FSCAN-83 Report and User's Manual

by

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

CU-CS-248-83 January 1983

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

CONTENTS

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 ANSI FORTRAN (1966) supplemented by a small number of short machine dependent subroutines. Included is an FSCAN program describing a FORTRAN-77 lexical analyzer.

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 standards [1,2] specify that spaces for the most part are meaningless in FORTRAN programs. 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 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 10 I = 1, 3
```

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

```
DO10I = 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 [3] 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 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 [4].

2. 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.

An FSCAN program consists of the keyword, TOKENS, followed by a list of the tokens to be generated, followed by a single FSCAN procedure (within which may be defined additional procedures) terminated by a period. An FSCAN procedure specifies in an extended BNF-style notation a grammar that describes a left-to-right pass over the source text. During this pass each character is examined and depending on the character and the current state of the lexical analyzer, one of the following actions is taken:

1. mark the character as kept or deleted and move ahead to the next character

2. call an FSCAN procedure

3. exit an FSCAN procedure

4. exit an FSCAN procedure and backup to the state and location in the source text at which the procedure was called

5. perform a specific token-action

The compiler verifies that an FSCAN program specifies a deterministic lexical analyzer, i.e., that for any state of the analyzer, the next action to be performed can be uniquely determined from the character currently being examined.

2.1. Procedures

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. Each rule in the sequence is terminated by a semicolon.

Example

```
SCANNER DIG:
   rule_1; rule_2; ... rule_n;
   END DIG
```

Semantics

One of the rules must be a definition for the goal symbol of the procedure. This rule specifies the finite-state stack-automaton scan of the source text which is performed when the procedure is called. The scan is performed in a longest match manner; namely, given the choice between finishing and scanning more of the source text, the procedure will always continue scanning.

2.2. Rules

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

2.2.1. 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. Each alternative may have an associated token-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.

Syntax

Each alternative is preceded by a single-right-arrow (->). The optional token-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 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, after an alternative has been successfully matched with the source text, the corresponding token-action, if any, is performed.

2.2.2. Procedure Rule

Syntax

A procedure rule is simply an FSCAN procedure.

Semantics

During execution of the interpreter, when a nonterminal associated with a procedure rule is to be matched with the source text, the appropriate procedure is called.

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

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 immediately following character is treated as the next character of the string, even if that character is a double-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.

Examples

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

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, and then the next character in the source text is examined.

2.3.1.2. Nonterminals

Syntax

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

Examples

```
A TEMP TEMP1 B3B
```

Semantics

Nonterminals can name macro rules 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. When the nonterminal names a procedure, it indicates that the appropriate procedure is to be called during execution.

2.3.1.3. Integers

Syntax

An integer is a string of digits.

Examples

54   0   05   1234567890

Semantics

Integers have their usual meaning.

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, let a, b, c be characters, and let n be a non-negative integer.

2.3.2.1. 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 do. 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

Syntax

<>	:	<abc...>	= ('a' 'b' 'c' ...)
<<>>	:	<<abc...>>	= ("a" "b" "c" ...)
+	:	A+	= A A*
?	:	A?	= A / ()
//	:	A // B	= A (B A)*
ELSE	:	A ELSE B ELSE ...	≅ A / B / ...
**	:	A**n	≅ A A ... A (n times)
	:	A**(n)	≅ A? A? .. A? (n times)
	:	A**()	≅ A*

Restrictions: The operands of ELSE and the first operand of ** 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. The operators, ELSE and **, are only approximately equivalent to their respective syntactic expansions, because they possess the following additional properties:

ELSE

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

**n

The only distinction between **n and its syntactic expansion occurs when the exponent, n, is zero. In this case A**0 matches the input stream only if A would match the next character in the input stream. Since the exponent is 0, no characters are actually matched by A, only the check is performed. This can be used to cause the success or failure of a particular branch of the ELSE operator.

**(n)

The **(n) operator provides limited backup, in the sense that, if less than n A's have been successfully matched, the scan is backed up to the state at which the last A (possibly no A's) has been successfully matched.

**()

The **() operator is the same as the **(n) operator except that there is no limit to the number of A's that can be matched.

2.4. Token-Actions

Syntax

A token-action is a kept or deleted string followed by a nonterminal in parentheses. Either the string or the nonterminal in parentheses may be omitted.

