at a time; the current loop is terminated and the next outer one takes over.

The example from the previous section can be modified slightly to perform a different function. The loop is now "infinite" to allow the program to process an arbitrary number of records; when an EOF is encountered, the BREAK statement causes control to jump to the CALL REPORT statement without calling subroutine CALC.

```
REPEAT
  {
     CALL CLEAR ( BUFFER )

     CALL READER ( FILE, EOFVAL, BUFFER )
     IF ( EOFVAL == "TRUE" )     BREAK

     CALL CALC ( BUFFER )
  }

CALL REPORT
```

In the above example, the calls to CLEAR and CALC represent any type of processing; the purpose is to show how clearly a RATFOR user can indicate the separation of *work to be done every time* from *work to be done after valid input*. The NEXT statement can be used if the situation calls for skipping only part of a loop.

3.8. The NEXT Statement

```
NEXT
```

This statement immediately initiates the next iteration of the current inside loop, skipping the rest of the body of that loop. Control jumps to the condition section of a WHILE, REPEAT-UNTIL or DO statement; to the top of an infinite REPEAT loop, or to the "reinitialize" portion of a FOR loop. This is the case which was mentioned earlier in which a FOR loop *cannot* be expanded in terms of a WHILE statement. It

should be noted that the NEXT statement is not essential for the logical completeness of RATFOR. The block of code to be skipped could be performed conditionally (with an IF statement) instead.

Expanding again on the previous example, we might find the following:

```
REPEAT
  {
     CALL CLEAR ( BUFFER )

     CALL READER ( FILE, EOFVAL, BAD, BUFFER )
     IF ( EOFVAL == "TRUE" )     BREAK

     IF ( BAD    == "TRUE" )     NEXT

     CALL CALC ( BUFFER )
  }

CALL REPORT
```

In this case, the added statement allows the computation to be done only when the input data satisfies some unspecified validity criterion. As mentioned in the previous paragraph, the NEXT could have been replaced by the slightly more involved:

```
IF ( BAD ¬= "TRUE" )
     CALL CALC ( BUFFER )
```

This completes our synopsis of the basic control flow structures of the RATFOR language. The interested reader may want to consult *Software Tools* [5] for more detailed information. The next section describes features of the RATFOR preprocessor which truly extend the capabilities of FORTRAN.

4. Additional RATFOR Statements

The four RATFOR statements described in this section offer the user capabilities which do not exist in FORTRAN. RATFOR can replace a programmer-defined symbol with various types of information. The DEFINE statement allows RATFOR to save a "symbol" and its "definition", later substituting the definition string for each occurrence of the symbol in the code. The DEFINE statement has been extended in some cases to allow the user to perform simple arithmetic computations in the definitions, allowing the user to define some program parameters in terms of other, more basic, parameters. The MACRO statement allows parameters to be passed to the definition represented by a defined symbol. This, in effect, allows the user to create new types of statements. The INCLUDE statement allows RATFOR to replace a symbol with an image of a complete external file. The STRING statement found in some versions of RATFOR provides the user with a way of defining a simple data structure, character strings, not available in most versions of FORTRAN.

4.1. The DEFINE Statement

```
DEFINE(symbol,definition)
```

DEFINE provides a string replacement capability at compilation time, implementing the concept of a "**symbolic constant**"([5], page 9). The idea is to use a **symbol** to represent something which might, for some reason, confuse the reader or obscure the meaning of the code. For example, a programmer might use the

would be even clearer if we could write:

```
DEFINE(XYSZ, <XSZ*YSZ>)     # DIMENSION OF HORZ SLICE
DEFINE(XZSZ, <XSZ*ZSZ>)     # DIMENSION OF VERT SLICE
```

This extension allows for simple integer arithmetic computations (*, /, + and -), evaluated in a strict left-to-right fashion with no precedence of operations defined. The <...> indicates the part of the definition to be evaluated rather than saved as a string.

