THE FSCAN LEXICAL ANALYZER GENERATING SYSTEM

by

Geoffrey Clemm
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-128-78 June, 1978

INTERIM TECHNICAL REPORT

U.S. ARMY RESEARCH OFFICE

CONTRACT NO. DAAG29-78-G-0046

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CU-CS-128-78

2. GOVT ACCESSION NO.:

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): "The FSCAN Lexical Analyzer Generating System"

5. TYPE OF REPORT & PERIOD COVERED:

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Geoffrey Clemm

8. CONTRACT OR GRANT NUMBER(s): DAAG29-78-G-0046
MCS77-02194

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Dept. of Computer Science
Univ. of Colorado at Boulder
Boulder, Colorado 80309

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
U. S. Army Research Office
Post Office Box 12211
Research Triangle Park, NC 27709

12. REPORT DATE: June, 1978

13. NUMBER OF PAGES: 14

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: NA

16. DISTRIBUTION STATEMENT (of this Report):

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

NA

18. SUPPLEMENTARY NOTES:

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):

lexical analysis, FORTRAN scanner, FORTRAN grammar

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

There has recently been much interest in the development of software validation tools for FORTRAN. Such tools are usually designed to analyze programs written in ANSI standard FORTRAN. However, because there are many dialects and extensions of FORTRAN in use, it would be desirable to analyze these as well. One solution is to develop a single diagnostic tool for standard FORTRAN which may be easily modified to accept variants of the language. Since most of the variations occur at lexical and syntactic levels, the design of a flexible

Abstract

There has recently been much interest in the development of software validation tools for FORTRAN. Such tools are usually designed to analyze programs written in ANSI standard FORTRAN. However, because there are many dialects and extensions of FORTRAN in use, it would be desirable to analyze these as well. One solution is to develop a single diagnostic tool for standard FORTRAN which may be easily modified to accept variants of the language. Since most of the variations occur at lexical and syntactic levels, the design of a flexible lexical analyzer is a key issue. The FSCAN Lexical Analyzer Generating System has been designed with this purpose in mind. This report describes the FSCAN language, a compiler for the language, and an interpreter for the resulting object code. An example of a complete FSCAN program is included.

I. INTRODUCTION

The first phase of the analysis of a computer program written in some programming language is "lexical analysis" or "scanning", where the source text is broken up into the words or "tokens" of the programming language. For most languages this is a relatively straightforward task, as spaces or some other delimiter is required at any token separation points that could be ambiguous. Unfortunately the ANSI FORTRAN standard specifies that spaces for the most part are meaningless in FORTRAN programs [1]. This creates several ambiguous situations that cannot be resolved without backtracking by a left to right scan with single character lookahead of the source text. For example, if the string 'DO' has been read, it is unclear whether the scan has reached the end of the keyword, 'DO', in a statement such as

```
DO 1Ø I = 1, 3
```

or whether the scan is in the middle of a variable name in a statement such as

```
DO1 = 5 * X
```

The problem of the lexical analysis of FORTRAN is further complicated by the existence of numerous dialects and extensions of FORTRAN that vary according to the installation and particular compiler in use. The problem is therefore most acute for a system such as the DAVE software validation system [2] where it is desirable that all variants of FORTRAN be readable. Ordinarily this would entail recoding the lexical analyzer module for each new FORTRAN variant, in addition to maintaining a library of already coded lexical analyzer modules.

To minimize these tasks, the FSCAN (Fortran SCANner) Lexical Analyzer Generating System was developed. The FSCAN system consists of a language, a compiler for the language, and an interpreter for the object code produced by the FSCAN compiler.

II. THE LANGUAGE

The FSCAN language (henceforth referred to simply as "FSCAN") was designed to allow the specification of a complex lexical analyzer, such as that required by FORTRAN, in as concise and understandable a manner as possible.

A. Programs and Procedures

An FSCAN program consists of a single FSCAN procedure (within which may be defined additional procedures). An FSCAN procedure specifies in an extended BNF-style notation a grammar that describes a left to right pass over the source text. Within the grammar, actions such as the generation of a token are indicated.

Syntax

An FSCAN procedure consists of a sequence of grammatical rules which are delimited by the keywords, 'SCANNER' and 'END'. Following each of these keywords is the goal symbol for the sequence of rules; this also serves as the name of the procedure. The redundant repetition of the goal symbol is used by the FSCAN compiler to ensure that the 'SCANNER' - 'END' pairs are matched in the way the programmer intended.

