SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 5, 395–406 (1975)

RATFOR—A Preprocessor for a Rational Fortran

BRIAN W. KERNIGHAN
Bell Laboratories, Murray Hill, New Jersey 07974, U.S.A.

SUMMARY

Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The RATFOR language attempts to conceal the main deficiencies of Fortran while retaining its desirable qualities, by providing decent control flow statements and some 'syntactic sugar'. RATFOR is implemented as a preprocessor which translates this language into Fortran.

Once the control flow and cosmetic deficiencies of Fortran are hidden, the resulting language is remarkably pleasant to use. RATFOR programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their Fortran equivalents.

It is readily possible to write RATFOR programs which are portable to other environments. RATFOR is written in itself in this way, so it is also portable; versions of RATFOR are now running on computers of six different manufacturers.

This paper discusses design criteria for a Fortran preprocessor, the RATFOR language and its implementation, and user experience.

INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs.[1] Finally, Fortran is often the most 'efficient' language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements —conditional branches and loops—which express the logic of the program. The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for '1 to N in steps of 1 (or 2 or ...)', but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N − 1. And of course the DO is useless if one's problem does not map into an arithmetic progression.

The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

Received 19 May 1975

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with RATFOR. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A conference on Fortran preprocessors[3] held in late 1974 drew 31 papers.)

LANGUAGE DESCRIPTION

Design

RATFOR attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of RATFOR is to conceal this part of Fortran from the user, by providing decent control flow structures. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the 'cosmetic' deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects—control flow and cosmetics—RATFOR does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in RATFOR and what should not has been *RATFOR does not know any Fortran.* Any language feature which would require that RATFOR really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the RATFOR language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc.; similarly the cosmetic changes are simple. We shall concentrate on showing what the language looks like.

Statement grouping

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction 'if a condition is true, do this group of things', for example,

```
if(x > 100)
    {call error("..."); err = 1; return}
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
      if(x .le. 100) goto 10
          call error(27h ...)
          err = 1
          return
10
```

When the program does not work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

RATFOR eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in RATFOR. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single RATFOR statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** and of course 'end' already has a Fortran meaning.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character '>' is clearer than '.GT.', so RATFOR translates it appropriately, along with several other similar shorthands. Although the construction "..." is recognized by many Fortran compilers as a character string, it is not allowed in ANSI Fortran, so RATFOR converts it into the right number of 'H's': computers count better than people do.

RATFOR is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if(x > 100){
    call error("...")
    err = 1
    return
}
```

No semicolon is needed at the end of each line because RATFOR assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (RATFOR or otherwise), no braces are needed:

```
if(y < = 0.0 & z < = 0.0)
    write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general RATFOR continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

The 'else' clause

RATFOR provides an **else** statement to handle the construction 'if a condition is true, do this thing, *otherwise* do that thing'.

```
if(a < = b)
    {sw = 0; write(...) a, b}
else
    {sw = 1; write(...) b, a}
```

The Fortran equivalent of this code is circuitous indeed:

```
      if(a .gt. b) goto 10
          sw = 0
          write(...) a, b
          goto 20
10    sw = 1
      write(...) b, a
20    ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the RATFOR version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood and forgotten. The program says what it means.

The syntax of the **if** statement is

```
if(legal Fortran condition)
    RATFOR statement
else
    RATFOR statement
```

where the **else** part is optional. The 'legal Fortran condition' is anything that can legally go into a Fortran Logical IF. RATFOR does not check this clause, since it does not know enough Fortran to know what is permitted.

RATFOR does not provide a **case** statement, since it may be readily simulated with a series of **else if** statements:

```
if(...)
    ---
else if(...)
    ---
...
else
    ---
```

This is an example where the desire for simplicity overcomes the desire to provide a 'complete' set of statements.

The 'do' statement

The **do** statement in RATFOR is quite similar to the DO statement in Fortran, except that it uses no statement number. Thus

```
do i = 1, n {
    x(i) = 0.0
    y(i) = 0.0
    z(i) = 0.0
}
```

is the same as

```
 do 10 i = 1, n
     x(i) = 0.0
     y(i) = 0.0
     z(i) = 0.0
10   continue
```

The syntax is:

```
do legal-Fortran-DO-text
    RATFOR statement
```

The part that follows the keyword **do** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not permitted in ANSI Fortran), they can be used in a RATFOR **do.**

As with the **if** a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
    x(i) = 0.0
```

'break' and 'next'

RATFOR provides statements for leaving a loop early, and for beginning the next iteration. **Break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **Next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
    if(x(i) < 0.0)
        next
    ---process positive element---
}
```

The 'while' statement

One of the problems with the Fortran DO statement is that it insists upon being done once, regardless of its limits. If a loop begins 'DO I = 2, 1' this will be done once with I set to 2, even though common sense would suggest that perhaps it should not be. Of course a RATFOR **do** can be easily preceded by a test

```
if(j < = k)
  do i = j, k {
      ---
  }
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it.