Another extension of the DEFINE statement has been implemented in the preprocessors described in the appendices. When using RATFOR in environments supporting only upper case letters (the 7600, for example) a "DEFINEFLAG" character is useful for distinguishing between defined symbols and normal variables. The character chosen for this purpose is the "@" (*at sign*). These preprocessors also recognize the standard unflagged defined symbols (although a programmer might not).

A fairly common extension is that of allowing the alternate syntax:

```
DEFINE ( symbol = definition )
```

This syntax is allowed by the versions of RATFOR described in the appendices. The blanks in this model are also allowed by most RATFOR preprocessors, including those described here.

The use of the DEFINE statement need not be restricted to the cases described here. The following items might be helpful when using this feature.

- The symbol may be longer than a standard FORTRAN variable name. It can be as long as the input line length currently in effect, but it cannot be continued from one line to another.

- Embedded blanks and special characters are not allowed in the symbol but may be in the definition. Some versions are touchier than others in this regard.

- The DEFINE syntax is very strict in some versions of RATFOR: blanks are not allowed even between certain tokens of the statement. These "bugs" should not be a problem in the versions described in the appendices.

- There are installation-dependent limits on the number of definitions and the total length of all definitions processed.

- Only one definition of a symbol is needed for a given RATFOR job. The latest one remains in effect if a DEFINE is repeated. In other words, there is no need to include the DEFINE macros in each routine.

- Where upper and lower case letters are supported, case is significant, so be very careful to avoid confusing the distinct versions of a defined symbol (e.g., EOF, eof and Eof are treated as distinct symbols).

The definition is, itself, processed by RATFOR and may contain one or more symbols which are in turn replaced. This can lead to recursive definitions which will cause RATFOR to abort. Don't, for instance, use a statement such as:

```
DEFINE(FALSE,.FALSE.)
```

In this case "FALSE" will be replaced by ".FALSE." which gets processed further. RATFOR doesn't do anything with the "." but then finds the string "FALSE" and replaces it with ".FALSE." again and so on, producing a long string of ..... and then aborting.

WARNING

A **definition** is available only to those routines compiled in a single execution of RATFOR. If a given **definition** is being changed, the user must be sure to recompile *all* routines needing that **definition**. Routines making use of the old **definition** may show subtle (or unsubtle) changes in behavior!

4.2. The MACRO Statement

```
MACRO ( symbol, replacement )
DEFINE( symbol, replacement )
```

This statement is another extension of the DEFINE statement. The two names may or may not be synonyms, depending on the implementation. The **replacement** string may contain dummy arguments, strings of the form **$n** (where n is an integer between, for example, 1 and 9). The **MACRO** is invoked later in the program by typing **symbol** with real arguments which are inserted by RATFOR for the corresponding dummy arguments in the **replacement** string. The macro expansion is, itself, processed by RATFOR.

A simple example of this usage is the following:

```
MACRO ( BUMP, $1 = $1 + 1 )
```

RATFOR then translates the source code **BUMP(J)** into J = J + 1. Another possibility is the following error reporting macro which would be expanded into two

subroutine calls.

```
MACRO ( ERRFATAL, CALL FATAL ( $1, $2 )
                  CALL ABEND              )
```

This macro might be invoked as follows:

```
IF ( N > NMAX )
  {   ERRFATAL ( ERROUT, "NMAX exceeded" )   }
```

RATFOR would then expand this into the following code which would be further processed as any other RATFOR source code:

```
IF ( N > NMAX )
  {   CALL FATAL ( ERROUT, "NMAX exceeded" )
      CALL ABEND   }
```

4.3. The INCLUDE Statement

```
INCLUDE file
```

Another macro built into the RATFOR preprocessor is INCLUDE. "file" is a (system-dependent) specification for a file external to the process. Most installations use this capability to allow the maintenance of only a single copy of information which is shared by more than one routine or program. When the information changes, a change to the single copy is known immediately through the whole system of routines or programs.