Example

```
SCANNER DIG:
    rule 1; rule 2; ...; rule n;
    END DIG
```

Semantics

The rule indicated by the goal symbol of a procedure specifies an LR(1) parse of the source text which is performed when the procedure is called. The parse is performed in a longest match manner; namely, given the choice between finishing and parsing more of the course text, the procedure will always continue parsing.

B. Rules

An FSCAN rule is either a macro rule, a variable defining rule, or a procedure rule. The scope of rule definitions corresponds to that of ALGOL.

1. Macro Rules

As in a BNF rule, the left side of a macro rule is a nonterminal while the right side is a sequence of alternatives. The extensions of FSCAN are that each alternative may optionally have an associated action, and that an alternative, rather than being simply a sequence of terminals and nonterminals, may contain any of a variety of regular expression style operators as well as parentheses for grouping.

Syntax

Each alternative is preceded by a single-right-arrow (→). The optional action is placed at the end of the corresponding alternative and is preceded by a double-right-arrow (⇒).

Example

```
TEXT → fscan_reg_exprn 1 ⇒ action 1
     → fscan_reg_exprn 2
     → fscan_reg_exprn 3 ⇒ action 2
```

Semantics

A macro rule is a standard macro in that the right part of the rule textually replaces any occurrence of the nonterminal of the left part, when the occurrence is in an FSCAN regular expression within the scope of the macro rule definition. A macro rule cannot be recursively defined. Thus in the above example, the nonterminal, TEXT, could not appear in any of the three FSCAN regular expressions in the right part. During execution when any of the alternatives have successfully been matched with the source text, the corresponding action, if any, is performed. The compiler ensures at compile time that during execution of the object code it is determinable which action, if any, is to be performed by examining the next source text character only.

2. Variable Defining Rules

A variable defining rule is similar in form to a macro rule except that the right side is restricted to being a single alternative. The nonterminal on the left side names the variable being defined, in addition to naming the regular expression on the right side, as in a macro rule.

Syntax

The single alternative is preceded by an equal sign (=).

Example

```
HCONST = fscan_reg_exprn
```

Semantics

A variable is used to convey numeric information from the source text to the FSCAN program. Its semantics correspond to those of a macro rule except that an implicit "evaluation-action" is attached to the single alternative of the right part. When executed this action evaluates the string processed by the right side of the variable defining rule.

The number produced is stored as the value of the variable defined by that rule. The variable can then be used in FSCAN contexts where integers are expected, in which case no macro substitution occurs, but rather, during execution the integer value is that produced by the most recent execution of that variable's execution action. The compiler ensures that it is always possible to derive an integer from strings matched by the right part of a variable defining rule.

3. Procedure Rule

A procedure rule is simply an FSCAN procedure, see II. A.

C. FSCAN Regular Expressions (abbreviation : FRE)

1. Atomic units

The atomic units of an FRE are terminals, nonterminals, and integers.

a. Terminals

Syntax

A terminal is either a "kept-string" or a "deleted string". A kept-string is a sequence of characters enclosed in double quotes (") while a deleted string is a sequence of characters enclosed in single quotes ('). If a sharp (#) appears in the string, the sharp is ignored and the next character is treated as the next character of the string, even if that character is a double-quote, single-quote, or a sharp. For terminals the strings are restricted to be of length one.

Examples

'A' ";" '##' "#""

Semantics

The character of the terminal is compared with the next character of the source text. If they match, the source text character is marked as "kept" or "deleted", depending on whether the terminal is a kept-string or a deleted-string. The FSCAN compiler will indicate * if it is ever possible for a given FSCAN program to mark a source text character simultaneously as "kept" and "deleted". *(with an appropriate error message at compile time)

b. Integers

Syntax

An integer is a string of digits.

Examples

53 0 05

Semantics

Integers have their usual meaning.

c. Nonterminals

Syntax

A nonterminal is a sequence of letters and digits, the first of which is a letter, that is terminated by a character that is neither a letter nor a digit.

Examples

A TEMP TEMP1 B3B

Semantics

Nonterminals can name macro rules, variables, or procedure rules. As mentioned earlier, macro rule names are textually replaced by the right part of the macro defining rule, for which the semantics have been described. The semantics of variable names vary according to their context. If a variable is used where an integer is expected, the current value of the variable is used during execution; otherwise, the right part of the variable definition (with implicit associated "evaluation action") textually replaces the use of the variable name. When the non terminal names a procedure, the appropriate procedure is called during execution. The compiler ensures at compile time that at any point in execution, it is determinable from the character presently being examined, whether to invoke a procedure, and which one to invoke.