Examples

"NAME"(KEYWORD) "STRING" (OPERATOR) 'BEGIN'

Semantics

A token-action generates a sequence of characters consisting of all characters marked as kept since the last token-action. The presence of a nonterminal in parentheses indicates that this sequence of characters is to be "screened" or rescanned by the procedure rule named by the nonterminal. If the screening procedure completely processes the characters without encountering any erroneous or "unmatchable" characters, all actions generated during the screening (including token-actions) are performed; otherwise, all such actions are ignored and a token is output. The string of the token-action names the type of the token to be output. All such strings used by an FSCAN program must be listed following the keyword, TOKENS, at the beginning of the FSCAN program. During runtime, the generation of the n'th token in this list is indicated by the output of the integer n+1 (the integer, 1, indicates end-of-file).

If the string is omitted, the screening is unconditionally performed with standard error recovery at erroneous characters. If the nonterminal in parentheses is omitted the token is unconditionally output, without any preceding attempt to screen.

2.4.1. End-of-File Token-Action

A special token is reserved to indicate the end of scanner processing. After this token has been generated, all following requests for tokens from the scanner will result in the return of this special token. The procedure that is the FSCAN program, i.e.,

```
TOKENS ...
SCANNER LEXANLYZ :
    LEXANLYZ ->   ...
    END LEXANLYZ.
```

is conceptually embedded in the following context:

```
        TOKENS EOFTOK ...
        SCANNER DEFAULT :
            DEFAULT -> LEXANLYZ EOF ;
            EOF -> () => 'EOFTOK' ;
            SCANNER LEXANLYZ : ... END LEXANLYZ ;
            END DEFAULT.
```

EOFTOK is therefore predefined in all FSCAN programs and is indicated during runtime by the output of the integer, 1.

2.4.2. Evaluation Token-Action

The FORTRAN Hollerith constant requires special treatment by the lexical analyzer. In particular, the lexical analyzer must be driven by a numeric value contained in the source text. To provide this function, a special "evaluate" token-action is included in FSCAN.

Syntax

The normal screening nonterminal is replaced by an equals sign.

Examples

```
(=)   "COUNT"(=)
```

Semantics

The sequence of characters generated by the token action are evaluated as a positive decimal integer. The compiler ensures that only digits can be marked as kept in an alternative possessing an evaluate token-action. The value resulting from this evaluation can then be referenced by the FSCAN program by using the name of the rule containing the evaluate token-action as an exponent in the ** or *? operators. The value of such a "variable" exponent is always the result of the most recent evaluate token-action performed by the rule named by the variable.

3. THE COMPILER

The FSCAN compiler consists of 5500 lines of standard ANSI FORTRAN code. In addition, there is a group of short (1 to 5 lines) routines that are machine dependent. (See Appendix A).

The compiler takes one input file containing an FSCAN program and produces three output files - a listing file annotated with the number of the first token on each line, a tables file containing the generated object code, and an errors file describing any errors in the input. The files are associated with the FORTRAN logical unit numbers five, six, seven, and zero respectively.

The 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:

(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).

(2) A nonterminal is defined by two different rules within the same scanner.

(3) No rule defines the goal symbol of a scanner.

(4) A variable exponent is defined in something other than a rule with an evaluate token-action.

(5) A symbol is used that has not been defined by any rule.

(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:

(1) A macro rule is recursively defined.

(2) A variable exponent is used before the variable could have received a value.

(3) A 'NOT' operator is applied to something other than a character set.

(4) A terminal string other than EOL consists of more than one character.

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

(6) A rule generating a token is used in a context where another token is currently being built.

(7) A rule containing untokenized kept characters and a rule producing tokens appear in the same context (either error 5 or error 6).

(8) Non-digit characters are kept in a context where an evaluate token-action could occur.

(9) A token type is used without being declared in the TOKENS section.

(10) A token type is multiply declared in the TOKENS section.

(11) A token type is declared to be deleted(kept), but used as kept(deleted).

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 lexical analyzer that will perform the analysis specified by the user's grammar is generated. This code is written out to a scratch file as it is produced.

3.6. Code Verification

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. Code Assembly and Optimization

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. Code Output

The final code is output in the form of FORTRAN BLOCK DATA subprograms and appropriate accessing functions.

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 one input formal parameter, MBUFFR, and four output formal parameters, BUFFER, LBUFFR, EOLFLG, and EOFFLG:

```
SUBROUTINE GETBUF (MBUFFR,BUFFER,LBUFFR,EOLFLG,EOFFLG)
INTEGER MBUFFR,BUFFER,LBUFFR
LOGICAL EOLFLG,EOFFLG
DIMENSION BUFFER(MBUFFR)
     .
     .
     .
```

MBUFFR specifies the maximum number of characters that should be placed in BUFFER, one character per array element.