4.4. The STRING Statement

```
STRING  name  'string'
STRING  name(length)  'string'
```

Another type of FORTRAN extension in some versions of RATFOR, including those considered in this document, is the **STRING** statement. The **name** becomes the name of an array, each element of which contains a single character. The array is dimensioned to be large enough for the **string** and a trailing "EOS" (*end of string*) character. Some versions (*not including those described in the appendices*) allow the user to allocate a **length** for the array, larger than that required by the **string**.

The characters in **string** are converted to the decimal equivalent of the ASCII code associated with each character. The corresponding ASCII code is stored one character per word. Thus the string **'A+B'** is stored as the numbers 65, 43, 66 and "EOS" in a four-word array. This feature can be used to enhance portability in programs which routinely make use of such a data structure (the RATFOR preprocessor is an example).

The **STRING** statement supported by Ames RATFOR may not appear useful in many scientific applications. FORTRAN compilers which do not support a **CHARACTER** data type essentially force the user to choose between inefficient storage (one character per word) and inefficient execution (extraction of a single byte from a word through bit manipulations). Furthermore, where ASCII is not the native character set, some type of conversion must be done internally.

Appendix A

A. RATFOR on the Ames Cray X-MP

A version of the RATFOR preprocessor described in the body of this document has been installed on a Cray X-MP computer. Although the implementation of RATFOR described in this appendix was called MOUSE4 by its author, Douglas Comer, of Purdue University [6], that program has been enhanced and released at Ames under the name *RATFOR*. The name RATFOR was chosen for the released program to avoid confusion — the program translates the RATFOR syntax — and to be consistent with the previous usage of RATFOR on the Ames CDC 7600.

The major features of this version are its speed (the program is between three and ten times as fast as the original RATFOR program [1], depending on the RATFOR source being processed), its improved listing format and a number of new options allowed. The speed of the MOUSE4 program received from Purdue has not been seriously impaired by the modifications made here at Ames. (Based on preliminary usage of the Cray version, however, the speed can still be improved substantially.) The listing has been reformatted, with breaks at program unit boundaries and with line number information printed as a debugging aid. Options which have been added to the preprocessor include control of the production of list output, page size of the listing, the width of the RATFOR source line, and the capability of "passing through" a complete FORTRAN routine without change.

A.1. Usage

This section provides the user with enough information to compile and run a basic RATFOR program on the Cray X-MP.

A.1.1. JCL for a Typical RATFOR Job

The following example shows how the RATFOR preprocessor would be accessed in a typical situation in which the user wishes to compile and execute a

program stored in UPDATE program library (PL) format.

```
TESTJOB,MICR.
USER,....
JOB,JN=TESTJOB,....
ACCOUNT,....
  .
  .
  .
UPDATE,....
ACCESS,DN=RATFOR,PDN=RATFOR,ID=LIBRARY.
RATFOR,I=$CPL,F=FTRAN,L=$OUT,PS=45.
RELEASE,DN=$CPL:$PL:RATFOR.
CFT,I=FTRAN,ON=CX,OFF=S,....
RELEASE,DN=FTRAN.
LDR,MAP=PART,SET=INDEF.
  .
  .
  .
```

In the above example, the input to RATFOR (source code) is produced by UPDATE on the file $CPL, the generated Cray FORTRAN (CFT) code is written on the file FTRAN, and the listing is written on the file $OUT. The RATFOR listing will be printed on 45-line pages (PS=45), the generated FORTRAN code will not be listed by CFT (OFF=S), and a full cross reference map (ON=CX) will be produced by CFT. (A discussion of the preprocessor options will be found in Section A.2 below.)

A.1.2. Execution-time Parameter Sequences

The RATFOR file attached in the example above is a core image file, a compiled and linked module which can be referenced by name, loaded and executed. When this is done, no load map will be produced. In effect, the "RATFOR loader call" JCL statement serves as a user-defined Cray Operating System (COS) *command*. In what follows, the RATFOR loader call statement will be referred to as the RATFOR *command*.