To overcome these difficulties, RATFOR provides a **while** statement, which is simply a loop: 'while some condition is true, repeat this group of statements'. It has no preconceptions about why one is looping. For example, this routine to compute sin (x) by the Maclaurin series combines two termination criteria.

```
function sin (x, e)
        # returns sin(x) to accuracy e, by
        # sin(x) = x-x**3/3!+x**5/5!-...
    sin = x
    term = x
    i = 3
    while(abs(term) > e & i < 100){
        term = -term * x**2 / float(i*(i-1))
        sin = sin+term
        i = i+2
    }
    return
    end
```

Notice that if the routine is entered with *term* already smaller than *e*, the loop will be done *zero times*, that is, no attempt will be made to compute *x**3* and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears—the code works at one of its boundaries.

As an aside, a sharp character '#' in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line—one can make marginal remarks, which is not possible with Fortran's 'C in column 1' convention.

The syntax of the **while** statement is

```
while(legal Fortran condition)
    RATFOR statement
```

As with the **if**, 'legal Fortran condition' is something that can go into a Fortran Logical IF.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then to find the first non-blank character requires simply

```
while(nextch(ich) = = iblank);
```

(A semicolon by itself is a null statement; '= =' is '.EQ.'.) When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in Fortran as

```
100 IF(NEXTCH(ICH) .EQ. IBLANK) GOTO 100
```

but few Fortran programmers even believe this line is legal. The language at one's disposal strongly influences how one thinks about a problem.

The 'for' statement

The **for** statement is the final RATFOR control flow construct. It attempts to carry the separation of loop-body from reason-for-looping a step further than the **while**. A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
for(i = 1; i< = n; i =i+1) ...
```

and the loop of the sine routine in the previous section could be re-written as

```
for(i = 3; abs(term) > e & i < 100; i = i+2){
    term = -term * x**2 / float(i*(i-1))
    sin = sin+term
}
```

The initialization and increment of *i* have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The syntax of the **for** statement is

```
for (init; condition; increment)
    RATFOR statement
```

init is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init*, *condition* and *increment* may be omitted, although the semicolons must remain. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat.

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out directly. For example, here is a 'backwards DO loop' to find the last non-blank character on a card:

```
for(i = 80; i > 0; i = i - 1)
    if(card(i) != blank)
        break
```

('!=' is the same as '.NE.'.) The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If *i* reaches zero, the card is all blank.

The increment need not be an arithmetic progression; the following program walks along a list until a zero pointer is found, adding up elements from a parallel array of values:

```
sum = 0.0
for(i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

Cosmetics

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if RATFOR can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
+ - * / , | & (
```

are assumed to be continued on the next line.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1–5 upon output. Thus

```
write(6, 100); 100 format('hello')
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to *n***H**... but is otherwise unaltered (except for formatting—it may get split across card boundaries during the reformatting process).

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program).

Symbols like '>' or '>=' are translated in the obvious manner unless they occur within either single or double quotes or on a line beginning with a '%'; '&' and '|' become '.AND.' and '.OR.' The brackets [and] are synonyms for the braces { and }.

define: Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest

of the definition line (comments are stripped off). **define** is typically used to make symbolic parameters:

```
define  ROWS  100
define  COLS  50
dimension a(ROWS), b(ROWS, COLS)
    if(i > ROWS | j > COLS) ...
```

include: The statement

```
include filenumber
```

inserts the file found on input stream *filenumber* into the RATFOR input in place of the **include** statement.

IMPLEMENTATION

RATFOR was originally written in C,[4] a high-level language reminiscent of BCPL, on the UNIX operating system.[5] The language is specified by a context free grammar and the compiler constructed using the YACC compiler–compiler.[6]

The RATFOR grammar is simple and straightforward:

```
prog : stat
     | prog  stat
stat : if(...) stat
     | if(...) stat else stat
     | while(...) stat
     | for(...; ...; ...) stat
     | do ... stat
     | break
     | next
     | digits stat
     | {prog}
     | anything unrecognizable
```

The observation that RATFOR knows no Fortran follows directly from the production that says a statement is 'anything unrecognizable'. In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition 'unrecognizable'.