LBUFFR specifies the number of characters that were placed in BUFFER. EOLFLG is set to be true iff an EOL character is to be appended to the stream of characters being returned in BUFFER. This EOL character is referenced in an FSCAN program by the terminal 'EOL' or "EOL". EOFFLG is set to be true iff there are no more characters to be sent. When EOFFLG is true, the values of BUFFER, LBUFFR, and EOLFLG are ignored.

4.2. Output Interface

The interpreter must be initialized by a call to the subroutine INISCN. Following this initialization, the stream of tokens is obtained by making successive calls to the 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 variable indicating the type of the token, KTFLAG is a logical variable that is true for a kept-token and false for deleted-token, ITKNCH is an integer variable indicating the number of kept-characters in the token, TKNCHR is an array containing the kept-characters (one character per array element).

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 GETBUF 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 there is room for in the internal character buffer of the lexical analyzer.

Recovery: The lexical analyzer is reset to its initial state and the previous contents of its internal buffer is flushed. Note: It may be necessary to increase the size of the internal buffer to prevent this error. See fatal error six.

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..4)

(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.

(3) Illegal action on call stack.

An internal error that should never occur.

(4) Error in backup.

An internal error that should never occur.

5. FSCAN-SUBSET OBJECT CODE INTERPRETER

For many 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:

- The operators, ELSE, **, and ?* may not be used.
- Nonterminal and evaluate token-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

See standard interpreter.

5.2. Output Interface

See standard interpreter.

5.3. Errors Reported by the Interpreter

5.3.1. Recoverable Errors

(1) Recoverable error 1 from standard interpreter.

(2) Recoverable error 2 from standard interpreter.

(3) Recoverable error 6 from standard interpreter.

(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 BUFFER is flushed.

5.3.2. Fatal Errors

(1) Fatal error 1 from standard interpreter.

(2) Fatal error 2 from standard interpreter.

(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 must be used instead of the subset interpreter.

(4) GETBUF returned too many characters.

To fix this error, the GETBUF routine should be rewritten to return fewer than MBUFFR characters, (i.e. MBUFFR > LBUFFR), where MBUFFR and LBUFFR are arguments to the GETBUF routine.

References

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

[2] ANSI : FORTRAN 77. X3.9-1978, American National Standards Institute 1978.

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

[4] DeRemer, F., SVG Memos #69-72, #76-77, #80, #83-84. Dept. of Computer Science, University of Colorado at Boulder, Boulder, Colorado, 1977.

Appendix A: Machine Dependencies in the FSCAN compiler

1. Machine Dependent Constants

1.1. NBTPWD

NBTPWD in /NBTPWC/ is the number of bits in a machine word.

2. Machine Dependent Primitives

2.1. INTEGER FUNCTION INTGER (CHAR)

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

Result:
The ASCII code for the character, CHAR (an integer between 0 and 127).

2.2. INTEGER FUNCTION CHRCTR (INT)

This is the inverse of the INTGER function.

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.

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.

2.5. INTEGER FUNCTION HOLCHR (HCONST,ICHAR)

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, i=1..n. ICHAR is an integer between 1 and n.

Result:
HOLCHR(HCONST,i) will return c_i, stored in A1 or 1H format.

2.6. 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.

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. INTEGER FUNCTION LRS (IVAL, ICOUNT)
5. INTEGER FUNCTION LLS (IVAL, ICOUNT)

These routines are described in Appendix A.

Appendix C : Syntax of FSCAN programs

```
PROGRAM -> 'TOKENS' TERMINAL+ SCANNER '.' ;

SCANNER
   -> 'SCANNER' GOAL_SYMBOL ':'
         (RULE ';')+ 'END' GOAL_SYMBOL ;

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

REG_EXPRN -> REG_TERM // '/' ;

REG_TERM -> REG_PHRASE+ ;

REG_PHRASE -> REG_FACTOR ('//' REG_FACTOR)? ;

REG_FACTOR
   -> REG_PRIMARY ('*'/'+'/'?')?
   -> 'NOT' REG_PRIMARY ;

REG_PRIMARY
   -> '(' REG_EXPRN? ')'
   -> NONTERMINAL // 'ELSE'
   -> NONTERMINAL '**' EXPONENT
   -> TERMINAL ;

ACTION
   -> TERMINAL SCREENER?
   -> SCREENER ;

SCREENER
   -> '(' NONTERMINAL ')'
   ->  '(' '=' ')' ;

EXPONENT
   -> INTEGER
   -> ( INTEGER )
   -> ( ) ;

INTEGER -> NONTERMINAL / '<INTEGER>' ;

GOAL_SYMBOL -> '<NAME>' ;

NONTERMINAL -> '<NAME>' ;

SCREENER -> '<NAME>' ;

TERMINAL -> '<KEPT_STRING>' / '<DELETED_STRING>' ;
```