The operation of the RATFOR command can be modified by means of a set of "keyword" parameters, as can most other COS commands. These parameters take

the form of **keyword=value** and may come *in any order*. Each parameter may or may not have a default value — a value that will be assumed unless overridden by the user. The Cray RATFOR keyword parameters are described in the section on Preprocessor Options.

A.1.3. Using RATFOR with UPDATE

The UPDATE source file maintenance utility provided by COS will affect Cray RATFOR users in one important way. The **COMDECK** feature of UPDATE is more powerful than the RATFOR **INCLUDE** statement (see Section 4.3). The reason for this is that changing a **COMDECK** automatically causes all routines accessing that information to be recompiled. Because of this, only a limited form of the **INCLUDE** statement has been provided in this version of RATFOR.

WARNING

Changing a DEFINE statement does *not* force the recompilation of routines that use the defined symbol. The user must insure that *all routines making use of a given definition are recompiled if the definition is changed.*

A.2. Preprocessor Options

The default operation of the RATFOR preprocessor can be altered by means of JCL keyword parameters and preprocessor directives which are embedded in the user's RATFOR source program.

A.2.1. JCL Keyword Parameters

The keyword parameters control options which can be specified for an entire RATFOR compilation. They are invoked as keywords on the RATFOR command in much the same way that options are specified on other COS commands. The keywords may come in any order in the RATFOR command. The options currently available are:

I=ldsname Defines the "input dataset" where **ldsname** is the dataset name by which the RATFOR source file is known to the current job. This dataset corresponds to the RATFOR "standard input" dataset. If I is listed, **ldsname** is required. The default value is **$IN**.

F=fdsname Defines the "FORTRAN output dataset" where **fdsname** is the dataset name by which the generated FORTRAN file will be known to the current job. This dataset corresponds to the RATFOR "standard output" dataset. If F is listed, **fdsname** is required. The default value is **FTRAN**.

L=ldsname Defines the "listing output dataset" where **ldsname** is the dataset name by which the listing file will be known to the current job. This dataset receives both RATFOR source listing and error output. If L is listed, **ldsname** is required. The default value is **$OUT**.

PS=nlines Defines the "page size" where **nlines** is the integer number of lines to be printed per page, including the page header. **nlines** must be ten or greater. Standard values are PS=60 for 11-inch pages and PS=45 for 8.5-inch pages. Using PS alone corresponds to PS=45. The default page size is PS=60.

RC=ncol Defines the number of "RATFOR source columns," where **ncol** is the number of columns (beginning in column 1) that will be processed as RATFOR source code. Anything beyond column **ncol** (e.g., UPDATE sequence identifiers) will not be regarded as part of the RATFOR source line. The RC position is denoted on the output page header by a "/" (slash). If RC is listed, **ncol** is required. **ncol** must be in the range 1 to 110, inclusive. The default line length is RC=72.

SL=slist Controls the production of a source listing. If SL=0 (zero) the RATFOR source listing will be suppressed, overriding the list-control directives (L+ and L-) described in the following section.

Omitting the SL parameter, omitting **sllst** or supplying a value other than zero effectively enables the preprocessor list-control directives. The default value is SL=1.

A.2.2. Preprocessor Directives

RATFOR preprocessor directives are a special form of the RATFOR comment statement:

```
#$bXsbXsb. . .
```

where the symbols are defined as follows:

#	RATFOR comment character
$	preprocessor directive flag
b	zero or more blanks
X	any legal directive character
s	sign: + for "ON" and - for "OFF"

The following directives are currently available:

L Controls the production of the RATFOR source listing. SL=0 *disables this directive.* The initial value is ON and the switch value reverts to ON after each END statement. L+ and L- cause the corresponding FORTRAN list-control directives CDIR$ LIST and CDIR$ NOLIST, respectively, to be generated in the FORTRAN output file.

