LGS: A Lexical Analyzer Generator

J. Wick

Research Report #44

J.Wick 18-May-74

A Lexical Analyzer Generator

1. Introduction

LGS is a program that generates lexical analyzers for languages defined by regular grammars. It takes as input a BNF description of the language and produces an IMP10 program which will parse an input string into the tokens of the language. Sample input and output files may be found in appendix 1 and 2.

2. Input

Input to the program is a single file containing statements in an augmented BNF language. Each allowable statement is described below; for those who desire a more formal specification, appendix 1 contains the syntax for symbols, and appendix 3 contains the syntax for syntax.

2.1 Lexical Conventions

The lexical conventions used by this program are exactly those defined in appendix 1 (and appendix 2). This definition will become clear as the input is described; briefly, tokens and their values are:

1) An operator (opr) is any special character; its value is its ASCII equivalent, right justified (for example, "?" has the value 63 in decimal). Certain special operators (like ":", "=", and ";") are reserved for defining syntax and hence have no values; if you wish to input one of these operators as a character value, use a string.

2) A name (nam) is a letter followed by any number of letters or digits. The value of a name, when it stands for itself and not some terminal or non-terminal symbol, is its right-justified ASCII string equivalent. For example, the name "A" has value 101B, and "AB" is 20302B. If the string is longer than five characters, the right-most five characters are used.

3) A number (num) is a digit followed by any number of letters or digits. Decimal values are represented by a string of digits. Octal numbers are followed by the letter B or b (12b, for example, is an ASCII line feed). Numbers in other bases end in Bn (or bn), where n is a decimal number (these are called flexadecimal). The letters and digits to the left of the last

B or b are interpreted as a constant in base n (for example, 2ABb16 is the base 16 constant 2AB (683 decimal) and 10100B2 is the binary constant 10100, or 20 decimal). The base may be arbitrarily large, but only digits and the letters A-Z (or a-z) may be used (A-Z represent the digits 10-35). Note that a constant starting with a letter is not legal: ABb12 is a name; the desired constant must be written 0ABb12.

4) A string (stg) is any sequence of characters enclosed in (single) quote marks. '' is the null string and ' within a string is represented by ''. An end-of-file also terminates a string (but is not a part of it). Control characters (ASCII codes less than 40b) may appear within strings but are ignored; a number should be used to represent a control character. The value of a string (like a name) is its right-justified ASCII equivalent, ignoring the delimiting quotes.

5) A comment begins with a left curly bracket ("{") and ends with a right curly bracket ("}"). An end-of-file also terminates a comment but is not a part of it. Comments are ignored and have no value.

Note: the maximum length of any token except comments is eighty characters, excluding delimiters like quote and blank; characters past the eightieth are ignored.

2.2 Parameter Definitions

Parameters are variables which define compile time constants in the generated program (such as input character size, word length, or number base). The format of a parameter definition statement is

```
stg=num;
```

where "stg" is the name of the parameter enclosed in quotes and "num" is its value. Currently, the following parameters are recognized:

'CHARSIZE'=7;	The character size in bits (see section 2.3 on handling character sets other than ASCII).
'TOKSIZE'=80;	The maximum number of characters in a token (this should be an integer multiple of the number of characters per 36 bit word).

The values given above are the defaults. A parameter may be defined at most once in the input file; definitions other than the first are ignored. If no definition appears, the default value is used.

2.3 Type Definitions

Normally in regular grammars each character is a terminal symbol standing for itself; however, in order to speed up the analyzer and save writing, each character is assigned a type standing for one or more characters (such as letters or digits). This is accomplished by productions of the form:

```
        terminal ::= char | char | ... | char;
```

where "terminal" is a name of your choosing standing for the type, and each "char" is a name, a number or a string which has a value in the range zero to (2^CHARSIZE)-1 (evaluates to a single character). For example, the productions

```
        mrk ::= '<' | '>' | ':' | '=' | 12B;
        dig ::= '0' | '1' | '2' | ... | '9';
        let ::= 'A' | 'B' | 'D' | ... | 'Z';
```

define the types "mrk" (marks: values 74B, 76B, 72B, 75B, and 12B), "dig" (digits: values 60B through 72B), and "let" (letters; values 101B through 132B). Note: the "..." operator is used here for abbreviation but is not allowed in the input.

