1977

MOUSE4: An Improved Implementation of the RATFOR Preprocessor

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Report Number:
77-236

MOUSE4: An Improved Implementation of the RATFOR Preprocessor

Douglas Comer
Computer Sciences Department
Purdue University
West Lafayette, Indiana

CSD-TR 236

June 1977

Abstract

RATFOR is a preprocessor language for FORTRAN that supports structured flow of control statements and macro substitution. The RATFOR processor, written in RATFOR, is modular, carefully coded, and portable, but extremely inefficient. A profile of the running time revealed that a linear search in the macro processor consumed over half of the CPU time. Running time was reduced by over 50% when a binary search was used. Our observation is of interest primarily because it differs from previously reported measurements.

A more dramatic improvement in running time was obtained by rewriting the ad hoc lexical scanner using a standard method based on finite automata. For a 3000 line source program the standard RATFOR required 185.470 CPU seconds on a CDC 6500 while the automata based version needed only 12.723 seconds.

We conclude that, contrary to evidence exhibited by the designer of RATFOR, sequential search is often inadequate for production software. Furthermore, in the case of lexical analysis, well-known techniques do seem to offer efficiency while retaining the simplicity, ease of coding, and modularity of ad hoc methods.

Introduction:

RATFOR[1] is a preprocessor language for FORTRAN designed by Kernighan [1]. It supports structured flow of control, macro substitution, and file inclusion. Additional information about RATFOR is available in the text [2], which includes most of the source code for RATFOR in the chapter on preprocessing. While the distributed software is reliable, modular, well structured, and surprisingly portable, it is extremely inefficient. The designer asserts[2], and we agree, that even if using a preprocessor doubles the cost of compiling a program it is worthwhile. Unfortunately, we found a much greater discrepancy in running times between RATFOR and a local FORTRAN compiler as shown in Table 1. The cost ratios, in terms of CPU times, are from 10:1 for small programs to 19:1 for RATFOR compiling itself.

We have used RATFOR in a course at Purdue University for the past year. With over 200 students using RATFOR, the total CPU usage averaged over 3 hours per day on a CDC 6500 -- more than any other account at the university. It became evident that the efficiency of RATFOR would have to be improved if we were to continue using it.

The improvements were done in two steps. First, several routines were identified as "high spots" in the processor and were recoded to improve efficiency. During this process it became apparent that a reorganization of the lexical scanner could yield a more dramatic improvement. In the second step, the scanning and macro processing routines were rewritten. The reorganization and its result on running time will be discussed after an overview of the pertinent parts of RATFOR is given.

---

[1] This paper refers to the software described in [2] which is distributed by Addison Wesley in machine readable form as supplemental material.

[2] [2] page 315.

2 Overview of RATFOR:

The organization of RATFOR is shown in Figure 1. At the lowest level, GETCH returns characters, one at a time, from the current input file (MAP being used to translate the native character set to ASCII). NGETCH maintains a stack of "pushed back" characters; it returns the top character on the stack when invoked, calling GETCH when the stack is empty. This push back mechanism is used extensively in RATFOR to permit look-ahead. Notice that if the parser, for example, needs to look ahead at the next token, all the routines in between will be called to get a token which is then pushed back at the character level. Thus, the next call will invoke the entire scanning process again.

GTOK is actually the lexical scanner and classifies the token as alphanumeric ("ALPHA") or gives the numeric value of other symbols. GTOK also handles string constants and a few other details. Since any alphanumeric token is a potential macro call, DEFTOK calls LOOKUP to search the table of macro names for each ALPHA it receives from GTOK. If a macro is called, DEFTOK "pushes back" the definition and begins again. New macro definitions are also handled by DEFTOK using GETDEF to install the name and value in the table. One level above DEFTOK, the routine GETTOK, does "inclusion" of alternate source files by changing the input stream temporarily. The details of file inclusion are installation and operating system dependent.

At the top level, PARSE calls LEX to classify keywords. LEX, in turn gets tokens from GETTOK. Naturally, many other routines not shown in the diagram are called to process each statement and generate code.