F Controls "FORTRAN pass-through". In pass-through mode, source code card images are copied from the input file ($IN by default) to the FORTRAN output file (FTRAN by default) without alteration, until either an F- directive or an END statement is encountered. This allows entire FORTRAN routines to be "passed through" RATFOR without translation. The initial value is OFF and the switch value reverts to OFF after each END statement.

P Causes an immediate page eject or top-of-form in the RATFOR listing. The default is OFF and the value reverts to OFF immediately

after the page eject. If the list-control switch is OFF (i.e., L-), this directive has no effect.

An example of the usage of these directives is:

```
#$    L- F+
```

which has the effect of enabling FORTRAN pass-through and disabling the source listing. The current line will not be listed (L- is in effect) and the translation of RATFOR to FORTRAN is disabled. This mode will be in effect until changed in another directive line or until the end of the program unit. When an END statement is encountered, the mode automatically reverts to L+ F-, regardless of the previous mode.

Unrecognized directives will be reported as RATFOR syntax errors and the rest of the line will be ignored. Such errors should have no effect on the preprocessor.

A.2.3. The Default RATFOR Command

Using the information supplied in the above sections allows us to specify what will happen if the default values of all parameters are assumed. Executing the RATFOR command as follows:

```
ACCESS,DN=RATFOR,PDN=RATFOR,ID=LIBRARY.
RATFOR.
```

is equivalent to:

```
ACCESS,DN=RATFOR,PDN=RATFOR,ID=LIBRARY.
RATFOR,I=$IN,F=FTRAN,L=$OUT,PS=60,RC=72,SL=1.
```

That is, RATFOR source is read from $IN, the FORTRAN output is written on FTRAN and the listing (including any error messages) is written on $OUT. The list

control directives (L+ and L-) will be in effect because SL is not 0 (zero) and the listing will be written at 60 lines per page. The RATFOR source is assumed to be limited to 72 columns, a value consistent with CFT and allowing UPDATE sequence identifiers to be printed to the right of the listing.

A.3. Additional Notes on the Cray Version

This section is meant to cover a number of loosely related topics relevant to the Cray RATFOR user, primarily in the areas of system and implementation dependencies.

A.3.1. Character Sets

RATFOR utilizes the 7-bit (128 character) ASCII code internally, primarily to increase portability. The same 7-bit ASCII code† is also supported by the Cray, each character being represented by the low order 7 bits of an 8-bit byte. However, the Cray's Cyber front end operates on the input and output characters in such a way as to compress the 128 characters into a 63 character subset. Some inconvenient translations are necessary to allow the user access to the special characters required by RATFOR, a situation which should improve when a Cray front end machine supporting the full ASCII character set is available. The following list summarizes these problems. All character names and symbols are standard ASCII characters.

BRACE There are three alternatives to the RATFOR compound statement delimiters: "{...}." The user may select square brackets "[...]" or either of the "diphthong" combinations "<<...>>" or "$(...$)".

NOT The "¬" (logical negation) symbol used in the book *Software Tools*[5] does not appear in many ASCII tables. The "~" (tilde) has been taken as the nearest equivalent in the Ames environment, but this character is currently unavailable because of the Cyber front end. The only character which works consistently is the "!" (exclamation point or "bang").

† See the FORTRAN (CFT) Reference Manual, Appendix A, for details.

OR The "|" (logical or) symbol is currently unavailable because of the Cyber front end. The upper case equivalent is the "\" (backslash) which is consistently available.

The problem is not one of simple availability. Some of these special characters are, for various reasons, translated incorrectly (from the standpoint of RATFOR) when submitted to the Cray through Remote Job Entry (RJE) terminals; also, some machines translate them differently. Thus, it is recommended that the RATFOR programmer use only the two standard FORTRAN operators .OR. and .NOT. rather than the synonyms provided by RATFOR, to avoid portability problems.

