FSCAN Report and User's Manual

by

Geoffrey M. Clemm
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-166-79 November, 1979

INTERIM TECHNICAL REPORT

U.S. ARMY RESEARCH OFFICE

CONTRACT NO. DAAG29-78-G-0046

We acknowledge U. S. Army Research support under contract no. DAAG29-78-G-0046 and National Science Foundation support under grant no. MCS77-02194.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CU-CS-166-79	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* FSCAN Report and User's Manual		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		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 November 1979
		13. NUMBER OF PAGES 34
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, parsers, parsing techniques, compilers, static program analysis

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

FSCAN is a language for specifying the lexical analysis of programs written in any current programming language, including FORTRAN. This report describes the FSCAN language, a compiler for the language, and an interpreter for the resulting object code. The interpreted object code forms an efficient lexical analyzer that takes as input a stream of characters and produces as output a stream of tokens (lexical units). The compiler and interpreter are designed for portability. Both are written in ANS FORTRAN (1966) supplemented by a small number of short machine dependent subroutines.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Abstract

FSCAN is a language for specifying the lexical analysis of programs written in any current programming language, including FORTRAN. This report describes the FSCAN language, a compiler for the language, and an interpreter for the resulting object code. The interpreted object code forms an efficient lexical analyzer that takes as input a stream of characters and produces as output a stream of tokens (lexical units). The compiler and interpreter are designed for portability. Both are written in ANS FORTRAN (1966) supplemented by a small number of short machine dependent subroutines.

CONTENTS

1. INTRODUCTION

The first phase of the analysis of a computer program 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 are 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 without backtracking be resolved by a left-to-right scan with single character look-ahead of the source test. 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

```
DO1ØI = 1 + 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 <u>all</u> 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. The FSCAN language and the LR style processing were initially specified by DeRemer [3].

† In this report we refer to the 1966 standard.

2. THE LANGUAGE

The FSCAN language (henceforth referred to as 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.

2.1 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.

Since it was not considered useful to allow a lexical analyzer to quit before reaching the end-of-file of the input stream, or to allow it to continue operating beyond the end-of-file, the writer of an FSCAN program is not allowed to reference the end-of-file. Instead, the procedure that is the FSCAN program, i.e.,

```
SCANNER LEXANLYZ :
    LEXANLYZ → ...
    END LEXANLYZ.
```

is conceptually imbedded in the following context:

```
LEXANLYZ* EOF
```

where EOF is defined to be:

```
EOF → 'end-of-file'  ⇒ 'EOFTOK';
```

and where 'end-of-file' matches the logical end-of-file of the input stream. EOFTOK is therefore predefined in all FSCAN programs to be the action (see 2.4) for recognizing end-of-file in the input stream.

2.1.1 Syntax

An FSCAN procedure or "scanner" consists of a sequence of grammatical rules 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.

2.1.2 <u>Example</u>

```
SCANNER DIG:
   rule_1; rule_1; ...; rule_n;
   END DIG
```

2.1.3 <u>Semantics</u>

The rule indicated by the goal symbol of a procedure specifies a finite-state stack-automaton 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 source text, the procedure will always continue parsing.

2.2 <u>Rules</u>

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.

2.2.1 <u>Macro Rules</u>

As in a BNF rule, the left side of a macro rule is a nonterminal while the right side is a sequence of alternatives. Each alternative may have an associated action, and an alternative, rather than being only a sequence of terminals and nonterminals, may contain any of a variety of operators, in the style of regular expressions, as well as parentheses for grouping.

2.2.1.1 <u>Syntax</u>

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 (=>).

2.2.1.2 <u>Example</u>

```
TEXT → fscan_reg_exprn  => action
     → fscan_reg_exprn
     → fscan_reg_exprn  => action
```

2.2.1.3 <u>Semantics</u>

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 except through a procedure rule call. Thus in the above example, the nonterminal, TEXT, could not appear in any of the three FSCAN regular expressions in the right part, but the following construction would be legal:

```
    TEXT1 → fscan_reg_exprn_containing_TEXT2;
    SCANNER TEXT2:
        TEXT2 → fscan_reg_exprn_containing_TEXT1;
        END TEXT2;
```

This is legal since execution time recursion is implemented, whereas recursively defined macros without intervening procedure rule calls would imply infinite textual expansion of the macro.

During execution of the interpreter, when an alternative has been matched with the source text, the corresponding action, if any, is performed. The compiler ensures at compile time that it is determinable which action is to be performed by one character look-ahead only.

2.2.2 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.

2.2.2.1 Syntax

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

2.2.2.2 Example