It is possible, though not especially easy, to process character sets other than ASCII by specifying each "char" as a number, rather than a name or a string. For example, if the character set were EBCDIC, the following statements might be applicable:

```
                    'CHARSIZE'=8;

   mrk ::= 4Cb16 {<} | 6Eb16 {>} | ... | 25b16 {LF};
   dig ::= 240 | 241 | 242 | 243 | ... | 249;
   let ::= 0C1b16 | 0C2b16 |0C3b16 | ... | 0E9b16;
```

Initially all characters are of type "ign" (value 0); therefore all characters which do not appear in type definitions are assumed to be of type "ign". This type usually includes blanks, most of the control characters, and any special characters not used in the language. See section 5.1 for more information about "ign".

2.4 Token Definitions

The remaining productions in the input file are used to define the transitions of a finite state machine which recognizes tokens of the regular grammar. Each non-terminal symbol of the language represents a state of the machine, and each production represents a transition. A set of states and transitions defines a token. In addition, a special state (called the initial state) is pre-defined as state zero. Productions are interpreted as follows:

1) <non-terminal> ::= terminal;
   The terminal must be a type name (see 2.3). A transition is constructed from the initial state to state <non-terminal> under input characters

of type terminal. The interpretation is that
when in the initial state (no token is being
constucted), we can begin to build up a token
of type <non-terminal> when a character of type
terminal is read.

2) <non-terminal-1> ::= <non-terminal-2> terminal;
The terminal must be a type name (see 2.3). A
transition is constructed from state <non-terminal-2>
to state <non-terminal-1> under input characters
of type terminal. The interpretation is that
when in state <non-terminal-2>, we can construct
a token of type <non-terminal-1> by adding a
character of type terminal to what has already
been built up.

3) <non-terminal> ::= alt | alt | ... | alt;
Each alt must be of the same form as the right part
of 1) or 2). This is abbreviation only; each alt is
handled separately as above.

For example, given the types "mrk", "dig", and "let" defined in the example in 2.3, the following productions define tokens of type operator, number and name (similar to appendix 1):

```
<opr> ::= mrk;
<num> ::= dig | <num> dig;
<nam> ::= let | <nam> let | <nam> dig;
```

These productions, using the rules defined above, result in the following finite state machine (given in tabular form):

Character Type

State	ign	mrk	dig	let
<init>		PR <opr>	PR <num>	PR <nam>
<opr>				
<num>			PR <num>	
<nam>			PR <nam>	PR <nam>

The capital letters in the state diagram represent actions that are performed before the transition is made to a new state; they are included so that the finite state machine will constuct tokens, rather than just recognize them. Tokens are built up in a buffer one character at a time. The possible actions are Read (read the next input character), Pack (add a character to the buffer), reTurn (return the buffer as a token), and Clear (clear the buffer). The action sequence "Pack,Read" is appropriate for the above transitions because of the interpretation placed on each of the productions.

Several possible transitions have been left unspecified; for example, if the current state is <nam> and the input character is type "mrk", what is to happen? The problem arises because the BNF specification is non-deterministic; the generator contains an algorithm for converting this to a deterministic machine, as follows:

1) The initial state is filled in first. If no transition out of the initial state is specified for a character type (for example, type "ign" above), then no token may begin with that type of character. The character is ignored by filling in action "Read" and a transition back into the initial state.

2) All other unspecified transitions mean that the new input character cannot be added to the current token. In fact, the current token has been completed, so actions "reTurn,Clear" are filled in. We could now go to the initial state to find out what to do with the new input, but it is more efficient to go directly to the state which starts the next token. This is accomplished by copying the entry which appears in the initial state for the new character type. For example, if in state <nam> with input "mrk", actions "reTurn,Clear" are filled in to return the name constructed, and the initial state entry for type "mrk" is added, giving the result TCPR <opr>.

The completed state diagram is as follows:

Character Type

S		ign	mrk	dig	let
t	<init>	R <init>	PR <opr>	PR <num>	PR <nam>
a	<opr>	TCR <init>	TCPR <opr>	TCPR <num>	TCPR <nam>
t	<num>	TCR <init>	TCPR <opr>	PR <num>	TCPR <nam>
e	<nam>	TCR <init>	TCPR <opr>	PR <nam>	PR <nam>

This machine will now construct all sequences of tokens in this simple language. To add flexibility, there are exceptions to all of the rules given above for determining actions and filling in unspecified transitions; these are discussed in section 4.