A.3.2. RATFOR Listings

A large part of the effort of implementing this version of RATFOR has gone into producing a source listing which is both useful and æsthetically pleasing. This work has been done on the preprocessor for several reasons. A complete and correctly paginated program listing is useful during the global editing of large program segments (i.e., several thousand lines). In addition, a flexible listing capability has proven essential during debugging in a batch environment. Producing the listing concurrently with the translated FORTRAN code allows the preprocessor to report information which aids the user in finding problems without resorting to listing the intermediate FORTRAN code.

The listing is paginated according to the PS parameter on the RATFOR command. Each page printed includes a header with identifying information for the preprocessor as well as for the routine being processed and a "column template" for the RATFOR source line. The template aids the user in reading program structures which run over more than a single page.

Adding the capability of breaking the listing output at program unit boundaries required that RATFOR recognize a number of new keywords. These keywords are: PROGRAM, SUBROUTINE, FUNCTION, BLOCKDATA, BLOCK, DATA and END. When an END statement, is encountered, RATFOR expects to find one of the program unit statements (PROGRAM, SUBROUTINE, FUNCTION, BLOCKDATA or BLOCK DATA) to follow shortly. If any of these keywords or a DEFINE statement is found in the next few lines (5 lines in the current implementation), a new page header is assembled from the information in the program unit statement. Otherwise, a default "UNNAMED SEGMENT" header is saved. The page header is then available if and when any listing is requested before the next END statement. RATFOR continues searching for a valid program unit statement until one is found, making a corrected page header available if and when another page is printed.

If one or more DEFINE statements are encountered outside a FORTRAN program unit (i.e., between an END statement and the next program unit statement or before the first program unit statement), the page header will include "RATFOR DEFINES". To allow normal pagination for the next routine, an END statement should follow the last DEFINE statement. This END statement will *not* be included in the FORTRAN source generated by RATFOR.

WARNING

> If a program unit statement is misspelled (e.g., SBROUTINE), omitted or unrecognized, the remainder of the program unit will be labelled "unnamed segment". The END statement will then disappear as it does after a DEFINE segment, and this code segment will not compile correctly.

Another major enhancement of this version of RATFOR is the inclusion on the RATFOR source listing of the statement number of the generated FORTRAN statements and the generated statement labels. These numbers, printed to the left of the RATFOR source statements, when used in conjunction with the FORTRAN cross-reference map (ON=CX), provide a powerful tool for tracing program aborts and should eliminate the need for the source listing produced by CFT. It is suggested that CFT be invoked with OFF=S.

Associated with the RATFOR source lines are the following types of information. These items are printed at given intervals in the source listing or as available.

1. RATFOR source line numbers, counted from the beginning of each recognized program unit. Every fifth source line (divisible by 5) is numbered along with the first line of each program unit.

2. FORTRAN statement numbers, counting the FORTRAN *statements* generated as they are written on the output file, FTRAN. These numbers correspond to the statement numbers given in the CFT cross-reference map.

3. FORTRAN statement labels generated by the RATFOR preprocessor are printed as available. For efficiency reasons, this feature obtains the last statement label generated by a given source statement, and should be taken as only approximate when used to locate, for example, the loop in which a run aborted.

A.3.3. Preprocessor Efficiency

With one very important exception, the FORTRAN code generated by RATFOR is nearly as efficient on the Cray as is code written in FORTRAN. The exception is that the RATFOR `FOR` loop (which is translated into a FORTRAN `IF` loop) cannot be recognized as a candidate for vectorization. This can have *grave* effects on program efficiency, but in only a small percentage of all cases. There are at least two factors to be considered in deciding which way to write a segment of code: *Is the loop really time-critical?* and *Is the code likely to be run on other machines which do not support FORTRAN 77?*

The first question can be answered by obtaining some timing data for the entire program. A very simple first step is to compile the program with the CFT "`OI=...F...`" (flowtrace) option and perform a typical run which is long enough to exercise all the normal parts of the code. The flowtrace report gives results in terms of whole routines (not individual loops), but in many cases this will tell the user if the code in question is *really* worth changing or recoding for more efficiency. Code executed only once (e.g., initialization code) should be a very low-priority candidate for additional work.