```
NUM = fscan_reg_exprn;
TEXT → NUM fscan_scanner ** NUM;
```

2.2.2.3 Semantics

A variable is used to convey numeric information from the source text to the FSCAN program. This feature is required to allow processing of FORTRAN hollerith constants; e.g., 3HABC. Its semantics correspond to those of a macro rule except that an implicit action is attached to the single alternative of the right part. This action evaluates, as an integer, the string matched by the right side of the variable defining rule. The integer 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. During execution of the interpreter 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.

2.2.3 Procedure Rule

A procedure rule is simply an FSCAN procedure, see 2.1.

2.3 FSCAN Regular Expressions (abbreviation: FRE)

2.3.1 Atomic units

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

2.3.1.1 Terminals

2.3.1.1.1 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 zero, length one, or the string of length three, eol. A length zero string matches no character, a length one string matches the character of that string, and eol represents the end-of-line character.

2.3.1.1.2 Examples

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

2.3.1.1.3 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, with an appropriate error message at compile time, if it is possible for a given FSCAN program simultaneously to mark a source text character as "kept" and "deleted."

2.3.1.2 Integers

2.3.1.2.1 Syntax

An integer is a string of digits.

2.3.1.2.2 Examples

54 0 05 1234567890

2.3.1.2.3 Semantics

Integers have their usual meaning.

2.3.1.3 Nonterminals

2.3.1.3.1 Syntax

A nonterminal is a sequence of letters and digits, the first of which is a letter.

2.3.1.3.2 Examples

A TEMP TEMP1 B3B

2.3.1.3.3 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 nonterminal 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.

2.3.2 Operations

The operations used to compose FSCAN regular expressions are divided into two types: basic operations and extended operations. Let A, B, C be FRE's and let n be an integer ≥ 0 or a variable.

2.3.2.1 Basic Operations

2.3.2.1.1 Syntax

Alternation	:	A | B | C | . . .
Concatenation	:	A B C . . .
Repetition	:	A*
Negation	:	NOT A

2.3.2.1.2 Example

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

2.3.2.1.3 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.

2.3.2.2 Extended Operations

2.3.2.2.1 Syntax

+	:	A +	≡ A (A*)
?	:	A ?	≡ A|()
LIST	:	A LIST B	≡ A (BA)*
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.

2.3.2.2.2 Semantics

The semantics of the extended operations are largely determined by those of the basic operations by which they are defined. The operators, ELSE, **, and ?*, are only approximately equivalent to their respective syntactic expansions, because they possess the following additional properties:

2.3.2.2.2.1 ELSE

The ELSE construct provides a backtrack feature where if the first operand fails to successfully match the source text, the second operand is tried, etc. Once the final operand is invoked, match failure will cause standard error recovery, rather than the backtract feature.

2.3.2.2.2.2 **

The only distinction between ** and its syntactic expansion occurs when the exponent, n, is zero. In this case A**∅ matches the input stream only if A would match the next character in the input stream. Since the exponent is ∅, no characters are actually matched by A, only the check is performed.

2.3.2.2.2.3 ?*

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.

2.4 Actions

2.4.1 Syntax

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

2.4.1.1 Examples

"INT" 'REAL' CARDS RESCAN

2.4.2 Semantics

A string indicates that a token is to be output. The type of the token output is indicated by a unique integer associated at compile time with that string. A deleted-string action and a kept-string action indicate that a deleted-token and a kept-token respectively are to be output. For a kept-token, the sequence of kept characters that were matched are output in addition to the token type.

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."

2.5 Comments

Comments can be included at any point within an FSCAN program except within atomic units, keywords, or operators. A comment begins with a sharp (#) and terminates at the first end-of-line.

3. THE COMPILER

The FSCAN compiler consists of 4000 lines of standard ANSI FORTRAN code. In addition there are two tables (both the parser and scanner for the FSCAN compiler are table driven) that would require straightforward and mechanical modification on machines with small wordsizes. (See appendices C and D). Finally, there is a group of short (1 to 5 lines) routines that are machine dependent. (See appendix A).

On the CDC 6400 machine the complete FSCAN compiler requires 130 K octal words to process the FSCAN program that recognizes an extended FORTRAN variant.

The FSCAN compiler contains eight processing modules that perform the following tasks:

3.1 Lexical Analysis, Syntactic Analysis, and Tree Construction

The input is read and all syntactic errors are reported. If the input is syntactically correct, a parse tree corresponding to the input grammar is built, otherwise processing stops after the entire input has been scanned for syntactic correctness.

3.2 Symbol Identification

Each applied occurrence of a symbol (i.e., in the right sides of rules) is associated with its defining occurrence (i.e., the rule in which that symbol was defined). In addition the following errors are detected and reported:

3.2.1 A scanner's beginning goal symbol is different from its ending goal symbol (probably due to improper scanner nesting that could not be detected by the parser).

3.2.2 A nonterminal is defined by two different rules within the same scanner.

3.2.3 No rule defines the goal symbol of a scanner.

3.2.4 A variable exponent is defined in something other than a variable defining rule.

3.2.5 A symbol is used that has not been defined by any rule.

3.2.6 A symbol that is an alternative of an ELSE, a screening action, or the base of ** or *?, is defined in something other than a procedure rule.

If any of the above errors occur, processing is halted following the completion of the symbol identification phase.

3.3 Character Set Creation

The terminals are converted to a set containing the appropriate character and, where feasible, set operations corresponding to FSCAN operators are performed (i.e., '|' and 'NOT') and the operator node is replaced by the resulting set. In addition, by propagating attribute vectors down and then back up the tree, the following errors are detected and reported:

3.3.1 A macro rule is recursively defined.

3.3.2 A variable exponent is used before the variable could have received a value.

3.3.3 A 'NOT' operator is applied to something other than a character set.

3.3.4 A terminal string other than 'EOL' consists of more than one character.

3.3.5 A rule containing a kept character is used in a context where the kept character is associated with no token.

3.3.6 A rule generating a token is used in a context where another token is currently being built.

3.3.7 A rule containing untokenized kept characters and a rule producing tokens appear in the same context (either error-3.3.5 or error 3.3.6).

3.3.8 Non-digit characters are kept by a variable definition rule (or by a rule used by a variable definition rule).

If any of the above errors occur, processing is halted following the completion of the character set creation phase.

3.4 Tree Threading

The tree is converted to a directed acyclic graph by the addition of directed edges. This additional linkage allows the LR processing to be performed efficiently.

3.5 Code Generation

The code for a parser that will accept the user's grammar is generated. This code is written out to a scratch file as it is produced.

3.6 <u>Code Verification</u>

The parse tree is purged and the code from the scratch file is read into memory. It is then verified that the code specifies a deterministic machine that will halt on finite input. If the grammar specified nondeterministic or non-halting behavior, this is reported as an error, and processing will halt following completion of the code verification phase. A nondeterminism error or "action conflict" is reported by listing the group of actions that, according to the grammar, would have to be performed concurrently or nondeterministically. A non-halting error is reported by indicating the action that, for certain input, would be repetitively executed infinitely.

3.7 <u>Code Assembly and Optimization</u>

Address locations are compiled and assembled into the code. Also the code is compacted by collapsing equivalent character sets into a single character set.

3.8 <u>Code Output</u>

The final code is output in the form of a FORTRAN BLOCK DATA subprogram.

4. THE OBJECT CODE INTERPRETER

The object code interpreter, in conjunction with the object code produced by the FSCAN compiler, forms a lexical analyzer that will process a stream of input characters and produce a stream of lexical units (tokens) as specified by the FSCAN program that was compiled. The interpreter is written in standard ANSI FORTRAN. In addition there is a group of short (1 to 5 line) routines that are machine dependent (see Appendix B).

4.1 Input Interface

The stream of input characters is obtained by the interpreter through repeated calls to the user-supplied routine, GETBUF. The subroutine, GETBUF, has four output parameters: three formal parameters and one array in a labeled common block:

```
SUBROUTINE GETBUF (IBEG,IEND, EOFFLG)
      .
      .
      .
COMMON  /user-defined-common-block/ ..., BUFFER(i), ...
      .
      .
      .
```

BUFFER is a user-defined array containing the characters to be sent to the scanner, with the characters stored one per word.

IBEG and IEND are integers pointing respectively to the first and last characters in BUFFER to be sent, and EOFFLG is a logical that is true iff there are no more characters to be sent. When EOFFLG is true, the values in BUFFER, IBEG, and IEND are irrelevant.

The common block containing BUFFER must be added to the routine EOIERR in the "Scanner Table Driver" module. The array containing the characters must be named BUFFER.

4.2 Output Interface

The stream of tokens is obtained by making repeated calls to the interpreter subroutine, SCANNR. SCANNR has four output parameters, all appearing in the labeled common block, /TOKENC/:

```
SUBROUTINE SCANNR
COMMON/TOKENC/TKNTYP, KTFLAG, ITKNCH, TKNCHR(30)
```

TKNTYP is an integer indicating the type of the token (see Appendix C), KTFLAG is a boolean that is true for a kept-token and false for a deleted-token, ITKNCH is an integer indicating the number of kept-characters in the token, TKNCHR is an array containing the kept-characters (one character per word).

4.3 Errors Reported by the Interpreter

4.3.1 Recoverable Errors

The following recoverable errors are reported by the lexical analyzer by generating a call of the form:

```
CALL SCNERR (i)
```

where i is an integer in the range, (1..10), indicating which error occurred.

1. Token is too long, i.e., the number of characters marked as kept is larger than the size of the array, TKNCHR. The default size of TKNCHR is 30. If longer tokens are desired the interpreter would have to be modified by increasing the size of TKNCHR and changing the initialization of the variable MTKNCH to be the new size.

Recovery: The token is truncated on the right.

2. Token contains erroneous characters. An erroneous character is one that is not an element of the set of expected characters of the state of the interpreter at the time the character was encountered. An erroneous character is processed by the interpreter by skipping over the erroneous character without changing the state of the interpreter.

Recovery: Erroneous characters are marked as deleted.

3. Token to be screened contains erroneous characters

Recovery: Erroneous characters are marked as deleted.

4. Screening terminated with characters remaining in token to be screened.

Recovery: The characters remaining in the token are ignored.

5. Erroneous characters occurred in token being screened, and screening terminated at the end of the token while skipping over erroneous characters.

Recovery: None necessary.

6. End of input stream occurred prematurely.

Recovery: An EOFTOK token is generated.

7. Erroneous characters occurred in input stream and end of input stream occurred while skipping over erroneous characters

Recovery: An EOFTOK token is generated.

8. End of token occurred prematurely while screening.

Recovery: Screening terminated and processing continues.

9. Erroneous characters occurred in input stream, and the end of the characters read in by the most recent call to GETBUT reached while skipping over erroneous characters.

Recovery: The lexical analyzer is reset to its initial state before the next call to GETBUF.

10. The current call to GETBUF returns more characters than for which there is room remaining in BUFFER.

Recovery: The lexical analyzer is reset to its initial state and the previous contents of BUFFER is flushed.

4.3.2 Fatal Errors

The following fatal errors are reported by the lexical analyzer by generating a call of the form:

```
CALL FTLERR (i)
```

where i is an integer in the range, (1..2)

1. The "call" stack overflowed.

To fix this error, the FSCAN program should be rewritten to generate less procedure-call nesting at run-time. Alternatively, the size of the array, CSTACK, in the labeled common block, /CSTAKC/, must be increased, and MCSTAC must be initialized in the block data subprogram, SCANBD, to a value corresponding to the new size of CSTACK.

2. The "keep" stack overflowed.

To fix this error, the FSCAN program should be rewritten to generate fewer tokens within the operands of an ELSE construct or the operand of a ?*. Alternatively, the size of the array, KSTACK, in the labeled common block, /KSTAKC/, must be increased, and MKSTAC must be initialized in the block data subprogram, SCANBD, to a value corresponding to the new size of KSTACK.

5. FSCAN-SUBSET OBJECT CODE INTERPRETER

For normal (non-FORTRAN) lexical analyzers, the full power of FSCAN is unnecessary. For these analyzers, a smaller and more efficient interpreter is available. This interpreter can be used on the object code produced from FSCAN programs that satisfy the following restrictions:

- Variable defining rules may not be used.
- The operators, ELSE, **, and ?* may not be used.
- Nonterminal actions may not be used.
- All characters of a token must occur in the characters returned from a single call to GETBUF.

5.1 Input Interface

The stream of input characters is obtained by the interpreter through repeated calls to the user-supplied routine, GETBUF. The subroutine, GETBUF, has one input formal parameter, NMCHRS, and two output formal parameters, CHRBUT and EOFFLG:

```
SUBROUTINE GETBUT (NMCHRS, CHRBUF, EOFFLG)
     .
     .
     .
DIMENSION CHRBUF(i)
     .
     .
     .
```

NMCHRS is an integer specifying the number of characters that should be placed in CHRBUF, one character per word. EOFFLG is a logical that is set to be true iff there are no more characters to be sent. When EOFFLG is true, the values in CHRBUF are irrelevant.

5.2 Output Interface

See 4.2.

5.3 Errors Reported by the Interpreter

5.3.1 Recoverable Errors

1, 2, 3. Errors 1, 2, and 6 from 4.3.1.

4. Token extends past end of the characters read in by the last call to GETBUF.

Recovery: The lexical analyzer is reset to its initial state and the current contents of CHRBUF is flushed.

5.3.2 Fatal Errors

1, 2. Errors 1 and 2 from 4.3.2.

3. Illegal action for the FSCAN-subset interpreter.

To fix this error, the FSCAN program should be rewritten to satisfy the requirements of the FSCAN-subset. Alternatively the regular interpreter (see Section 4) must be used instead of the subset interpreter.

6. ACKNOWLEDGMENT

The author gratefully acknowledges Frank DeRemer who initially specified the FSCAN language and the LR style processing of it. PTL.

References

[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.

[3] DeRemer, F., SVG Memos #69-72, #76-77, #80, #83-84. University of Colorado at Boulder, Boulder, Colorado.

[4] Waite, W., Dunn, R.C., "SYNPUT - a Tool for Processing Programming Language Syntax," SEG 78-5, Dept. of Electrical Engineering, University of Colorado at Boulder, Boulder, Colorado 80309.

Appendix A: Machine Dependencies in the FSCAN compiler

A.1 Machine Dependent Constants

A.1.1 EOLCHR

EOLCHR in /CHARSC/ is the "end-of-line" character, that conceptually should be equal to an "end-of-line" read in under A1 format, and should print an end-of-line when written out under A1 format. Since CDC does not have an end-of-line character, the null character is made the end-of-line character for the FSCAN compiler running on CDC equipment. The null character is left justified with blank fill to correspond to the standard A1 format for CDC FORTRAN.

A.1.2 NCHARS

NCHARS in /NCHARSC/ is the number of distinct characters in the character set of the machine. For the CDC 6400 implementation, NCHARS is 64.

A.2 Machine Dependent Primitives

A.2.1 INTEGER FUNCTION INTGER (CHAR)

Input: CHAR contains a character stored in 1H (or A1) format.

Result: An integer between 1 and NCHARS (see A.1.2), a unique value for each distinct character.

A.2.2 INTEGER FUNCTION CHRCTR (INT)

This is the inverse of the INTGER function.

A.2.3 INTEGER FUNCTION DIG (CHAR)

Input: same as INTGER

Result: If the character is a digit the result is the integer value of the digit (0-9); otherwise the result is -1.

A.2.4 INTEGER FUNCTION IAND (I1,I2)
INTEGER FUNCTION IOR (I1,I2)
INTEGER FUNCTION INOT (I1)

These functions return the result of the bitwise logical operation of AND, OR and NOT, respectively.

A.2.5 LOGICAL FUNCTION EOFILE (ICHANL)

Input: ICHANL is a logical channel number.

Result: True iff channel ICHANL is at logical end of file.

A.2.6 SUBROUTINE OHCONS (HCONST)
INTEGER FUNCTION NHCCHR(HCONST)

Input: HCONST is a hollerith constant of the form $nHc_1c_2...c_n$ where n is an unsigned positive integer and c_i is a character.

Result: After an initial call of OHCONS, each successive call to NHCCHR will return the next character of HCONST, in the same format as the result of CHRCTR. Thus the i^{th} call to NHCCHR will return c_i, stored in A1 or 1H format. For $i > n$, the result of the i^{th} call to NHCCHR is undefined.

A.2.7 SUBROUTINE TIMMES (ICHANL)

This subroutine prints a message indicating the current time and date, to the logical channel, ICHANL.

A.2.8 SUBROUTINE INIETM
SUBROUTINE ENDETM (EXCTIM)

Result: EXCTIM is a real, set to be the number of seconds of CPU execution time that have elapsed since the most recent call to INIETM.

A.2.9 INTEGER FUNCTION LRS (IVAL, ICOUNT)
INTEGER FUNCTION LLS (IVAL, ICOUNT)

LRS and LLS return the logical shift (end-off, zero-fill), right and left respectively, of ICOUNT binary positions of the value, IVAL.

A.3 Table Generation

Table generation is performed by the "Print Final Tables" module which contains the one subroutine, PTABL2. The final table (object code) is printed out in the format described in Appendix C. If a different machine requires another table format, the FORMAT statements of PTABL2 would have to be appropriately modified.

A.4 Character Sets

Character sets are implemented as bit vectors. This implementation is encapsulated in the "Character Set" module. CSET in /CSETCM/ is an array of character-set bit-vectors. NBSPWD is the number of bits per word. NWSPCS is the number of words per character set, whose value is the first dimension of CSET. The value of NWSPCS must satisfy the condition,

$$\text{NWSPCS} * \text{NBSPWD} \geq \text{NCHARS} + 2$$

otherwise a system error is reported.

In the CDC 6400 implementation, NBSPWD is 59 and NCHARS is 64, so NWSPCS is set to be 2. Only 59 of the 60 bits in a word on the 6400 are used, because the CDC choice of 1's complement representation of integers can cause a bit vector with all bits set (-0) to be confused with a bit vector with no bits set (+0).

Appendix B: Machine Dependencies in the FSCAN object code interpreter.

The following machine dependent primitives are required:

1. INTEGER FUNCTION INTGER (CHAR)
2. INTEGER FUNCTION CHRCTR (INT)
3. INTEGER FUNCTION DIG (CHAR)
4. LOGICAL FUNCTION EOFILE (ICHANL)
5. SUBROUTINE OHCONS (HCONST)
6. INTEGER FUNCTION NHCCHR (HCONST)
7. INGEGER FUNCTION LRS (IVAL, ICOUNT)
8. INTEGER FUNCTION LLS (IVAL, ICOUNT)

These routines are described in Appendix A.

Appendix C: Scanner tables and accessing primitives.

For the purpose of interfacing the lexical analyzer produced from an FSCAN program with the parser that uses the tokens generated, the FSCAN compiler outputs a FORTRAN block data subprogram, TKTPBD, in which each token specified in the FSCAN program is initialized to the integer value that will be output by the lexical analyzer when that token is generated.

The object code produced by the FSCAN compiler is in the form of tables stored in a block data subprogram named STBLBD. Since the FSCAN compiler uses a scanner produced by the FSCAN system, such a block data subprogram appears in the source code for the FSCAN compiler.

For storage efficiency the tables are packed, and since transportation to machines with different word sizes could require different packing formats, the object code interpreter accesses the tables only through accessing primitives. In this way, if the tables are reformatted only the accessing primitives need be correspondingly modified. There is one accessing subroutine that modifies a value in the table and seven accessing functions. The subroutine is SETEXP. The seven functions are: IN, AKTYPE, AKBITV, CALLAK, VALLOC, NEXTAK, and EXPVAL, where IN returns a logical and the rest return an integer.

The tables actually consist of three arrays, EXPONT, BTVCTR, and AKSHUN, appearing respectively in the common blocks, /EXPTCM/, /BTVRCM/, /AKSHCM/.

Each element of the EXPONT array contains an integer value, where the i'th EXPONT is referenced by the function call EXPVAL (i), and where the i'th EXPONT is assigned the value NUM by the subroutine call CALL SETEXV (i,NUM).

Each element of the BTVCTR array contains N single bit items, where N = 2 + NCHARS (see A.1.2). The i'th item of the j'th element of BTVCTR is tested by the function call IN(i,j) where IN returns .TRUE. iff the item is 1 (i.e., the bit is set).

Each element of the AKSHUN array contains five integer values, accessed by the routines, AKTYPE, AKCSET, CALLAK, VALLOC, and NEXTAK, where,for example,the AKTYPE of the i'th element of STATE is referenced by the function call AKTYPE (i). The space required for the AKTYPE, AKCSET, CALLAK, VALLOC, and NEXTAK fields is 5 bits, 10 bits, 11 bits, 11 bits, and 11 bits,respectively.

In the CDC 6400 implementation the arrays are formatted as follows: (Note: The CDC 6400 has 60 bit words, with the bits labeled 0 to 59 from left to right).

EXPONT: Each value is stored in a single word in standard binary integer format.

BTVCTR: As N = 66, each BTVCTR element consists of two words. Items 1 to 59 are stored in bits 0 to 58 of the first word. Items 60 to 66 are stored in bits 0 to 6 of the second word. Bit 59 of the first word and bits 7 to 59 of the second word are always 0.

AKSHUN: Each element of AKSHUN consists of one word, where each of the five items occupies 12 bits, in the order, AKTYPE, AKCSET, CALLAK, VALLOC, NEXTAK. Thus, AKTYPE is stored in bits 0-11, AKCSET in bits 12-23, CALLAK in bits 24-35, VALLOC in bits 36-47, and NEXTAK in bits 48-59.

Appendix D: FSCAN compiler parser tables.

The FSCAN parser tables were produced from the tables generated by the SYNPUT parser generating system [4]. Their format corresponds to the tables described in Appendix C, where the array PRSBTV corresponds to the array BV, and the array PRSTAB corresponds to the array STATE. In addition, in the parser tables the values, NDRSET, NLXEME, and NPCMND are made available through the common block, /PTCONC/, where NDRSET is the number of elements in the PRSBTV array, NLXEME is the number of items in a PRSBTV element, and NPCMND is the number of elements in the PRSTAB ARRAY.

The accessing functions are DIRSET, PARSOP, PARDST, AND PDATUM, corresponding to IN, AKTYPE, AKCSET, and CALLAK respectively. Thus, DIRSET returns a logical and the rest return an integer.

In the CDC 6400 implementation the arrays are formatted as follows:

PRSBTV: There are 23 items, which are stored in bits 0-22.

PRSTAB: PARSOP is stored in bits 43-46, PARDST in bits 47-50, and PDATUM in bits 51-59.

Appendix E: Syntax of FSCAN programs