Appendix D : Examples of FSCAN Programs

Following are three complete FSCAN programs. They describe lexical analyzers for the FSCAN language, PASCAL, and FORTRAN-77 respectively.

```
TOKENS
   "IDNTFR" "INTEGR" "KSTRNG" "DSTRNG" "DELMTR" "OPRATR"
SCANNER FSCAN :
# THIS IS THE FSCAN PROGRAM USED TO CREATE THE LEXICAL ANALYZER FOR
# THE FSCAN COMPILER.  SCREENING OF KEYWORDS FROM IDENTIFIERS
# AND LEGAL OPERATORS FROM OPERATORS IS DONE AUTOMATICALLY BY THE
# SYMBOL TABLE MECHANISM AND IS THEREFORE NOT PERFORMED BY THE
# LEXICAL ANALYZER.

   FSCAN -> (SPACES FSCAN1)* SPACES ;
   SPACES -> (' ' / 'EOL')* ;

   SCANNER FSCAN1 :
      FSCAN1 -> NAME/INTEGER/KSTRING/DSTRING/KKEYWORD
               /DKEYWORD/DELIMITER/OPERATOR/COMMENT ;
      END FSCAN1 ;

   NAME -> KACHAR (KACHAR / KDIGIT)*                     => "IDNTFR" ;

   INTEGER -> KDIGIT+                                    => "INTEGR" ;

   KSTRING -> DQ (NOTDQSH / SHARP KC)* DQ                => "KSTRNG" ;
   DSTRING -> SQ (NOTSQSH / SHARP KC)* SQ                => "DSTRNG" ;

   KKEYWORD -> '<' '<' KKEYCHAR* '>' '>' ;
   DKEYWORD -> '<' DKEYCHAR* '>' ;
   KKEYCHAR -> NOT(KEYDLM/"##") / SHARP KEYDLM           => "KSTRNG" ;
   DKEYCHAR -> NOT(KEYDLM/"##") / SHARP KEYDLM           => "DSTRNG" ;
   KEYDLM -> "<" / ">" ;

   DELIMITER -> ":" / ";" / "(" / ")" / "."              => "DELMTR";

   OPERATOR -> ("-" / ">" / "/" / "="
                / "?" / "*" / "+")+                      => "OPRATR" ;

   COMMENT -> SHARP (NOT 'EOL')* 'EOL' ;

   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" ;
   KDIGIT -> "0"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9" ;

   DQ -> '"' ; SQ -> '#'' ; SHARP -> '##' ;
   NOTDQSH -> NOT("##"/"#"") ; NOTSQSH -> NOT("##"/"'") ;

   KC -> NOT("") ;

   END FSCAN.
```

```
TOKENS
   "IDENT" "NUMBER" "SCONST"
   'AND' 'ARRAY' 'BEGIN' 'CASE' 'CONST' 'DIV' 'DO' 'DOWNTO' 'ELSE'
   'END' 'FILE' 'FOR' 'FUNCT' 'GOTO' 'IF' 'IN' 'LABEL' 'MOD' 'NIL'
   'NOT' 'OF' 'PACKED' 'PROC' 'PROG' 'RECORD' 'REPEAT' 'SET' 'THEN'
   'TYPE' 'UNTIL' 'VAR' 'WHILE' 'WITH'
   'LPARN' 'RPARN' 'LBRKT' 'RBRKT'
   'ASGN' 'COLON' 'SCOLON' 'PD' 'COMMA' 'RANGE'
   'PLUS' 'MINUS' 'DIVD' 'MULT' 'LT' 'GT' 'LE' 'GE' 'EQ' 'NE' 'PNTR'

SCANNER PASCAL :
   PASCAL    -> (SPACES PASCAL1)* SPACES ;
   SPACES    -> (' '/'EOL')* ;

   SCANNER PASCAL1 :
      PASCAL1  -> NAME/NUMBER/SCONST/DELIMITER/OPERATOR/COMMENT ;
      END PASCAL1 ;
```

```
NAME  -> ALPHA (ALPHA/DIGIT)*        => "IDENT" (KEYWORD) ;

SCANNER KEYWORD :
   KEYWORD  -> <AND>              => 'AND'
            -> <ARRAY>            => 'ARRAY'
            -> <BEGIN>            => 'BEGIN'
            -> <CASE>             => 'CASE'
            -> <CONST>            => 'CONST'
            -> <DIV>              => 'DIV'
            -> <DO>               => 'DO'
            -> <DOWNTO>           => 'DOWNTO'
            -> <ELSE>             => 'ELSE'
            -> <END>              => 'END'
            -> <FILE>             => 'FILE'
            -> <FOR>              => 'FOR'
            -> <FUNCTION>         => 'FUNCT'
            -> <GOTO>             => 'GOTO'
            -> <IF>               => 'IF'
            -> <IN>               => 'IN'
            -> <LABEL>            => 'LABEL'
            -> <MOD>              => 'MOD'
            -> <NIL>              => 'NIL'
            -> <NOT>              => 'NOT'
            -> <OF>               => 'OF'
            -> <PACKED>           => 'PACKED'
            -> <PROCEDURE>        => 'PROC'
            -> <PROGRAM>          => 'PROG'
            -> <RECORD>           => 'RECORD'
            -> <REPEAT>           => 'REPEAT'
            -> <SET>              => 'SET'
