Published: 10/25/68
(Supersedes: BZ.8.04, 08/29/68)

Identification

Lexical Analysis
J. D. Mills

Purpose

Lexical analysis is the vehicle by which the PL/1 compiler reads its input, a PL/1 source program, from its environment which is Multics. The primary task of lexical analysis is to translate its input from a character string and produce as output a string of operators, identifiers, constants, and delimiters (these are called "tokens"). This string of tokens becomes the input to the parse. Taken together then, lexical analysis and the parse form the syntactic analysis phase, which is the first phase, of the PL/1 compiler.

Lexical analysis also performs the secondary task of preparing the source program listing (if the list option is on).

Definition of Tokens

Defined below are the tokens which the lexical analyzer recognizes.

Operators

The syntax of PL/1 has no unit called operator but the notion exists and is useful. Thus, we identify the set of operators as:

{+, -, *, /, **, ^, =, ^=, <, >, <=, >=, ^>, ^<, |, ||, &}

Delimiters

Again, the term delimiter is not in the PL/1 syntax but sort of falls out. Note that though the space does in fact delimit, it is not useful to identify it as a delimiter. The set of delimiters is:

{:, (,), ->, ., ;, ,}

The comma is a delimiter and is underlined here to distinguish it from the set-notational comma.

Identifiers

Identifiers are recognized according to the PL/1 syntax rules.

Constants

The various forms of constants listed below are defined in the syntax. Note that for strings the lexical analyzer applies the replication factor, if present, to the string before passing it to the parse as a token.

bit string	
character string	isub
floating binary	imaginary floating binary
floating decimal	imaginary floating decimal
binary integer	imaginary binary integer
decimal integer	imaginary decimal integer
fixed binary	imaginary fixed binary
fixed decimal	imaginary fixed decimal

Implementation

Input

Lexical analysis addresses the input segment as a synchronous, packed array of characters. The array is based on the pointer to the source segment.

Interface with the Parse

The lexical analyzer must be initialized before its first use. This is accomplished by the call:

```
call      lex$initiate(seg_ptr,length);
```

where the arguments are defined:

```
dcl       seg_ptr ptr,              /*pointer to the source
                                      segment*/

          length fixed bin;         /*number of characters
                                      in the source segment*/
```

The parse calls lexical analysis once for each source statement to be translated. The calling sequence is:

```
call      lex;
```

The results of lexical analysis are contained in the external static variables defined by this declaration:

```
dcl  (token_list(1000)ptr,          /*pointers to token
                                      nodes*/

      statement_id fixed bin(31)),  /*a coded number repre-
                                      senting the line
                                      number and statement
      external static;                number of the
                                      statement being
                                      parsed*/
```

Each pointer in the token list points to a node representing the source token recognized. The declaration for the token node is:

```
dcl  1 token_table based(p),

     2 node_type fixed bin(15),     /*identifies the node
                                      as a token*/

     2 size fixed bin(15),          /*number of characters,
                                      size=n*/

     2 context ptr,                 /*used during dcl
                                      processing*/

     2 declaration ptr,             /*used during dcl
                                      processing*/

     2 next ptr,                    /*ptr to next token
                                      in token table*/

     2 type fixed bin(15),          /*type of token -
                                      identifier, bit string,
                                      etc.*/

     2 string char(n);              /*source language
                                      representation*/
```

Method

The lexical analyzer reads input characters until a semi-colon is encountered, whereupon it returns to its caller with the array of token pointers. PL/1 comments are ignored except for writing onto the source listing.

The recognition of tokens is accomplished by an approximation to a finite-state machine. The token grammar is not quite finite state so various programming devices are added to the FSM. Since the lexical analyzer must produce output as well as recognize tokens various actions are attached to the state transitions in the FSM. These actions result in the concatenation of the individual characters from the input until the complete token is built up. It is not desirable to convert numerical constants at this point because the context of the constant may influence its conversion.

The token table is hash-coded. A hash function of the token gives an index into an array of pointers. Each pointer in the array is null, or points to a list of tokens. By ordering the tokens on lists according to size and choosing the hash function so that no two single-character tokens appear on the same list the search for those tokens is ultra-rapid.

Subsidiary Entries

Other entries are provided, not for lexical analysis, but for the use of the token table maintaining apparatus.

Token Insertion

Tokens are inserted into the hash-coded token table with the procedure "insert_token" whose calling sequence is:

```
call  insert_token(p, string, type);
```

The arguments are defined by the declaration:

```
dcl  p ptr,                          /*insert token returns
                                       with p pointing at the
                                       token node*/

     string char(n) varying,         /*the token to be
                                       inserted*/

     type fixed bin(15);             /*the type of the token
                                       being inserted*/
```

Multiple Declaration Scanning

At a later stage of compilation when all declarations have been made the token nodes for all declared identifiers contain pointers to lists of symbol table nodes representing the declarations. There are circumstances under which

an identifier may legally have more than one declaration, e.g., when the declarations are in different blocks. The entry "insert_token$scan_token_table" is provided to search the entire token table for cases when multiple declarations have been made for a declaration outside the set of permissible situations.

The call is:

```
call insert_token$token_table;
```

The only results are the issuing of diagnostics for illegal cases.