Code generation is also simple. If the first token on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L + 1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if(.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L  continue
```

is generated, unless there is an **else** clause, in which case the code is

```
    goto L+1
L   continue
```

In this latter case, the code

```
L+1 continue
```

is produced after the *statement* part of the **else.** Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing **else,**

```
if(i > 0) x = a
```

should be left alone, not converted into

```
    if(.not. (i .gt. 0)) goto 100
    x = a
100 continue
```

But what are optimizing compilers for, if not to improve code ? It is a rare program indeed where this kind of 'inefficiency' will make a large difference. In the few cases where it does, the offending lines can be protected by '%'.

The use of a compiler–compiler is definitely the preferred method of software development. The language is well-defined, with no syntactic irregularities. Implementation is quite simple; the original construction took under a week. However, the language is sufficiently simple that an *ad hoc* recognizer could be readily constructed to do the same job if no compiler–compiler were available.

The C version of RATFOR is used on our local Honeywell 6070 and PDP-11. C programs are not portable, however, and there was a need for a RATFOR that could be moved to other machines. A new version of RATFOR was written in itself and bootstrapped with the C version. The RATFOR version was written so as to translate into the portable subset of Fortran described in Reference 1, so it is portable. This code has been run essentially without change on the machines of six different vendors. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v \pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. RATFOR itself will not gratuitously generate non-standard Fortran.)

The RATFOR version is about 1,500 lines of RATFOR (compared to about 750 lines of C); this compiles into 2,500 lines of Fortran. Both figures are deceptive, however, since the compiled code contains unnecessary occurrences of COMMON declarations and numerous CONTINUE statements that are never referenced. Similarly the RATFOR source uses white space generously in an attempt to be readable. The expansion ratio seems typical in spite of this.

The execution time of the RATFOR version is dominated by three routines. Sixty per cent of the time is spent in two routines that read and write cards; 20 per cent is spent deciding whether input characters are letters, digits or others. Clearly these three routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is irrelevant. It does confirm the folk-theorem that 10 per cent of the code takes 90 per cent of the run time.

EXPERIENCE

Good things

At the moment there are perhaps forty RATFOR users at Bell Labs. 'It's so much better than Fortran' is the most common response of users when asked how well RATFOR meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

One interesting and encouraging fact is that programs written in RATFOR tend to be as readable as programs written in more modern languages like Pascal. For example, here is a RATFOR implementation of the linear table search discussed by Knuth:[7]

```
A(m+1) = x
for(i = 1; A(i) != x; i = i+1);
if(i > m) {
    m = i
    B(i) = 1
}
else
    B(i) = B(i)+1
```

Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even aesthetically pleasing.

Although there are no quantitative results, users feel that coding in RATFOR is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read.* The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. An of course it is easy to do structured programming in RATFOR; this self-discipline also contributes markedly to reliability.

Bad things

The biggest single problem is that Fortran syntax errors are not detected by RATFOR but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in some cases this may be difficult to relate back to the offending RATFOR line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Users also complain that the generated Fortran is 'unreadable' because it is not tastefully formatted and contains extraneous CONTINUE statements.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, the continuation convention says that a line which ends with a slash '/' should be continued, since the slash is probably an arithmetic operator. But the Fortran DATA statement also ends with a slash. Since RATFOR truly does not know any Fortran it cannot reliably recognize when it is dealing with a DATA statement. Thus one must terminate each DATA statement with a semicolon. Another less serious difficulty is that keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. It is hard to work up much sympathy for them, however.

The construction

```
if(...)
    stop      # or return or goto
else
    thing
```

generates an inaccessible GOTO after the STOP statement. Most Fortran compilers produce a warning diagnostic, which is disconcerting the first time encountered. The problem may be solved by removing the **else**, which is logically unnecessary.

A few standard Fortran constructions are not accepted by RATFOR, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap.

CONCLUSIONS

RATFOR demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in 'features'—things which the user may trivially construct within the existing framework. For example, RATFOR does not provide a **repeat** statement which is a loop with its test at the bottom. This statement encourages programs which fail at their boundaries. In the few cases where it is needed, it can be easily simulated with an infinite loop and a test and break at the bottom.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for RATFOR to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

ACKNOWLEDGEMENTS

C. A. R. Hoare once said that 'One thing (the language designer) should not do is to include untried ideas of his own.' RATFOR follows this precept very closely—everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran.[8]

I am grateful to Stuart Feldman, whose patient simulation of an innocent user led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the RATFOR version of RATFOR.

REFERENCES

1. B. G. Ryder, 'The PFORT verifier', *Software—Practice and Experience*, **4**, 359–377 (1974).
2. American National Standard Fortran, American National Standards Institute, New York, 1966.
3. 'Workshop on Fortran preprocessors for numerical software', Pasadena, Calif., Nov. 1974.
4. D. M. Ritchie, 'C Reference Manual', Bell Labs. internal memorandum, 1974.
5. D. M. Ritchie and K. L. Thompson, 'The UNIX time-sharing system', *Commun. ACM*, **17**, No. 7, 365–375 (1974).

6. S. C. Johnson, 'YACC—yet another compiler-compiler', Bell Labs. internal memorandum, 1974.
7. D. E. Knuth, 'Structured programming with goto statements', *Computing Surveys*, **6**, No. 4, 261–302 (1974).
8. A. D. Hall, 'The Altran system for rational function manipulation—a survey', *Commun. ACM*, **14**, No. 8, 517–521 (1971).