3 Results of a Profile:

In [2] measurements of RATFOR are given. Although LOOKUP (which is called for each alphanumeric token in the input) employs a linear search, the measurement indicates that it consumes so little time that it is probably not worth improving. The authors even remark that "(the measurement) ... supports our contention that a linear table search is often adequate."[3]

The results from our profile were quite unexpected. While compiling itself, RATFOR spent 60% of its time looking up tokens (there were 127 defined symbols in our version). By changing LOOKUP to use a binary search, the running time of RATFOR was cut to 40% of its former value. Some time was saved from the routine which actually compared the macro names, but most of it was from LOOKUP itself. Even with a binary search the program spends 15% of its time in LOOKUP; a hash method reduces this to about 4%.

An "optimized" version of RATFOR was produced using the binary search in LOOKUP. By making a few more changes like skipping blank lines and not translating comments to ASCII, the running time was reduced to 1/3 of its original value for a 3000 line program.

4 A Reorganized Version of RATFOR:

While RATFOR was being modified it became apparent that substantial improvements could result from a reorganization of the lexical analyzer. A new processor, called MOUSE4, was written with the organization shown in Figure 2. At the lowest level, it was observed that a separate routine, NEXTCH, caused unnecessary overhead and could be replaced by a simple data structure. In MOUSE4, as each line is read, it is placed in the right-hand

---

[3] in [2] page 316.

end of an array as shown in Figure 3. NEXT and LAST give the positions of the next character to examine and the last character to use. If "push-back" becomes necessary, the string to be pushed back is copied into the array immediately to the left of NEXT, which is then updated. Characters are always removed from the same array; no distinction is made between pushed back strings and input characters. Whenever NEXT exceeds LAST, GETLINE is called to read a new line and reset the pointers.

The lexical analyzer, SCAN, is based on a finite automaton and replaces GTOK in the old organization. More information about the method used can be found in [3]. Many syntactic details like recognizing continuation operators, skipping comments, and handling double operators (eg. >=) formerly distributed into many other routines are done in SCAN. Although this makes the code slightly longer, we feel that it is as straightforward and easy to understand as the original version. The finite automaton from which SCAN was derived has 23 states arranged in a very simple structure. Most of the states are used to classify each token -- a classification that remains intact up to the parser. In many cases, the classification scheme eliminates the need to continually push back strings.

The new GETTOK replaces DEFTOK and LEX. A single hash table is used for both macro names and language keywords. Each alphanumeric token returned by SCAN is further classified by GETTOK as one of the language keywords, a macro call, or a simple alpha token. Macro calls are processed immediately by pushing back the definition and calling SCAN again. While the definition of new macros and the inclusion of files were processed in routines lower than the parser in the original RATFOR, these features have been promoted to the statement level in MOUSE4. Thus, they are handled in routines called by the parser rather than in GETTOK. While this forbids the (excessive) generality of allowing macro definitions in the middle of a statement, it improves the

simplicity and efficiency. It appears that the original motivation for having separate file inclusion and macro processing routines was that RATFOR was implemented under UNIX [4] which provides these facilities independently. Since a macro processor is discussed independent of RATFOR in [2], it became natural to use the routines the way they had been developed. We assert that they make more sense as statements.

5 Improvements in Efficiency:

Table 1 shows typical CPU times for FORTRAN, RATFOR, "optimized" RATFOR, and MOUSE4. It should be pointed out that the ad hoc parser and code generation routines from RATFOR are included in MOUSE4 with little or no change. While some minor improvements in running time might have been made by rewriting them, we felt certain that the major inefficiencies occur in the lexical routines of RATFOR. The timings confirmed our belief.

A profile of MOUSE4 shows that it spends about 80% of its time in getting lines, outputting strings and in the system I/O routines. LOOKUP accounts for 4% of the CPU usage, even with a large number of defines in the table. In all cases, we find that MOUSE4 is competitive with the standard production software on our system.

6 Conclusions:

Often basic algorithmic changes have a much stronger effect on efficiency than local improvements in the code. We have found an example of this phenomenon in our attempt to improve the efficiency of RATFOR. A profile of CPU times revealed a significant bottleneck in the code, a linear table search, that was recoded to increase the efficiency. Even with a binary search, the CPU usage remained too high for our production environment. A reorganization of the lexical routines was required to reduce CPU costs to the level of other system software. We feel that we have retained the simplicity and

modularity of RATFOR while reducing the running time.