            -> <THEN>             => 'THEN'
            -> <TYPE>             => 'TYPE'
            -> <UNTIL>            => 'UNTIL'
            -> <VAR>              => 'VAR'
            -> <WHILE>            => 'WHILE'
            -> <WITH>             => 'WITH' ;
   END KEYWORD;
```

```
ALPHA ->"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" ;

NUMBER -> DIGIT+ DECPART?*1
                     ("E" ("+"/"-")? DIGIT+)?        => "NUMBER" ;
SCANNER DECPART :
   DECPART  -> "." DIGIT+ ; END DECPART ;
DIGIT -> "0"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9" ;

SCONST -> DQT ( NOT(KQT/"EOL") / (DQT KQT) )* DQT    => "SCONST" ;
DQT -> '#'' ; KQT -> "'" ;

DELIMITER   -> '('         => 'LPARN'
            -> ')'         => 'RPARN'
            -> '['         => 'LBRKT'
            -> ']'         => 'RBRKT'
            -> ':' '='     => 'ASGN'
            -> ':'         => 'COLON'
            -> ';'         => 'SCOLON'
            -> '.'         => 'PD'
            -> ','         => 'COMMA'
            -> '.' '.'     => 'RANGE' ;

OPERATOR -> '+'          => 'PLUS'
         -> '-'          => 'MINUS'
         -> '/'          => 'DIVD'
         -> '*'          => 'MULT'
         -> '<'          => 'LT'
         -> '>'          => 'GT'
         -> '<' '='      => 'LE'
         -> '>' '='      => 'GE'
         -> '='          => 'EQ'
         -> '<' '>'      => 'NE'
         -> '^'          => 'PNTR' ;

COMMENT -> '(' '*' CMNTCHRS '*' ')' ;
CMNTCHRS -> (NOT('*') / ('*' NOTRP**0))* ;
SCANNER NOTRP :
   NOTRP -> NOT(')') ; END NOTRP ;

END PASCAL.
```

```
TOKENS
  "LBLFLD" "NAME" "DCONST" "LCONST" "RCONST" "DPCNST" "FIELD"
  'SCONST' 'HCONST'
  'KASSIG' 'KBACKS' 'KBLOCK' 'KCALL' 'KCLOSE' 'KCOMMO' 'KCONTI'
  'KDATA' 'KDO' 'KDIMEN' 'KELSE' 'KEND' 'KENDFI' 'KENTRY' 'KEQUIV'
  'KEXTER' 'KFUNCT' 'KFORMA' 'KGO' 'KIF' 'KIMPLI' 'KINQUI' 'KINTRI'
  'KOPEN' 'KPARAM' 'KPAUSE' 'KPRINT' 'KPROGR' 'KREAD' 'KRETUR'
  'KREWIN' 'KSAVE' 'KSTOP' 'KSUBRO' 'KTHEN' 'KTO' 'KWRITE'
  'KINTEG' 'KREAL' 'KDOUBL' 'KPRECI' 'KCOMPL' 'KLOGIC' 'KCHARA'
  'COMMA' 'EQUALS' 'COLON' 'LPAREN' 'RPAREN'
  'LE' 'LT' 'EQ' 'NE' 'GE' 'GT' 'AND' 'OR' 'EQV' 'NEQV' 'NOT'
  'ASTRSK' 'DBASTR' 'PLUS' 'MINUS' 'SLASH' 'CONCAT'
  'EOS' 'ERROR'

SCANNER FORTRAN77 :

   FORTRAN77 -> (COMMENT ELSE ENDSTMT ELSE ULSTMT)* ;

   SCANNER COMMENT :
      COMMENT -> ('C' / '*') DC**79
              -> SBLANK**80 ;
      SCANNER SBLANK : SBLANK -> ' ' ; END SBLANK ;
      END COMMENT ;

   SCANNER ENDSTMT :
      ENDSTMT -> LABELF BLANKCF (' '*) FEND EOS ; END ENDSTMT ;

   SCANNER ULSTMT :
      ULSTMT -> BSTMT CSTMT**() => (FORTRANSTMT) ;
      END ULSTMT ;

   BSTMT -> KC**72 DC**8 ;

   SCANNER CSTMT :
      CSTMT -> (COMMENT ELSE CLINE) ;
      END CSTMT ;

   SCANNER CLINE :
      CLINE -> SBLANK**5 NOT(' '/'0') KC**66 DC**8 ;
      SCANNER SBLANK : SBLANK -> ' ' ; END SBLANK ;
      END CLINE ;

   SCANNER DC :
      DC -> NOT('') ; END DC ;
   SCANNER KC :
      KC -> NOT("") ; END KC ;
```

```
SCANNER FORTRANSTMT :
   FORTRANSTMT -> LABELF BLANKCF (' '*)
                       (ASGN ELSE STMT) TEXT EOS  ;
   END FORTRANSTMT ;

LABELF -> KC**5 => (SCANLBL) ;
SCANNER SCANLBL :
   SCANLBL -> (' '*)  LBLFLD? ;
   LBLFLD -> DIGIT+                                   => "LBLFLD" ;
   END SCANLBL ;

BLANKCF -> (' '/'0')
        -> NOT (' '/'0')                              => 'ERROR' ;

SCANNER ASGN :
   ASGN -> NAME PARENS? PARENS? EQUALS EXPR ANYTRAP ;
   ANYTRAP -> (NOT'' '')? ;
   END ASGN ;

# STMT MATCHES ALL FORTRAN STATEMENTS WITH KEYWORDS
# I.E. ALL STATEMENTS EXCEPT ASSIGNMENT AND STATEMENT FUNCTION DEF'S.
  SCANNER STMT:
   STMT -> BACKSPACE/BLOCK DATA/CONTINUE/DIMENSION/ENDFILE
           /EQUIVALENCE/EXTERNAL/FUNCTION/INQUIRE/INTRINSIC
           /PARAMETER/PROGRAM/SUBROUTINE
           /FELSE/FEND IF
           /DO/CALL/CLOSE/COMMON/DATA/ENTRY/GO TO/OPEN/PAUSE/PRINT
           /READ/RETURN/REWIND/SAVE/STOP/WRITE/TYPE(FCN ELSE NULL)
        -> ASSIGN LABEL TO
        -> FELSE IF PARENS THEN
        -> FORMAT FORMATSPEC
        -> IF PARENS (ASGN ELSE SCTHEN ELSE STMT ELSE NULL)
        -> IMPLICIT ((TYPE PARENS) // COMMA) ;


   TYPE -> INTEGER/REAL/DOUBLE PRECISION/COMPLEX/LOGICAL
           /CHARACTER (ASTRSK LENSPEC)? ;
   SCANNER LENSPEC :
      LENSPEC -> ICONST/PARENS ; END LENSPEC ;

   LABEL -> DIGIT+                                    => "DCONST" ;

   END STMT;
```

```
SCANNER TEXT:
   TEXT -> (NAMLITOP/SEPARATOR/LPAREN/RPAREN)* ;
   END TEXT;

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

SCANNER EXPR :
   EXPR -> (NAMLITOP/PARENS)+ ; END EXPR ;

SCANNER NAMLITOP:
   NAMLITOP -> NAME/LITERAL/OPERATOR ;
   END NAMLITOP ;

SCANNER FCN:
   FCN -> FUNCTION ;
   END FCN;

SCANNER SCTHEN :
   SCTHEN -> THEN ; END SCTHEN ;

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

EOS -> ()                                            => 'EOS' ;
```

```
ASSIGN          -> A S S I G N                      => 'KASSIG' ;
BACKSPACE       -> B A C K S P A C E                => 'KBACKS' ;
BLOCK           -> B L O C K                        => 'KBLOCK' ;
CALL            -> C A L L                          => 'KCALL' ;
CHARACTER       -> C H A R A C T E R                => 'KCHARA' ;
CLOSE           -> C L O S E                        => 'KCLOSE' ;
COMMON          -> C O M M O N                      => 'KCOMMO' ;
COMPLEX         -> C O M P L E X                    => 'KCOMPL' ;
CONTINUE        -> C O N T I N U E                  => 'KCONTI' ;
DATA            -> D A T A                          => 'KDATA' ;
DIMENSION       -> D I M E N S I O N                => 'KDIMEN' ;
DO              -> D O                              => 'KDO' ;
DOUBLE          -> D O U B L E                      => 'KDOUBL' ;
FELSE           -> E L S E                          => 'KELSE' ;
FEND            -> E N D                            => 'KEND' ;
ENDFILE         -> E N D F I L E                    => 'KENDFI' ;
ENTRY           -> E N T R Y                        => 'KENTRY' ;
EQUIVALENCE     -> E Q U I V A L E N C E            => 'KEQUIV' ;
EXTERNAL        -> E X T E R N A L                  => 'KEXTER' ;
FORMAT          -> F O R M A T                      => 'KFORMA' ;
FUNCTION        -> F U N C T I O N                  => 'KFUNCT' ;
GO              -> G O                              => 'KGO' ;
IF              -> I F                              => 'KIF' ;
IMPLICIT        -> I M P L I C I T                  => 'KIMPLI' ;
INQUIRE         -> I N Q U I R E                    => 'KINQUI' ;
INTEGER         -> I N T E G E R                    => 'KINTEG' ;
INTRINSIC       -> I N T R I N S I C                => 'KINTRI' ;
LOGICAL         -> L O G I C A L                    => 'KLOGIC' ;
PARAMETER       -> P A R A M E T E R                => 'KPARAM' ;
PRECISION       -> P R E C I S I O N                => 'KPRECI' ;
OPEN            -> O P E N                          => 'KOPEN' ;
PAUSE           -> P A U S E                        => 'KPAUSE' ;
PRINT           -> P R I N T                        => 'KPRINT' ;
PROGRAM         -> P R O G R A M                    => 'KPROGR' ;
READ            -> R E A D                          => 'KREAD' ;
REAL            -> R E A L                          => 'KREAL' ;
RETURN          -> R E T U R N                      => 'KRETUR' ;