```
PROGRAM → SCANNER '.' ;
SCANNER

   → 'SCANNER' GOAL_SYMBOL ':' (RULE ';') + 'END' GOAL_SYMBOL;

RULE
   → NONTERMINAL ('→' REG_EXPRN('=>' ACTION)?)+
   → VARIABLE '=' REG_EXPRN
   → SCANNER ;

REG_EXPRN → REG_TERM list '|',

REG_TERM → REG_PHRASE + ;

REG_PHRASE → REG_FACTOR ('LIST' REG_FACTOR) ? ;

REG_FACTOR

   → REG_PRIMARY ('*'|'+'|'?')?

   → 'NOT' REG_PRIMARY ;

REG_PRIMARY

   → '(' REG_EXPRN ? ')'

   → NONTERMINAL list 'ELSE'

   → NONTERMINAL ('**'|'?*') EXPONENT

   → TERMINAL ;

ACTION → SCREENER | TERMINAL ;

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)*

Appendix F:

In this appendix an example of a complete FSCAN program is presented. This program specifies a lexical analyzer for the FORTRAN dialect accepted by the CDC FTN compiler. The program appears on the next seven pages.

```
SCANNER FC :
   FC     -> (ULSTMT/COMMENT)* ;

   ULSTMT -> BSTMT CSTMT?*19 "EOL" => FORTRANSTMT ;
   BSTMT  -> NOT("C"/"*") KC**71 DC* ;
   SCANNER CSTMT :
      CSTMT -> 'EOL' SBLANK**5 NOT(' '/'0') BSTMT ;
      SCANNER SBLANK :
         SBLANK -> ' ' ; END SBLANK ; END CSTMT ;
   COMMENT -> ('C'/'*') DC* 'EOL' ;
   SCANNER DC :
      DC -> NOT('EOL') ; END DC ;
   SCANNER KC :
      KC -> NOT("EOL") ; END KC ;
   SCANNER FORTRANSTMT:
      FORTRANSTMT -> LABELF BLANKCF (' '*)
                     (STMT ELSE NULL) TEXT EOS  ;
      END FORTRANSTMT ;

   LABELF -> KC**5 => SCANLBL ;
   SCANNER SCANLBL :
      SCANLBL -> (' '*)  LABEL? ;
      END SCANLBL ;
   LABEL ->  DIG (DIG/' ')*                                  => "DCONST" ;
   DIG   ->  "0"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9" ;

   BLANKCF -> (' '/'0')
           -> NOT (' '/'0')  => 'EOS' ;  # EOS USED AS 'ERROR-TOKEN'

   SCANNER STMT:
      STMT
      -> (BLOCKDATA/SUBROUTINE/FUNCTION/EXTERNAL/DIMENSION
          /EQUIVALENCE/CONTINUE)
      -> IMPLICIT ((DATATYPE PARENS) LIST COMMA)
      -> (FEND/RETURN) EOLCHK
      -> TYPE? DATATYPE (FCN ELSE NULL) NAME PARENS? (EOLCHK/COMMA)
      -> DATA NAME PARENS? (COMMA/SLASH)
      -> COMMON (SLASH/NAME)
      -> DO LABEL NAME EQUALS (NAME/ICONST) COMMA
      -> IF PARENS ((LABEL LIST COMMA) / (STMT ELSE NULL) EQTRAP)
      -> ASSIGN LABEL TO NAME
      -> GOTO (LABEL
               /(NAME COMMA?)? PARENS (COMMA NAME)?)  EOLCHK
      -> (CALL/PROGRAM) NAME PARENS? EOLCHK
      -> (STOP/PAUSE) OCONST? EOLCHK
      -> (READ/WRITE/PRINT/PUNCH) IOREF
      -> (REWIND/BACKSPACE/ENDFILE) (ICONST/NAME)
      -> FORMAT FORMATSPECIFICATION EOLCHK
      -> LEVEL ICONST COMMA
      -> (ENCODE/DECODE) PARENS? EQTRAP
      -> ENTRY NAME EOLCHK ;

# EOLCHK ENSURES THAT THE NEXT CHARACTER IS AN EOL,
# WITHOUT PROCESSING THE EOL.
      EOLCHK -> SCEOL**0 ;
      SCANNER SCEOL : SCEOL -> 'EOL' ; END SCEOL ;
# EQTRAP CAUSES THE CURRENT ALTERNATIVE TO FAIL IF AN EQUALS-SIGN IS
# THE NEXT INPUT STREAM CHARACTER, SINCE '' MATCHES NO CHARACTERS.
      EQTRAP -> ('=' '')? ;

      DATATYPE -> (LOGICAL/INTEGER/DOUBLEPRECISION/COMPLEX/REAL) ;

      IOREF -> LPAREN (NAME/ICONST) (COMMA FORM)? RPAREN EQTRAP
            -> FORM (COMMA/EOLCHK) ;
      FORM  -> NAME/LABEL/SGLSTR ;

      END STMT;

#
```

```
SCANNER NULL : NULL -> (); END NULL ;

SCANNER FCN:
   FCN -> FUNCTION ;
   END FCN;

SCANNER PARENS:
   PARENS -> LPAREN (NAMLITOPSEP/PARENS)* RPAREN;
   END PARENS;

SCANNER NAMLITOPSEP:
   NAMLITOPSEP -> NAME/LITERAL/OPERATOR/SEPARATOR ;
   END NAMLITOPSEP ;

SCANNER TEXT:
   TEXT -> (NAMLITOPSEP/LPAREN/RPAREN)* ;
   END TEXT;
   ASSIGN          -> A S S I G N                                          => 'KASSI' ;
   BACKSPACE       -> B A C K S P A C E                                    => 'KBACK' ;
   BLOCKDATA       -> B L O C K D A T A                                    => 'KBDAT' ;
   CALL            -> C A L L                                              => 'KCALL' ;
   COMMON          -> C O M M O N                                          => 'KCOMM' ;
   COMPLEX         -> C O M P L E X                                        => 'KCOMP' ;
   CONTINUE        -> C O N T I N U E                                      => 'KCONT' ;