It is often tempting to believe that a simple minded approach to a program will suffice. We have demonstrated a case where a linear table search was not adequate for production software, despite the claims of the designer. Moreover, it would appear that formal methods for lexical analysis have distinct advantages over the ad hoc techniques of good programmers even for simple programs such as preprocessors.

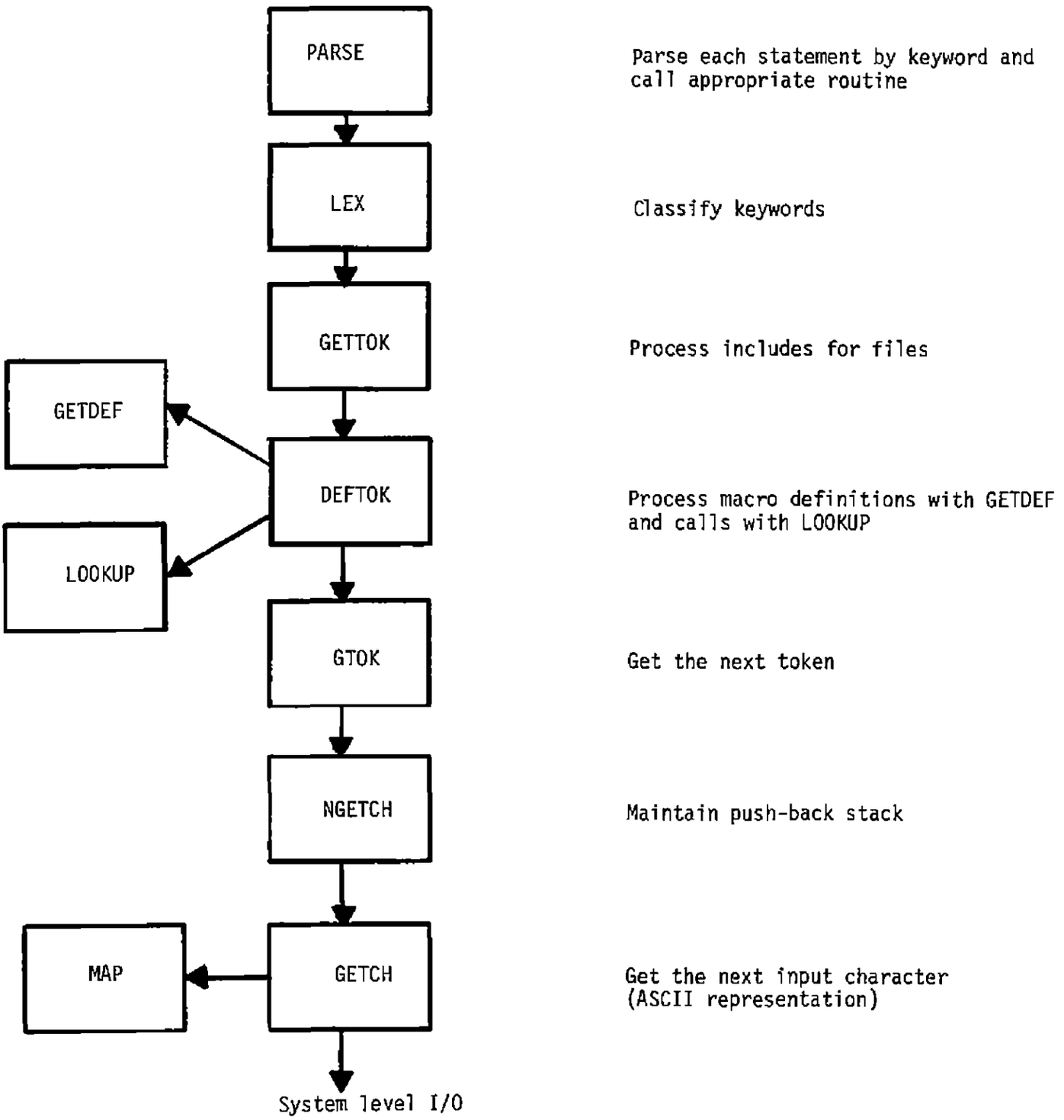
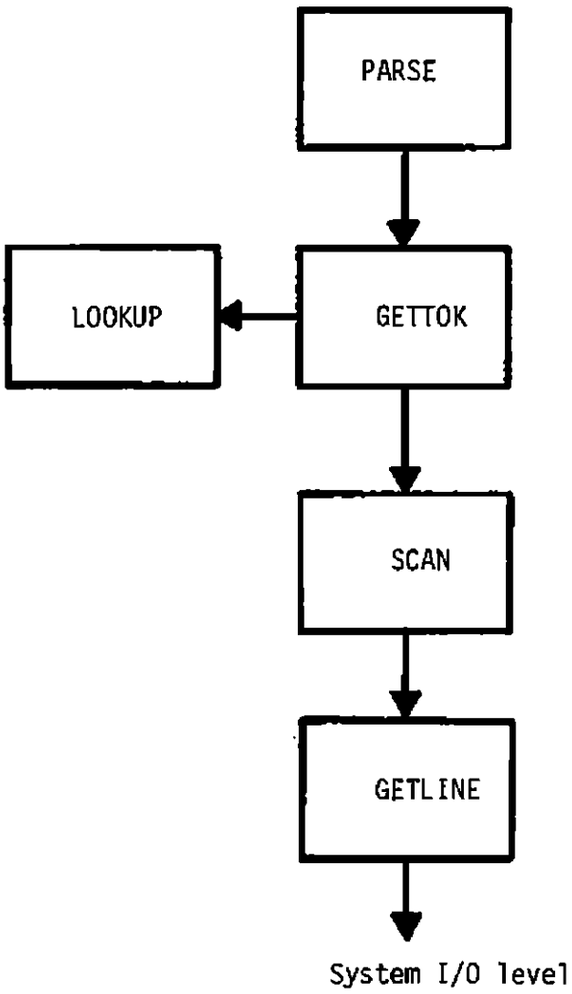