A few more useful details: the generated program also returns the type and character length of each token. Types are determined by the number of the current state when the token is stored. State numbers are assigned to non-terminals in the order they are defined (not referenced) starting at one (zero is reserved as the initial state number). In the above example, operators will be returned as type one, numbers as type two, and names as type three.

3. Output

LGS produces an IMP10 program which parses the tokens of the language; after initialization, it returns one token each time it is called. Several compile-time variables are defined in the program to add flexibility; they include the wordsize (WORD), charactersize (CHAR), and maximum number of characters in a token (TOKSIZE). The following global symbols are also defined or referenced:

1) ILEX(): A function with no parameters which must be called to initialize the analyzer. It may not be called after a physical end-of-file has been read unless the file has been closed and a new (possibly the same) file has been opened (see 5.2 for further details on handling end-of-file conditions).

2) GET(CHR): A function provided by the user which returns the next character of the input file in CHR (right-justified and zero-filled); its value is non-zero if physical end-of-file was read (CHR is ignored) and zero otherwise.

3) LEX(): A function with no parameters and zero value which is called each time a token is desired; it assigns a meaningful value to TOK, destroying the old value.

4) TOK: A global vector defined by LEX of word length 3+TOKSIZE/(WORD/CHAR). The first word contains the token type (a state number), the second word contains the length of the token (in characters); subsequent words contain the token, terminated by enough zero characters (at least one) to complete the last word of the token.

4. Semantics

As stated in section 2.4, there are exceptions to all of the rules for assigning actions and filling in unsecified transitions; special semantic actions may be associated with token definitions to add flexibility to the output of the lexical analyzer. They are enclosed in square brackets and may appear in productions as any combination of the following:

1) <non-terminal-1> ::= <non-terminal-2> terminal [S]; where S is one of NOP, OMIT or SAME. Normally, actions "Pack,Read" are associated with the transition from state <non-terminal-2> to state <non-terminal-1>. SAME turns off the read action, OMIT turns off the pack action, and NOP turns off both. OMIT is especially useful for omitting the delimiters of a token from the symbol returned, such as quotes surrounding strings (see the definition of <str> and <stg> in appendix 1).

2) <non-terminal-1> ::= <non-terminal-2> [S] terminal; where S is one of TOKEN, RETURN or CLEAR. Normally, no actions are performed on the token built up so far when the transition is made from state <non-terminal-2> to state <non-terminal-1>. CLEAR adds the clear action, RETURN adds the return action, and TOKEN adds both. Note that return and clear actions are always performed before pack or read.

3) <non-terminal> [S] ::= ......; where S is one of HOLD, IGNORE or KEEP. These semantics apply when filling in unspecified transitions out of state <non-terminal>. Normally actions "reTurn, Clear" are supplied; KEEP turns off clear, IGNORE turns off return, and HOLD turns off both. The initial state "init" is predefined with semantics IGNORE. For an example, see the definition of <cmt> in appendix 1.

5. Special Considerations

5.1 Ignoring Characters

As previously mentioned, a special character type "ign" (ignore) has been predefined, on the assumption that at least one character will be ignored by any lexical analyzer. It is possible, however, to define all characters with some other type, making the type "ign" superfluous; storage is allocated for this type anyway, so this is not recomended. Use the predefined type whenever possible.

The predefined type "ign" may be used anywhere a terminal symbol is required in a production; it is automatically ignored only in the initial state (unless a production indicates otherwise). Use the OMIT or IGNORE semantics to achieve the effect you want (see the definition of <str> and <cmt> in appendix 1 for an example).

