# RATFOR = FORTRAN/3000 + ELEMENTS OF STRUCTURED PROGRAMMING

Björn Dreher

Institut für Kernphysik der Universität

D 6500 Mainz, West-Germany





Björn Dreher
Institut für Kernphysik der Universität
D-6500 Mainz, West-Germany

## 1. Introduction

RATFOR is a language introduced by Kernighan and Plauger [1] based on FORTRAN-66. In their book "Software Tools" they present a preprocessor that translates RATFOR into standard FORTRAN-IV.

In this paper I will first show why RATFOR is a very useful addition to other common languages and what we are using it for in Nuclear Physics Research and System Programming.

In chapter 3 the RATFOR syntax will be shortly described and some examples will be given.

In the forth chapter I will present our implementation of a RATFOR preprocessor and how to use it on an HP3000 system.

Conclusions about our experience with RATFOR will be drawn in chapter 5.

## 2. Why use RATFOR as an additional language

The primary reason for the authors of RATFOR was to make FORTRAN a better programming language. With RATFOR it is possible to write much more readable and better structured programs. This is achieved by providing additional control structures that are not available in FORTRAN-66, and by improving the "cosmetics" of the language.

The added control structures for better structuring of programs are IF-ELSE, WHILE-DO, REPEAT-UNTIL, FOR loops, DO loops, and others. An INCLUDE statement allows the inclusion of predefined code or definition sequences at certain points. The cosmetics is improved by allowing the programs to be in free-form. The end of the line marks usually

the end of the statement, but statements can easily be continued on the next line by ending with a comma or with a special continuation character. A sharp # anywhere in the line marks the beginning of a comment, thus allowing trailing comments on each line. This certainly encourages programmers to add more documentation to the source code.

Because almost all constructs of standard FORTRAN are retained in RATFOR, it is very easy for a FORTRAN programmer to learn RATFOR. There is only a very low psychological barrier to switch from FORTRAN to RATFOR.

In addition, you are not lost if you have to transfer one of your RATFOR programs to an other installation that has no RATFOR compiler available. You simply move the intermediate FORTRAN code to the other system. This is of course the way, how the RATFOR preprocessor itself is "boot-strapped" on a new machine.

In addition to the already mentioned features, RATFOR comes with a built-in macro processor, which allows not only such constructs like EQUATES and DEFINES as in SPL, but also enables you to add additional language constructs (in the form of macros) to RATFOR as you need it.

From all this you see that RATFOR is a better choice than FORTRAN in at least all those cases, where you have somewhat more complicated control paths in a program. There are only few instances where GOTO constructs are needed, and avoiding those makes programs usually more readable and better self-documenting.

Besides applications in Nuclear Physics, we are using RATFOR to implement data acquisition and measurement control subsystems as well as computer communications systems. RATFOR helps us to write these systems to a large extent in a machine independent way, burrying machine dependencies in macro definitions. We are currently using three type of minicomputers in our institute: HP3000, HP1000, and PE3220.



## 3. RATFOR syntax

### 3.1 General rules

There are several characters recognized as special ones in RATFOR:

```
$(  or {  for LEFT BRACE
$)  or }  for RIGHT BRACE
```

Left and right braces act as delimiters for groups of statements like BEGIN and END in SPL. Other special characters are:

```
^ or ~  for NOT
\ or |  for OR
&  for AND
[* for MACRO LEFT BRACKET
*] for MACRO RIGHT BRACKET
#  for the begin of comments
%  is the line continuation character
```

Any line ending in a comma will also be continued. Include files may be nested 3 levels deep. A statement which starts and ends with a quote will be stripped of the quotes and placed in column one in the output. This is useful for 'hiding' Ratfor keywords and for putting FORTRAN compiler commands ($CONTROL ...) in column 1.

Examples:

```
    '$CONTROL USLINIT,NOSOURCE'
or
          IF(arith expr) label1,label2,label3'
```

(Note: Arithmetic IF statements are not allowed in RATFOR).

Input is free-field with only few exceptions. Capitals and small letters can be used. Embedded comments in a source line start with "#" or "!".

Blocks are one single statement or several surrounded by braces. This is similar to the BEGIN/END structure in ALGOL or SPL. The left brace "{" corresponds to BEGIN, the right brace "}" to END.