   DATA            -> D A T A                                              => 'KDATA' ;
   DECODE          -> D E C O D E                                          => 'KDECO' ;
   DIMENSION       -> D I M E N S I O N                                    => 'KDIME' ;
   DO              -> D O                                                  => 'KDO' ;
# IN FTN FORTRAN, "DOUBLEPRECISION P,Q,R" AND "DOUBLE P,Q,R" ARE BOTH
# LEGAL STATEMENTS, THUS THE FOLLOWING IS REQUIRED TO RESOLVE THE
# AMBIGUITY WHEN "P" IS READ.
   DOUBLEPRECISION -> D O U B L E PRECISION?*1                             => 'KDOUB' ;
   FEND            -> E N D                                                => 'KEND' ;
   ENCODE          -> E N C O D E                                          => 'KENCO' ;
   ENDFILE         -> E N D F I L E                                        => 'KENDF' ;
   ENTRY           -> E N T R Y                                            => 'KENTR' ;
   EQUIVALENCE     -> E Q U I V A L E N C E                                => 'KEQUI' ;
   EXTERNAL        -> E X T E R N A L                                      => 'KEXTE' ;
   FORMAT          -> F O R M A T                                          => 'KFORM' ;
   FUNCTION        -> F U N C T I O N                                      => 'KFUNC' ;
   GOTO            -> G O T O                                              => 'KGOTO' ;
   IF              -> I F                                                  => 'KIF' ;
   IMPLICIT        -> I M P L I C I T                                      => 'KIMPL' ;
   INTEGER         -> I N T E G E R                                        => 'KINTG' ;
   LEVEL           -> L E V E L                                            => 'KLEVE' ;
   LOGICAL         -> L O G I C A L                                        => 'KLOGI' ;
   PAUSE           -> P A U S E                                            => 'KPAUS' ;
   PRINT           -> P R I N T                                            => 'KPRIN' ;
   PROGRAM         -> P R O G R A M                                        => 'KPROG' ;
   PUNCH           -> P U N C H                                            => 'KPUNC' ;
   READ            -> R E A D                                              => 'KREAD' ;
   REAL            -> R E A L                                              => 'KREAL' ;
   RETURN          -> R E T U R N                                          => 'KRETU' ;
   REWIND          -> R E W I N D                                          => 'KREWI' ;
   STOP            -> S T O P                                              => 'KSTOP' ;
   SUBROUTINE      -> S U B R O U T I N E                                  => 'KSUBR' ;
   TO              -> T O                                                  => 'KTO' ;
   TYPE            -> T Y P E ;
   WRITE           -> W R I T E                                            => 'KWRIT' ;

   SCANNER PRECISION:
      PRECISION -> P R E C I S I O N ; END PRECISION ;

#
```

```
A    -> 'A'  ('  '*)  ;
AK   -> "A"  ('  '*)  ;
B    -> 'B'  ('  '*)  ;
C    -> 'C'  ('  '*)  ;
D    -> 'D'  ('  '*)  ;
DK   -> "D"  ('  '*)  ;
E    -> 'E'  ('  '*)  ;
EK   -> "E"  ('  '*)  ;
F    -> 'F'  ('  '*)  ;
FK   -> "F"  ('  '*)  ;
G    -> 'G'  ('  '*)  ;
GK   -> "G"  ('  '*)  ;
H    -> 'H'  ('  '*)  ;
I    -> 'I'  ('  '*)  ;
K    -> 'K'  ('  '*)  ;
L    -> 'L'  ('  '*)  ;
LK   -> "L"  ('  '*)  ;
M    -> 'M'  ('  '*)  ;
N    -> 'N'  ('  '*)  ;
NK   -> "N"  ('  '*)  ;
O    -> 'O'  ('  '*)  ;
OK   -> "O"  ('  '*)  ;
P    -> 'P'  ('  '*)  ;
Q    -> 'Q'  ('  '*)  ;
QK   -> "Q"  ('  '*)  ;
R    -> 'R'  ('  '*)  ;
RK   -> "R"  ('  '*)  ;
S    -> 'S'  ('  '*)  ;
SK   -> "S"  ('  '*)  ;
T    -> 'T'  ('  '*)  ;
TK   -> "T"  ('  '*)  ;
U    -> 'U'  ('  '*)  ;
UK   -> "U"  ('  '*)  ;
V    -> 'V'  ('  '*)  ;
W    -> 'W'  ('  '*)  ;
X    -> 'X'  ('  '*)  ;
XK   -> "X"  ('  '*)  ;
Y    -> 'Y'  ('  '*)  ;

COMMA   ->   ','  ('  '*)                                              => 'COMMA'  ;
SLASH   ->   '/'  ('  '*)                                              => 'SLASH'  ;
EQUALS  ->   '='  ('  '*)                                              => 'EQUALS' ;