5.2 End-of-file Handling

One special type, designated by the terminal symbol "eof", must be defined somewhere in the input file; it is predefined with semantics SAME (no read action). This is to insure that the lexical analyzer produced will never attempt to read past a (physical) end-of-file. If you wish to recognize only physical end-of-file as terminating the input, choose a single character not used elsewhere (null, rubout, ^Z or some other control character are the natural choices). For example, the productions

```
eof   ::= 177B;
<opr> ::= ... | eof | ... ;
```

mean that when physical end-of-file is reached, a token of type "opr" with value 177B will be returned. On the other hand, if some special characters also indicate (logical) end-of-file, they should be listed first in the type definition of "eof" (the rule is that the last alternate of the production is used to represent physical end-of-file). The productions

```
eof   ::= '%';
<opr> ::= ... | eof | ... ;
```

mean that when physical end-of-file is reached, or a percent-sign is read, a token of type "opr" with value 45B will be returned (the two cases are indistinguishable). But,

```
               eof   ::= '%' | 177B;
               <opr> ::= ... | eof | ... ;
```

means that when physical end-of-file is reached, type "opr" value 177B will be returned, but when a logical end-of-file (a percent-sign) is read, type "opr" value 45B will be returned. The important point is that in the second case, more tokens may be read from the same file, provided ILEX() (the initialization function) is called to clear the end-of-file condition.

5.3 The Initial State

The pre-defined state "init" may be used with caution wherever a non-terminal symbol is required in a production. It is especially useful in defining bracketed tokens, such as strings (bracketed by quotes) or comments (bracketed by left and right braces). For example,

```
     exc    ::= '!';
   <id>     ::= exc [OMIT] | <id> mrk | <id> let | <id> dig;
   <init> ::= <id> [TOKEN] exc [OMIT];
```

These productions allow an <id> to contain special characters (except "!") if it is surrounded by exclamation points. The danger here is that no path into the initial state (even by a round-about route) may have a "pack" action after the last "clear" action in the path. This would leave junk in the buffer which erroneously would become part of the next token. So the semantics TOKEN and OMIT are required here.

5.4 Re-typing Tokens and "null" Transitions

As mentioned above, the type returned with a token is determined by the current state when the token is stored. This may not always be convenient for the module which is processing the tokens (usually a parser). For example, suppose we want the lexical analyzer to recognize "/" as the division operator and "//" as the remainder operator. The natural definition is

```
                mrk   ::= '+' | '-' | '*';
                sla   ::= '/';
               <div> ::= sla;
               <opr> ::= mrk | <div> sla;
```

This has almost the desired effect, except that "/" is returned as type <div>, whereas all other operators are returned as type <opr>. This might be annoying to the parser, so a special production of the form

```
                    <opr> ::= <div>;
```

may be added to the set above. It causes a "null" transition to be constructed from state <div> to state <opr> with no actions whatsoever for every character type not defined by a production (in this case every character type except "sla"). Another example may be found in appendix 1; there a <str> followed by an "eof" is defined to be the same as a <stg> ("eof" is the only character type not included in a production). This is not equivalent to the production

```
<stg> ::= <str> eof;
```

which would make an "eof" part of the string.

6. Running the Generator

LGS is started by the standard RUN monitor command; it responds with an asterisk. Type a command of the form

```
*output=dev:fil.ext[prj,prg]
```

where "output" has the same format as the input filename. A left arrow (_) may be used in place of the equal sign. If no device is specified, DSK is assumed. If "output=" is omitted, the input filename is used with extension TMP.

7. Errors

Each error message contains a statement number (if applicable), an alternate number within the statement (if applicable), text indicating the nature of the error, and possibly a symbol from the input string (represented by "?" below). There are very few fatal errors (if a fatal error is detected, the generator cannot continue); an attempt is made to produce some semblance of a lexical analyzer for almost every syntactically correct input. However, if any messages appear, check the output very carefully (or correct the error and rerun).

%OPEN FAILURE
   Usually file not found. Re-enter the command.

?OUT OF CORE
   This error is fatal. Go buy some more memory.

SYNTAX ERROR AT "?"
   This error is fatal. Check section 2 and appendix 3.

CHAR NOT A DIGIT IN "?"
   A character in a number is not a digit or a letter.

DIGIT > BASE IN "?"
   The value of a digit in the number exceedes the base.

"?" IS NOT A PARAMETER
   See section 2.2 for a list of legal parameters.

"?" WAS PREVIOUSLY DEFINED
   The value of a parameter may be defined at most once.

"?" IS NOT A CHAR

The value of the character in a type definition is not between zero and (2^CHARSIZE)-1 inclusive.

"?" WAS PREVIOUSLY TYPED
The character in a type definition has already been given a type by a previous production. The first definition wins.

"?" IS NOT A TERMINAL
The terminal referenced is undefined; see section 2.3.