REWIND          -> R E W I N D                      => 'KREWIN' ;
SAVE            -> S A V E                          => 'KSAVE' ;
STOP            -> S T O P                          => 'KSTOP' ;
SUBROUTINE      -> S U B R O U T I N E              => 'KSUBRO' ;
THEN            -> T H E N                          => 'KTHEN' ;
TO              -> T O                              => 'KTO' ;
WRITE           -> W R I T E                        => 'KWRITE' ;
```

```
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' (' '*) ; VK -> "V" (' '*) ;
W  -> 'W' (' '*) ;
X  -> 'X' (' '*) ; XK -> "X" (' '*) ;
Y  -> 'Y' (' '*) ;


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") (' '*) ;
```

```
# 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)
            ->  DECPTACONST/LCONST/STRCNST ;
   SCANNER SCANACONST :
      SCANACONST -> ACONST ; END SCANACONST ;
   SCANNER SCANICONST :
      SCANICONST -> ICONST (LCONST/OPERATOR) ?
                 -> DIGIT+ EK SIGN? DIGIT+          => "RCONST"
                 -> DIGIT+ DK SIGN? DIGIT+          => "DPCNST" ;
      END SCANICONST ;

   ICONST -> DIGIT+                                   => "DCONST" ;
   ACONST -> DIGIT+ POINT DIGIT* (EK SIGN? DIGIT+)?   => "RCONST"
          -> DIGIT+ POINT DIGIT* DK SIGN? DIGIT+      => "DPCNST" ;
   DECPTACONST -> POINT DIGIT+ (EK SIGN? DIGIT+)?     => "RCONST"
               -> POINT DIGIT+ DK SIGN? DIGIT+        => "DPCNST" ;
   POINT -> "." (' '*) ;
   SIGN  -> ("+"/"-") (' '*) ;

   STRCNST -> APOST
              (NOT(APOST) / (APOST APOST))+
              APOST  (' '*)                           => 'SCONST' ;
   APOST   -> '#'' ;

   HCONST     -> LENGTH 'H' HCONSTVAL ;
   LENGTH     -> DIGIT+                               => (=) ;
   HCONSTVAL  -> DC**LENGTH (' '*)                    => 'HCONST' ;
```

```
LCONST    -> DOT TK RK UK EK DOT                          => "LCONST"
          -> DOT FK AK LK SK EK DOT                       => "LCONST" ;

SEPARATOR -> COMMA/EQUALS/COLON ;

OPERATOR -> DOT LK EK DOT                                 => 'LE'
         -> DOT LK TK DOT                                 => 'LT'
         -> DOT EK QK DOT                                 => 'EQ'
         -> DOT NK EK DOT                                 => 'NE'
         -> DOT GK EK DOT                                 => 'GE'
         -> DOT GK TK DOT                                 => 'GT'
         -> DOT AK NK DK DOT                              => 'AND'
         -> DOT OK RK DOT                                 => 'OR'
         -> DOT EK QK VK DOT                              => 'EQV'
         -> DOT NK EK QK VK DOT                           => 'NEQV'
         -> DOT NK OK TK DOT                              => 'NOT'
         -> ASTRSK
         -> DBASTRSK
         -> PLUS
         -> MINUS
         -> SLASH
         -> DBSLASH ;

PLUS    -> '+' (' '*)                                     => 'PLUS' ;
MINUS   -> '-' (' '*)                                     => 'MINUS' ;
ASTRSK -> '*' (' '*)                                      => 'ASTRSK' ;
DBASTRSK  -> '*' (' '*) '*' (' '*)                        => 'DBASTR' ;
SLASH   -> '/' (' '*)                                     => 'SLASH' ;
DBSLASH   -> '/' (' '*) '/' (' '*)                        => 'CONCAT' ;
LPAREN -> '(' (' '*)                                      => 'LPAREN' ;
RPAREN -> ')' (' '*)                                      => 'RPAREN' ;
EQUALS -> '=' (' '*)                                      => 'EQUALS' ;
COMMA   -> ',' (' '*)                                     => 'COMMA' ;
COLON   -> ':' (' '*)                                     => 'COLON' ;

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