The second factor is a little more subtle. If the program is likely to be compiled with a pre-FORTRAN 77 compiler or if the program is to be run on several machines interchangeably, the user should be very cautious about changing all `FOR` loops to `DO` loops. The `DO` loop of most FORTRAN compilers, before FORTRAN 77, was usually implemented in such a way that *at least one pass was made through the loop body, no matter what the upper limit was.* In other words, the loop body of the statement

```
DO    I = 1, N
```

will be executed once even if `N` is zero or negative because the exit test is done at the end of the loop body. This type of `DO` loop and the `FOR` loop are *not* equivalent in this way; indiscriminately changing `FOR` loops to `DO` loops may lead to the introduction of errors, some of which will be almost unrecognizable.

A reasonably cost-effective approach to this problem when not using a FORTRAN 77 compiler is to use the `FOR` loop initially, without regard for efficiency. When the program has been debugged and a timing profile has been obtained, *and only then,* replace those few time-critical `FOR` loops at the very lowest level with `DO` loops if this can be done safely (i.e., there is a reason, such as an explicit test, that

the "0-pass" case cannot arise). If the 0-pass case is not prohibited or if nonstandard increments are required, another approach may be necessary.

A.3.4. Fatal Errors and Program Limitations

The RATFOR preprocessor is intended as a tool and should be treated as such and not misused. RATFOR is not as complete or as smart as a compiler — the user bent on tricking RATFOR can do so in a number of ways. Even so, when not mistreated, the program is very reliable. RATFOR has performed well for several years, serving a number of programmers using various styles in a wide variety of applications.

In the rare cases in which RATFOR is pushed past its internal limits and terminates early, the message

```
RATFOR FATAL ERROR.  JOB TERMINATED.
```

should appear in the job's logfile and one of the following messages should be printed on the listing file:

- `END OF INPUT IN A DEFINE`. RATFOR was probably confused by a missing terminating ")" (right parenthesis) in a `DEFINE` statement, possibly many lines earlier.
- `END OF INPUT IN A FOR STATEMENT`. Look for invalid `FOR` loop syntax, also possibly much earlier.
- `INPUT BUFFER OVERFLOW (POSSIBLE RECURSIVE DEFINE)`. This is caused by "pushing back" too many definitions, each of which is being rescanned and further expanded. This probably indicates a set of circular definitions or a recursion (e.g. `DEFINE(FALSE,.FALSE.)`).
- `MAXIMUM STATEMENT LABEL EXCEEDED`. RATFOR generates labels beginning at 23000. A warning is printed after 23999 and this fatal error occurs at label 99999.
- `STACK OVERFLOW IN PARSER`. This probably means that you've really confused RATFOR, but it could mean an internal limit (see below) must be reset. If this is so, you should consider restructuring the current routine.

Cray front end.

- Hollerith strings of the form "6HA B C " will not be recognized as strings by RATFOR and embedded blanks will be compressed out, causing serious problems in the following code. Quoted strings must be used for this situation or the line must be passed through RATFOR by using a "%" (percent sign) in column 1. Reminder: the "%" causes the rest of the line to be shifted one place to the left.

- Users wishing to use the STRING statement must make use of the RATFOR internal ASCII character definitions.

- The STRING statement must, because of a CFT restriction, come *between* the last specification statement (DIMENSION, COMMON, etc.) and the first DATA statement in a routine.

- This version of RATFOR may list error messages near program unit boundaries with the wrong source line.

A.3.7. Differences from Standard RATFOR

- An extension to the DEFINE syntax allows the use of names which begin with the "@" (at sign). This allows users to avoid confusing variable names and DEFINE strings in an environment which supports only upper case letters.