"?" IS NOT A NON-TERMINAL
The non-terminal referenced is undefined; see section 2.4.

"?" IS NOT SEMANTICS
See section 4 for a list of available semantics.

"?" SEMANTICS NOT APPLICABLE
See section 4 for the use of available semantics.

RIGHT PART ALREADY DEFINED
This state transition was defined by a previous production. The first definition wins.

NULL TO SELF
This production defines a null transition from a state to itself, which may result in an infinite loop.

RETURN FROM <init>
A "return" action occurs on a transition out of the initial state (this is a warning only).

NULL FROM <init>
A null transition has been specified out of the initial state (this is a warning only).

NO CLEAR INTO <init>
A "clear" action is missing on a transition into the initial state (this is a warning only).

PACK INTO <init>
A "pack" action occurs on a transition into the initial state (this is a warning only).

EOF NOT DEFINED
A definition of character type "eof" cannot be found.

NO TYPES
No character type definitions could be found.

NO STATES
No state definitions could be found.

EOF IS IGNORED
End-of-file has not been used in defining any token.

8. References

1. Johnson, et al. Automatic generation of efficient lexical processors using finite state techniques. Comm. ACM 11,12 (December 1968), 805-813.

2. Conway, M. E. Design of a separable transition-diagram compiler. Comm. ACM 6,7 (July 1963), 396-408.

3. Gries, D. Compiler Construction for Digital Computers. Wiley, New York, 1971, 49-83.

4. Hennie, F. C. Finite-state Models for Logical Machines. Wiley, New York, 1968, 1-223.

5. Meehan, J. Private communication. December, 1973.

6. Hoey, D. Private communication. April, 1974.

Appendix 1.

Sample Syntax for a Lexical Analyzer

```
{PGSLEX.SYN                                          30-Apr-74}

                    {Parser Generating System}
                      {Syntax for Symbols}

                  'CHARSIZE'=7;          'TOKSIZE'=80;

  mrk ::= '!' | '"' | '#' | '$' | '%' | '&' | '(' | ')' |
          '*' | '+' | ',' | '-' | '.' | '/' | ':' | ';' |
          '<' | '=' | '>' | '?' | '@' | '[' | '\' | ']' |
          '^' | '_' | '`' | '|' | '~' ;

  let ::=       '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' |
                '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' ;

  dig ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
          '8' | '9' ;

  blk ::= ' ';    qot ::= '''';
  lcb ::= '{';    rcb ::= '}';    eof ::= 177b;


<opr> ::= mrk | eof ;

<nam> ::= let | <nam> let | <nam> dig ;

<num> ::= dig | <num> let | <num> dig ;

<stg> ::= <str> qot [OMIT] | <str> ;

<str> ::= qot [OMIT] | <str> ign [OMIT] | <str> blk |
          <str> mrk | <str> let | <str> dig |
          <stg> qot | <str> lcb | <str> rcb ;

<cmt> [IGNORE] ::= lcb [OMIT] | <cmt> ign [OMIT] |
          <cmt> blk [OMIT] | <cmt> mrk [OMIT] |
          <cmt> let [OMIT] | <cmt> dig [OMIT] |
          <cmt> qot [OMIT] | <cmt> lcb [OMIT] ;

<init> ::= <cmt> [CLEAR] rcb [OMIT] ;
```

Appendix 2.