Figure 1. The organization of the lexical parts of RATFOR



Parse each statement by token type, calling appropriate routine

Look up alphanumeric tokens and classify as keyword, macro call, or other

Scan and extract the next token, classifying all but alphanumeric tokens

Get the next input line

Figure 2. The organization of the lexical parts of MOUSE4

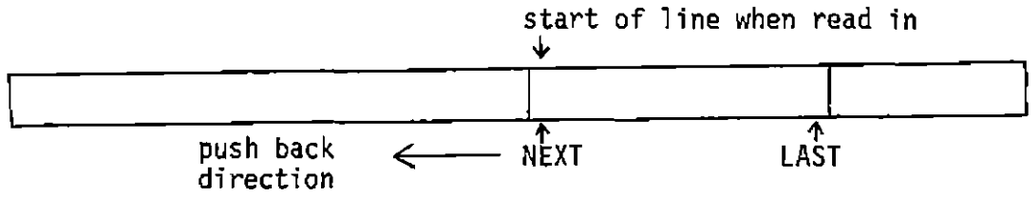


Figure 3. The next character array with a line read into the right-hand end. NEXT and LAST point to the next and last character, respectively.

	RATFOR	FORTRAN	"optimized" RATFOR	MOUSE4
600 line (225 statements)	12.620	1.332	7.381	1.866
3000 line (2050 ")	185.470	9.722	70.157	12.723

Table 1. CPU times for MNF FORTRAN and various versions of RATFOR taken from actual runs on a CDC 6500. The FORTRAN times are for the program after RATFOR had removed blanks and comments.

References

[1] Kernighan, B., "RATFOR a Preprocessor for Rational FORTRAN," Software Practice and Experience, vol. 5:4 (Oct/Dec) 1975, pp. 395-406.

[2] Kernighan, B. and Plauger, P., Software Tools, Addison Wesley, 1976.

[3] Gries, D., Compiler Construction For Digital Computers, Wiley, 1971.

[4] Ritchie, D. and Thompson, K., "The UNIX Time Sharing System," CACM, vol 17:7 (July) 1974, pp. 365-375.