### 3.2 The DO statement

It resembles the well known FORTRAN DO-statement without the need to use a label at the final statement.

```
DO I=1,MAX,IDELTA
  A(I)=I
```

or

```
DO I=1,MAX
   {
   A(I)=SIN(X(I))
   B(I)=A(I)**2
   }
```

### 3.3 The FOR statement

```
FOR (I=1 ; I<=100 ; I=I+1)
          <BLOCK>
```

<BLOCK> stands for one statement or { several statements }. The three parts between the parentheses have the following meanings:

1: (I=1) Initialization statement. This may be ommitted, thus starting with a previously defined value.

2: (I<=100) As long as this condition holds true, the following block will be executed. This is tested at the beginning of the block.

3: (I=I+1) Modification, that is performed at the end of the block.

All three clauses may be almost arbitrarily complicated, as the following example shows:

```
FOR (X=0 ; EXP(X)<=1.E70 ; X=ARCSIN(X)+10./Y)
   {
   PRINT X
   }
```



### 3.4 The WHILE statement

```
WHILE ( X(I) ^= 5 )
   {
   DISPLAY X(I)
   X(I)=FUNCT(X(I))
   }
```

This allows a block of statements to be repeated while a certain condition holds true, which is tested at the beginning of each step.

### 3.5 The REPEAT statement

This is the counterpart to the WHILE statement. A block of statements is continued until a certain condition, which is tested at the end of the block, becomes true:

```
REPEAT
   <BLOCK>
UNTIL (X==Y)
```

One may omit the UNTIL-clause to get a REPEAT FOREVER construct.

### 3.6 Exits

The two statements NEXT and BREAK allow to change the sequence of execution in DO, FOR, WHILE and REPEAT blocks without the need for GOTO statements (which is considered as bad programming style!) and labels.

NEXT starts over at the beginning of the currently executing block (i.e. starts again at the first statement of the DO or FOR block after the appropriate modification of the running index - or whatever was requested - has been done: corresponds to a GOTO to the CONTINUE statement of a FORTRAN DO loop)

BREAK continues behind the current block. The DO, FOR, WHILE, or REPEAT statement is terminated.

### 3.7 Relational expressions

The following is a table of the correspondence between FORTRAN and RATFOR relational and logical operators:

| FORTRAN | RATFOR |
|---|---|
| .EQ. | == |
| .NE. | ^= |
| .GT. | > |
| .GE. | >= |
| .LT. | < |
| .LE. | <= |
| .AND. | & |
| .OR. | \| |
| .NOT. | ^ |

### 3.8 IF and ELSE clauses

This is similar as in ALGOL or SPL and many other languages:

```
   IF (logical expression)
      <block>
or
   IF (logical expression)
      <block>
   ELSE
      <block>
```

### 3.9 The INCLUDE statement

The INCLUDE statements allows the inclusion of program parts, which are stored on a different file, at the point of the INCLUDE statement. INCLUDEs may be nested 3 levels deep.

### 3.10 The Ratfor Macro

Ratfor contains a macro processor. It is useful for simple character string replacements, string replacements with parameters, as well as for powerful extensions of the Ratfor syntax. Macros are defined with the DEFINE statement. In the following we give a few examples of simple macro definitions.



```
define(pi,3.141593)
```

Following this macro definition, every occurance of pi (or PI) in the text will lead to the insertion of 3.141593 instead of the two letters.

```
define(maxind,200)
...
integer iarray(maxind),rarray(maxind,2)
...
do i = 1, maxind
  iarray(i) = 0
```

This is useful to define dimensions and maximum index values globally.

```
define(tan,[*sin($1)/cos($1)*])
```

This is a macro with parameters. tan(x/2) will be replaced by sin(x/2)/cos(x/2). With the macro definition, you can write the program as if "tan" were a function, but there are no function calls at run-time. For complicated expressions, however, the object code will be quite long when you call the macro often.

Macros can be globally defined for a complete source file. It is best to make the definitions at the beginning of the file, maybe with an include statement for a file containing the macros.

The following example illustrates how powerful the RATFOR macro processor can be, if you have understood its operation and syntax in detail. For instance, it is possible to write easy to use constructs for condition code checking after the call of system intrinsics:

```
IFN=FOPEN( ... )
   BEGINCC
      CCE
      << block >>
      CCG
      << block >>
      CCL
      << block >>
   ENDCC
```

There is no need to use all three conditions (CCE,CCG,CCL). If one is omitted, control continues for that case after the ENDCC. The sequence of CCE,CCG and CCL may be chosen arbitrarily.