Program Produced from Sample Syntax

```
# LGS 2.3                                                  17-May-74 #

                       # LEXICAL ANALYZER #
                          CALL ME PGSLEX;


# PGSLEX.I10[22,54]=PGSLEX.SYN[22,54]          17-May-74   19:54 #


LET CHAR=7,WORD=36,TOKSIZ=80,EOFCHR=177B;
LET NT=9,MASK=4,RETURN=10B,CLEAR=4B,PACK=2B,READ=1B;

LET LTYP=TOK,LLEN=TOK[1],LVAL=TOK[2];
TOK IS 3+TOKSIZ/WORD/CHAR LONG,COMMON;

<ST> ::= ENTER ::= "GO TO [RT]; GO:0";
<ST> ::= LEAVE <EXP,A>
     ::= LOCAL L IN "RT_LOC(L); RETURN A; L:0";

SUBR ILEX() IS
  (STATE_0; LEN_0; ARC_READ;
   BP_BYTEP LVAL<CHAR,WORD>;
   LTYP_0; LLEN_0; LVAL_0;
   RT_LOC(GO); 0);

SUBR LEX() IS
  (ENTER;
   WHILE 1 DO
     (ARC AND READ =>
        (GET(CHR) => CHR_EOFCHR; TYPE_TYPES[CHR]);
      ARC_STATES[TYPE+STATE*NT];
      ARC AND RETURN =>
        (TP IS REGISTER; TP_BP;
         <+TP>_0 UNTIL (TP RS 30)<CHAR;
         TP IS RELEASED; LEAVE 0);
      ARC AND CLEAR =>
        (BP_BYTEP LVAL<CHAR,WORD>; LLEN_LEN_0);
      LTYP_STATE_ARC RS MASK;
      ARC AND PACK =>
        (LEN<TOKSIZ => (<+BP>_CHR; LLEN_LEN_LEN+1));
      #ARC AND READ => LIST(CHR)# 0);
   0);

# LET init=0,opr=1,nam=2,num=3,stg=4,str=5,cmt=6; #

STATES:
   DATA(001B,023B,043B,063B,001B,121B,141B,001B,022B);
   DATA(015B,037B,057B,077B,015B,135B,155B,015B,036B);
   DATA(015B,037B,043B,043B,015B,135B,155B,015B,036B);
   DATA(015B,037B,063B,063B,015B,135B,155B,015B,036B);
   DATA(015B,037B,057B,077B,015B,123B,155B,015B,036B);
   DATA(121B,123B,123B,123B,123B,101B,123B,123B,100B);
   DATA(141B,141B,141B,141B,141B,141B,141B,005B,026B);
```

```
LET ign=0,mrk=1,let=2,dig=3,blk=4,qot=5,lcb=6,rcb=7,eof=8;

TYPES:
   DATA(ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign);
   DATA(ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign);
   DATA(ign,ign,ign,ign,ign,ign,blk,mrk,mrk,mrk,mrk,mrk,mrk);
   DATA(qot,mrk,mrk,mrk,mrk,mrk,mrk,mrk,mrk,dig,dig,dig,dig);
   DATA(dig,dig,dig,dig,dig,dig,mrk,mrk,mrk,mrk,mrk,mrk,mrk);
   DATA(let,let,let,let,let,let,let,let,let,let,let,let,let);
   DATA(let,let,let,let,let,let,let,let,let,let,let,let,let);
   DATA(mrk,mrk,mrk,mrk,mrk,mrk,let,let,let,let,let,let,let);
   DATA(let,let,let,let,let,let,let,let,let,let,let,let,let);
   DATA(let,let,let,let,let,let,lcb,mrk,rcb,mrk,eof) %%%
```

Appendix 3.

```
{LGSSYN.SYN                                              17-May-74}

                {Lexical Analyzer Generating System}
                        {Syntax for Syntax}


            'HASHSIZE'=13;    num = .opr;     stg = .opr;

.opr = 1;         .nam = 2;          .num = 3;          .stg = 4;

<prg> ::= <stl> 177b [LEXEND] ;

<stl> ::= <stm> ';' [LEXTAX] | <stl> <stm> ';' [LEXTAX] ;

<stm> ::= .stg [LEXPRM] '=' .num [LEXSET] |
          .nam [LEXTYP] ':' ':' '=' <typ> |
          '<' .nam [LEXDEF] '>' ':' ':' '=' <def> |
          '<' .nam [LEXDEF] '>' '[' .nam [LEXDSM] ']'
              ':' ':' '=' <def> ;

<typ> ::= <lit> | <typ> '|' <lit> ;

<lit> ::= .opr [LEXVAL] | .nam [LEXVAL] |
          .num [LEXVAL] | .stg [LEXVAL] ;

<def> ::= <alt> [LEXALT] | <def> '|' <alt> [LEXALT];

<alt> ::= <trm> | <ntm> | <ntm> <trm> ;

<ntm> ::= '<' .nam [LEXNTM] '>'  |
          '<' .nam [LEXNTM] '>' '[' .nam [LEXNSM] ']';

<trm> ::= .nam [LEXTRM]  |
          .nam [LEXTRM] '[' .nam [LEXTSM] ']';
```