```
SCANNER FORMATSPEC :
   FORMATSPEC -> LPAREN
      (FIELD/SLASH* /COLON*) // (COMMA/SLASH* /COLON*)
                 RPAREN ;
   SCANNER FIELD:
      FIELD -> HCONST/STRCNST/NHDESC/(ICONST? FORMATSPEC) ;
      END FIELD ;

   COMMA -> ',' (' '*) ;

   NHDESC -> DIGIT* A DIGIT*                          => "FIELD"
          -> DIGIT* L DIGIT+                          => "FIELD"
          -> DIGIT* I DIGIT+ (POINT DIGIT+)?          => "FIELD"
          -> SCALE? FDEG
          -> SCALE? FDEG
          -> T LR? DIGIT+                             => "FIELD"
          -> DIGIT+ X                                 => "FIELD"
          -> S PS?                                    => "FIELD"
          -> SCALE
          -> B NZ                                     => "FIELD" ;
   SCALE  -> SIGN? DIGIT+ P                           => "FIELD" ;
   FDEG   -> DIGIT* FD DIGIT+ POINT DIGIT+            => "FIELD"
          -> DIGIT* EG DIGIT+ POINT DIGIT+ (EK DIGIT+)?
                                                      => "FIELD" ;

   SIGN -> ("+"/"-") (' '*) ;

   A  -> "A" (' '*) ;
   L  -> "L" (' '*) ;
   I  -> "I" (' '*) ;
   FD -> ("F"/"D") (' '*) ;
   EG -> ("E"/"G") (' '*) ;
   T  -> "T" (' '*) ;
   LR -> ("L"/"R") (' '*) ;
   X  -> "X" (' '*) ;
   S  -> "S" (' '*) ;
   PS -> ("P"/"S") (' '*) ;
   P  -> "P" (' '*) ;
   B  -> "B" (' '*) ;
   NZ -> ("N"/"Z") (' '*) ;

   END FORMATSPEC ;

END FORTRAN77.
```

Appendix E: Interpreter Size and Speed

All size and speed measurements were done on a VAX 11/780 using the f77 compiler. All numbers are given in decimal.

1. Size

1.1. Interpreter Size

The FSCAN interpreter consists of :

8.5k bytes code
1.5k bytes data

1.2. Table Sizes

The object code or tables for the programs listed in Appendix D require :

1.2.1. Fscan

1.5k bytes

1.2.2. Pascal

5k bytes

1.2.3. Fortran77

15k bytes + 9k bytes increased interpreter data space

2. Speed

Following are timing measurements on the FORTRAN lexical analyzer produced from the specification in Appendix D.

(1) Standard interpreter
16 lines/second

(2) Interpreter (1) with FORTRAN READ statement replaced with a call to a 'read-line' routine written in C :
22 lines/second

(3) Interpreter (2) with most primitive function calls macro expanded in line :
52 lines/second

(4) Interpreter (3) with logical left and right shift primitives done in line :
80 lines/second (est.)