## 4. Our RATFOR/3000 implementation

Our RATFOR implementation was derived from a RATFOR preprocessor originally written for the HP1000 family of computers. Therefore it is capable to produce output for the HP1000 FTN-IV compiler as well as for FORTRAN/3000.

RATFOR/3000 may be invoked by the following UDC:

For FORTRAN/3000:
```
        RATFOR <in>,<out>,<list>,<opts>,<incld>,<ftnlist>
```

For FTN4/1000:
```
        RAT4 <in>,<out>,<list>,<opts>,<incld>,<ftnlist>
```

the Ratfor program will be read from <in> (default $NULL)

the intermediate Fortran code will be written to <out>
(the default is a SESSION temporary file RATTEMP)

the source listing will be written to <list> (default $STDLIST)

options are specified by <opts> (default %17 or %13)
The options are given as an integer constant (octal or decimal). If the most significant bit is number 0 and the least significant is number 15, the bits have the following meanings:

15 list the source, otherwise only errors are listed.
bit 15=0 is automatically set, if <list> = $NULL;
errors are then output to $STDLIST.

14 for future enhancements

13 =1 FORTRAN/3000 code
=0 FTN4/1000 code
automatically set by the two UDC's

12 merge all RATFOR (and other) comments into the generated FORTRAN program

the file <incld> will be included in front of <in> (default $NULL)

the FORTRAN compiler listing will go to <ftnlist> (default $STDLIST)



### 4.1 Limitations

Due to the fact that this version of RATFOR is an adaptation from the HP1000 version there were some features in RATFOR/1000 that did not conform with FORTRAN/3000 syntax. Therefore, if in HP3000 mode some original RATFOR features are switched off. In particular, in HP3000 mode the following applies:

1. CHARACTER declarations are passed as they are, because FORTRAN/3000 supports type CHARACTER variables.

2. Character strings between quotes, e.g. "ABCDEF", are kept as they are. In non-HP3000 mode this is converted to 8HABCDEF.

To allow for the use of substring designators, e.g. I[3:5], in both modes brackets are not recognized as delimiters of blocks as they were in the original version. Use braces "{}" instead!

RATFOR does NOT understand FORTRAN arithmetic IF statements. If you have to use them, e.g. to check the condition code after returning from a system intrinsic, you have to put the statement between quotes. (There is a special RATFOR macro available to check condition codes)

For FORTRAN/3000 applications it is good practice to use the following compiler command:

```
$CONTROL USLINIT,NOSOURCE
```

If you then use the default setting for the FORTRAN/3000 output ($STDLIST) you will not get the awkward FORTRAN listing, but instead all (if at all) error messages with the line in error on your terminal. The FORTRAN line numbers are derived from the original RATFOR source line numbers, with an increment of .001 if there are more than one FORTRAN lines generated from one RATFOR line. Therefore it should be easy to find the RATFOR line, which is in error.

## 5. Conclusion

In conclusion, we found the RATFOR preprocessor a very valuable programming tool, especially since a FORTRAN-77 version for the HP3000 seems to be still far away. Although FORTRAN-77 adds some of RATFOR's control structures, we find the cosmetics, the appearance of the program text, of RATFOR much more appealing.

Now, what is the pay-off? Certainly compilation time is increased. In the current version, the RATFOR compiler needs about the same CPU time to transform RATFOR to FORTRAN as the FORTRAN compiler needs for its job. This can be somewhat improved in the future by sampling the most frequently used parts of the preprocessor and improving on these pieces of code.

In addition you have to be aware, that RATFOR/3000 checks only RATFOR syntax, most of the FORTRAN statements go unchecked to the FORTRAN compiler. FORTRAN/3000 will then give you the errors. Since the line numbers of the intermediate FORTRAN code a derived from the original RATFOR line numbers, it is very easy to track an error reported by the FORTRAN compiler back to the original RATFOR source line.

Regarding run-time performance, we did not find any significant difference between a RATFOR program and a corresponding version written directly in FORTRAN.

Of course, a globally optimizing FORTRAN compiler, which we are all waiting for, would improve the run-time behaviour of RATFOR programs as well as FORTRAN programs.

[1] B.-W. Kernighan, P.J. Plauger: Software Tools, Addison-Wesley Publishing Co. 1976