SGLSTR  ->   '*'  ('  '*)                                              => 'SGLSTR' ;
LPAREN  ->   '('  ('  '*)                                              => 'LPAREN' ;
RPAREN  ->   ')'  ('  '*)                                              => 'RPAREN' ;

#
```

```
   NAME    ->  LETTER  (LETTER/DIGIT) *                                   => "NAME" ;

   LETTER -> ("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") (' '*) ;
   DIGIT  -> ("0"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9") (' '*) ;
   ODIGIT -> ("0"/"1"/"2"/"3"/"4"/"5"/"6"/"7")  (' '*) ;

# TO PROCESS FORTRAN CORRECTLY, SCANICONST MUST PRECEDE SCANACONST, BUT
# IN CASE BOTH FAIL, SCANICONST PROVIDES SUPERIOR ERROR RECOVERY, THUS THE
# FOLLOWING CONSTRUCT IS USED.
   LITERAL  -> (SCANICONST ELSE SCANACONST ELSE SCANICONST)
            ->  POINTACONST/LCONST/QUOTHCONST ;
   SCANNER SCANACONST :
      SCANACONST -> ACONST ; END SCANACONST ;
   SCANNER SCANICONST :
      SCANICONST -> ICONST (LCONST/OPERATOR) ?
                 -> EXTHCONST
                 -> DIGIT+ EEXP SIGN? DIGIT+                              => "RCONST"
                 -> DIGIT+ DEXP SIGN? DIGIT+                              => "DPCNST" ;
      END SCANICONST ;

   ICONST -> DIGIT+                                                       => "DCONST"
          -> ODIGIT+ B                                                    => "OCONST" ;
   OCONST -> ODIGIT+                                                      => "OCONST" ;
   ACONST -> DIGIT+ POINT DIGIT* (EEXP SIGN? DIGIT+)?                     => "RCONST"
          -> DIGIT+ POINT DIGIT* DEXP SIGN? DIGIT+                        => "DPCNST" ;
   POINTACONST -> POINT DIGIT+ (EEXP SIGN? DIGIT+)?                       => "RCONST"
               -> POINT DIGIT+ DEXP SIGN? DIGIT+                          => "DPCNST" ;
   EEXP  -> "E" (' '*) ;
   DEXP  -> "D" (' '*) ;
   POINT -> "." (' '*) ;
   SIGN  -> ("+"/"-") (' '*) ;

   HCONST      -> LENGTH 'H' DC**LENGTH (' '*)                            => 'HCONST' ;
   EXTHCONST   -> LENGTH ('H'/'L'/'R') DC**LENGTH (' '*)                  => 'HCONST' ;
   QUOTHCONST -> '#'' (NOT('EOL'/'#'')/('#'' '#''))+ '#'' (' '*)         => 'HCONST' ;
   ASTHCONST   -> '*' NOT('EOL'/'*')+ '*'  (' '*)                         => 'HCONST' ;

   LENGTH      =  DIGIT+ ;

#
```

```
LCONST    -> DOT ((TK RK UK EK)/(FK AK LK SK EK)) DOT              => "LCONST" ;
OPERATOR -> DOT LK TK DOT                                          => "RELOP"
         -> DOT LK EK DOT                                          => "RELOP"
         -> DOT EK QK DOT                                          => "RELOP"
         -> DOT NK EK DOT                                          => "RELOP"
         -> DOT GK EK DOT                                          => "RELOP"
         -> DOT GK TK DOT                                          => "RELOP"
         -> DOT OK RK DOT                                          => 'OR'
         -> DOT AK NK DK DOT                                       => 'AND'
         -> DOT NK OK TK DOT                                       => 'NOT'
         -> '*' (' '*) '*' (' '*)                                  => 'DBLSTR'
         -> SGLSTR
         -> SLASH
         -> '+' (' '*)                                             => 'PLUS'
         -> '-' (' '*)                                             => 'MINUS' ;

DOT -> "." (' '*) ;

SEPARATOR -> COMMA/EQUALS ;

SCANNER FORMATSPECIFICATION :
   FORMATSPECIFICATION
   -> LPAREN (SLASH/COMMA/FIELD)* RPAREN ;
   SCANNER FIELD:
      FIELD -> HCONST/QUOTHCONST/ASTHCONST
              /NHDESC/(ICONST? FORMATSPECIFICATION) ;
      END FIELD ;
   NHDESC -> DIGIT+ XK
              / DIGIT* ILAORZ DIGIT+
              / SCALE? DIGIT* FEGD DIGIT+ POINT DIGIT+             => "FIELD" ;
   ILAORZ -> ("I"/"L"/"A"/"O"/"R"/"Z") (' '*) ;
   FEGD   -> ("F"/"E"/"G"/"D") (' '*) ;
   SCALE  -> MINUS? DIGIT+ PE ;
   MINUS  -> "-" (' '*) ;
   PE     -> "P" (' '*) ;
   END FORMATSPECIFICATION ;

EOS -> 'EOL'                                                       => 'EOS' ;

END FC.
```
#