- The DEFINE syntax has been extended to allow statements of the type DEFINE (A = B).

- The original RATFOR and MOUSE4 programs had grave difficulties with the syntax:

```
FOR ( I = 1 , I <= N , I = I + 1 )
```

(Note the commas in place of the normal semicolons.) This implementation correctly recognizes and reports a syntax error for a mistake of this type.

- This implementation has also been extended to handle two problems with DO loop processing. If a statement label is supplied by the user, it will

be removed and reported as an error. Also, this release now allows the "{" (open brace) character to appear on the same line as the DO, a syntax previously allowed for each of the other RATFOR control statements.

- The errors found in any program unit are reported in the listing output regardless of the SL (source listing) parameter. Even if SL=0 or L- (no listing) is in effect, the errors are reported along with the source line in which the error was found.

- There is no external separation between the listing file and the error report file. Internally, however, the files are maintained separately and, with some effort, could be separated for output as well.

- A relatively common error in RATFOR occurs when brace delimiters are not balanced (e.g., a missing "{" or "}"). In earlier versions, this condition would be recognized by RATFOR when it occurred, but could not be reported until the end of the run, many routines later. This error is now reported at the END statement of the routine where it first occurs.

- A summary of the errors encountered in each program unit is printed in the logfile. This is done to warn the user of errors which might not have been noticed in the listing.

References

1. B.W. Kernighan, "RATFOR -- A Preprocessor for a Rational Fortran," *Software-Practice and Experience*, 5, No. 4, 395-406 (1975).

2. D. P. Sykes, "(RSX) RATFOR Documentation, Version 22," (distributed by DECUS), American Management Systems, Arlington, VA, (1980).

3. D. Hanson, J. Sventek, D. Scherrer, A. Akin, "RATFOR Primer," distributed by the Software Tools User's Group, Menlo Park, CA, (1977).

4. D. Comer, "RATFOR Language Specifications and User's Guide," Purdue University, West Lafayette, Indiana, (1977).

5. B. Kernighan and P. Plauger, *Software Tools*, Addison-Wesley Publishing Co., Reading, MA (1976).

6. D. Comer, "MOUSE4: An Improved Implementation of the RATFOR Preprocessor," *Software-Practice and Experience*, 8, 35-40 (1978).

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-166601		

4. Title and Subtitle	5. Report Date
RATFOR User's Guide Version 2.0	January 1985
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Leland C. Helmle	
	10. Work Unit No.
9. Performing Organization Name and Address	K1707
Informatics General Corp. 1121 San Antonio Road Palo Alto, CA 94303	11. Contract or Grant No. NAS2-11555
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics & Space Administration Moffett Field, CA 94035	14. Sponsoring Agency Code 99-53-02 (RTOP)

15. Supplementary Notes

Point of contact: Technical Monitor, Robert Carlson, MS 233-10
Ames Research Center, Moffett Field CA 94035 (415)694-6627, FTS 464-6627

16. Abstract

This document is a user's guide for RATFOR at Ames Research Center. The main part of the document is a general description of RATFOR, and the appendix is devoted to a machine specific implementation for the Cray X-MP.

The main body discusses the general stylistic features of RATFOR, including the block structure, keywords, source code, format, and the notion of tokens. There is a section on the basic control structures (IF-ELSE, ELSE IF, WHILE, FOR, DO, REPEAT-UNTIL, BREAK, NEXT), and there is a section on the statements that extend FORTRAN's capabilities (DEFINE, MACRO, INCLUDE, STRING).

The appendix discusses everything needed to compile and run a basic job, the preprocessor options, the supported character sets, the generated listings, fatal errors, and program limitations and the differences from standard FORTRAN.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
RATFOR, rational FORTRAN, FORTRAN preprocessor, block structure, Cray X-MP, user's guide	Unclassified - Unlimited Star Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	48	

*For sale by the National Technical Information Service, Springfield, Virginia 22161