2. Operations

The operations from which FSCAN regular expressions are composed can be divided into two types; basic operations, and extended operations that can be defined in terms of the basic operations. Let A, B, C be FRE's and let N be a variable or integer.

a. Basic Operations

Syntax

Alternation : A | B | C | . . .

Concatenation : A B C . . .

Repetition : A*

Negation : NOT A

Example

NOT (","|";"|"?") ('x'*)

Semantics

An alternation successfully matches the source text if any of its alternates does. A concatenation matches the source text if its operands sequentially match the source text. A repetition matches an arbitrary number (possibly zero) of its operand with the source text. The operand of a negation is restricted to regular expressions that specify a set of characters, all of which are kept-strings or all of which are deleted strings. A negation then matches any character that is not in its operand's character set. If matched, a source character is marked as "kept" or "deleted" if the operand character set consists of kept-strings or deleted-strings, respectively.

b. Extended Operations

Syntax

```
      +  :  A  +                          ≡  A (A*)
      ?  :  A  ?                          ≡  A 1()
   LIST  :  A  LIST B                     ≡  A (B A)*
   ELSE  :  A ELSE  B  ELSE  C  ELSE ...  ≡  A | B | C | ...
     **  :  A ** N                        ≡  A A A ... A  (N times)
     ?*  :  A ?* N                        ≡  A? A? A? .. A?  (N times)
```

Restrictions: The operands of ELSE and the first operands of ** and ?* are restricted to being the names of procedures.

Semantics

The semantics of the extended operations are largely determined by those of the basic operations by which they are defined. In addition, though,the ELSE construct provides a "backup and restore" feature where if the first operand fails to successfully match the source text, the second operand is tried, etc. Also the ?* operator provides limited backup in the sense that, if less than N A's have been successfully matched, the parse is backed up to the state at which the last A (possibly no A's) has been successfully matched.

c. Actions

Syntax

Actions are either kept-strings, deleted-strings, integers, or nonterminals.

Examples

```
"INT"  'REAL'  8  203  CARDS  RESCAN
```

Semantics

A string or an integer indicates that a token is to be output. For a string, the type of the token output is indicated by a unique integer associated at compile time with that string; for an integer, the type of token is indicated during execution by outputting the value (e.g., "8" or "203") of the integer action. Also output during execution is the sequence of kept characters that were matched by the alternative corresponding to that action being performed. Actions that are deleted-strings indicate that their corresponding alternatives only mark characters as deleted, and thus it is sufficient to simply generate the token type when the action is performed.
(Note: A program cannot contain both integer and string actions.) A nonterminal action indicates that the sequence of kept characters matched by that action's alternative is to be rescanned by the FSCAN procedure named by the nonterminal. This process of rescanning is sometimes referred to as "screening".

III. EXAMPLE OF A COMPLETE FSCAN PROGRAM

This FSCAN program specifies the scanner used by the FSCAN compiler, i.e., it performs the lexical analysis of an FSCAN program.

```
SCANNER FSCAN :
   FSCAN -> (' '* (KEYWORD ELSE NAME / INTEGER / KSTRING / DSTRING /
                   DELIMITER / OPERATOR / COMMENT) )* ;
   SCANNER KEYWORD :
      KEYWORD -> KEYWD NOTACHAR**0 ;
      KEYWD -> 'S' 'C' 'A' 'N' 'N' 'E' 'R' => 8
            -> 'E' 'N' 'D' => 7
            -> 'E' 'L' 'S' 'E' => 9
            -> 'L' 'I' 'S' 'T' => 14
            -> 'N' 'O' 'T' => 19 ;  END KEYWORD ;
   SCANNER NAME :
      NAME -> KACHAR (KACHAR / KDIGIT)* => 20 ; END NAME ;
   INTEGER -> KDIGIT+ NOTD**0 => 21 ;
   KSTRING -> DQ (NOTDQSH / SHARP KC)* DQ => 22 ;
   DSTRING -> SQ (NOTSQSH / SHARP KC)* SQ => 23 ;
   DELIMITER -> ':' => 2
             -> ';' => 3
             -> '(' => 5
             -> ')' => 6 ;
   OPERATOR -> '-' '>' => 4
            -> '/' => 10
            -> '=' NOTRAB**0 => 11
            -> '=' '>' => 12
            -> '?' NOTAST**0 => 13
            -> '*' '*' => 15
            -> '?' '*' => 16
            -> '*' NOTAST**0 => 17
            -> '+' => 18 ;
   COMMENT -> SHARP (NOT SHARP)* SHARP ;
   KACHAR -> "A"/"B"/"C"/"D"/"E"/"F"/"G"/"H"/"I"/"J"/"K"/"L"/"M"/
             "N"/"O"/"P"/"Q"/"R"/"S"/"T"/"U"/"V"/"W"/"X"/"Y"/"Z" ;
   SCANNER NOTACHAR :
      NOTACHAR -> NOT ('A'/'B'/'C'/'D'/'E'/'F'/'G'/'H'/'I'/'J'/'K'/'L'/'M'/
          'N'/'O'/'P'/'Q'/'R'/'S'/'T'/'U'/'V'/'W'/'X'/'Y'/'Z') ; END NOTACHAR ;
   KDIGIT -> "0"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9" ;
   SCANNER NOTD :
      NOTD -> NOT KDIGIT ; END NOTD ;
   DQ -> '"' ; SQ -> '#'' ; SHARP -> '##' ;
   NOTDQSH -> NOT("##"/"#"") ; NOTSQSH -> NOT("##"/"'") ;
   SCANNER NOTRAB :
      NOTRAB -> NOT '>' ; END NOTRAB ;
   SCANNER NOTAST :
      NOTAST -> NOT '*' ; END NOTAST ;
   KC -> (NOT" ")/" " ;
   END FSCAN
```

LINE NUMBER	INTERPRETATION
1	The top level procedure, and therefore the program, is called FSCAN.
2	The scanner accepts a sequence of KEYWORD's, NAME's, INTEGER's, etc., each of which can be preceded by an arbitrary number of spaces. As KEYWORD's and NAME's cannot be differentiated by an SLR (1) process, the ELSE operation must be used to allow the acceptance of either. Note that KEYWORD's being the first

LINE NUMBER	INTERPRETATION
2 (cont'd)	operand of ELSE will cause a string that could be accepted as either a KEYWORD or a NAME, to be accepted as a KEYWORD.
3	A KEYWORD is a KEYWD followed by some nonalphabetic (NACHAR) character. Note that the exponent of zero indicates that although it is checked that the following character is nonalphabetic, no (zero) nonalphbetic characters are actually processed at this point. In case the following character were alphabetic, the KEYWORD scanner would fail, and the alternative, NAME, would be invoked at the point in the input where the KEYWORD scanner had been initiated.
4	KEYWD will accept, and mark as deleted, the strings "SCANNER:, "END", "ELSE", "LIST", and "NOT", and will output tokens numbered 8, 7, 9, 14, and 19, respectively.
5	NAME will accept, and mark as kept, an alphabetic character followed by an arbitrary number of alphanumeric characters. Token number 20 will then be output, as well as the sequence of kept characters marked by NAME.

The rest of the program is interpreted in an analogous fashion. It thus provides a rigorous and complete specification of the lexical analysis of FSCAN programs. The abbreviated non-terminals are to be read as follows:

KSTRING	:	kept string
DSTRING	:	deleted string
NOTACHAR	:	not an alphabetic character
KACHAR	:	kept alphabetic character
KDIGIT	:	kept digit
NOTD	:	not a digit
DQ	:	double quote
NOTDQSH	:	not a double quote or a sharp
KC	:	kept character
SQ	:	single quote

NOTSQSH : not a single quote or a sharp
NOTRAB : not a right angle bracket
NOTAST : not an asterisk

With this information, the sample FSCAN program also provides a structured and understandable description for a human reader.

IV. PROGRAMMING HINTS FOR FSCAN

Much of FSCAN programming is similar to writing a grammar for some parser generator. The regular expression-style operators are, for the most part, straightforward extensions. The distinction, though, between a procedure and a macro-rule, i.e., "SCANNER A : A → B C, END A" vs. "A → B C" does not correspond to any grammatical concepts, but rather to the normal programming language concepts of a procedure and a macro. In particular, a macro (when used in more than one place) causes a larger object program to be generated (as a copy of the macro is inserted at each use of the macro) while a procedure executes more slowly (due to the overhead of the procedure call and return). An additional distinction that is important for programming is that while only one procedure can be executing at any particular time, several macro rules can conceptually be executing in parallel.

The "ELSE" operator involves considerably more overhead than the "|" operator in that the state of the scanner must be saved so that it can be restored in case a particular alternative of the "ELSE" operator fails, implying the next alternative must be tried. In contrast, the "|" operator conceptually applies all of its alternatives in parallel. Thus whenever possible, the "|" operator should be used for the sake of efficiency.

The "**" and "?*" operators are conceptually straightforward, except possibly for the following two characteristics. First, "A**Ø" indicates that the next character in the input is checked for a match with a legal first character of A, but that A does not actually process any characters, due to the exponent of Ø. Second, the "?*" operator involves the same overhead as the "ELSE" operator, since "A ?* 5" must have the ability to back up to the state of the scanner after the third A was accepted, in case the entire fourth A could not be matched.

V. IMPLEMENTATION DETAILS

Compiler. The complete FSCAN compiler runs in 36,000 (decimal) words on a CDC 6400 machine. The compile time for an FSCAN program for ANSI FORTRAN is 28 seconds. The size of the object code (tables) produced for this scanner is 1400 (decimal) words.

The compiler is written in machine-independent standard ANSI FORTRAN, with the following exceptions:

1. Certain non-standard functions are assumed:
   a. IAND (A,B), IOR (A,B), INOT (A)
      These should return the respective bitwise logical operation on their arguments
   b. LRS (A,I), LLS (A,I)
      These should return the logical binary right and left shift, respectively, of the argument A by the integer amount I, with zero fill.
   c. INTGER (A)
      The argument A is a character stored in 1H or A1 format (assumed equivalent). The result is an integer such that:
      1. $0 <$ INTGER (A) $\leq$ # distinct characters
      2. INTGER (A) = INTGER (B) iff A is the same char. as B
      3. INTGER (1Hx) - INTGER (1Hy) = x-y if x, y are digits, i.e., INTGER (1H7) - INTGER (1H3) = 4
   d. ENDFIL (I)
      This returns true iff logical unit I is at end of file.
2. It is assumed that the # characters $\leq 2 \cdot$(# bits in a word). If this is not the case, the bit vector module would have to be altered to represent a vector as more than 2 words.
   Note: This machine dependency is being replaced by the requirement that on importation to a new machine, the constants NMBIT'S and NMCHRS be initialized to correspond to the new machine. For CDC, the initialization is
   DATA NMBITS/60/, NMCHRS/64/

Interpreter. The object code for the interpreter consists of 3000

words excluding the space required for the tables. The interpreter source code is machine-independent standard ANSI FORTRAN, with the same exceptions found for the compiler.

The tables output by the compiler are in the form of a BLOCK DATA ANSI FORTRAN subprogram which is to be compiled and loaded with the object code for the interpreter.

Note: For maximal time efficiency, the routines IN and ADVANC should be replaced by equivalent optimized machine-language routines.

The source for the compiler is on file COMPIL.
The compiler is called by: CB, input, listing, errors, tables.
The source for the interpreter is on file NTD.
The interpreter is called by: NTDB, input, listing, errors.
The FSCAN program for ANSI FORTRAN is on the file SCAN.
The file SCANT was produced by a CB, SCAN, listing, errors, SCANT run.
(NTDB is produced from the compilation of SCANT and NTD)
To use the scanner, NTDB,

for each desired token, call SCANNER;

the token will be returned in /TOKENC/,

where TKNTYP is the type of the token
TKNCHR (30) is an array of A1 characters (the sub-rosa info)
where TKNCHR (1) ... TKNCHR (ITKNCH) are the characters
TOKERR is a logical flag which is true iff the token
contains an error

Appendix

Syntax of FSCAN programs

```
SCANNER
   → 'SCANNER' GOAL_SYMBOL ':' (RULE ';') + 'END' GOAL_SYMBOL;
RULE
   → NONTERMINAL ('→' REG_EXPRN('⇒ ' ACTION)?)+
   → VARIABLE '=' REG_EXPRN
   → SCANNER ;
REG_EXPN → REG_TERM list '|',
REG_TERM → REG_FACTOR + ;
REG_FACTOR
   → REG_PRIMARY ('*'|'+'|'?')?
   → 'NOT' REG_PRIMARY
   → REG_PRIMARY 'LIST' REG_PRIMARY;
REG_PRIMARY
   → '(' REG_EXPRN ? ')'
   → NONTERMINAL list 'ELSE'
   → NONTERMINAL ('**'|'?*') EXPONENT
   → TERMINAL ;
ACTION → SCREENER | TERMINAL | '<INTEGER>' ;
EXPONENT → VARIABLE | '<INTEGER>' ;
GOAL_SYMBOL → '<NAME>' ;
NONTERMINAL → '<NAME>' ;
VARIABLE → '<NAME>' ;
SCREENER → '<NAME>' ;
TERMINAL  → '<KEPT_STRING>' | '<DELETED_STRING>' ;
```

Note: "A?" is equivalent to "(A|ε)"
"A list B" is equivalent to A(B A)*

VI. Bibliography

[1] ANSI : FORTRAN. X3.9-1966, American National Standards Institute 1966.

[2] Osterweil, L. J.; and Fosdick, L. D. "DAVE - a validation, error detection and documentation system for FORTRAN programs," Software Practice and Experience.