TO: Distribution

FROM: Gary C. Dixon

DATE: June 20, 1974

SUBJECT: The reduction_compiler and lex_string_

This MTB describes the reduction language outlined in MTB-080, and provides writeups for the reduction_compiler command and for the lex_string_ subroutine.

The reduction compiler compiles the BNF-like statements of the reduction language into the syntax analyzer of a compiler. By coupling this syntax analyzer with the lexing functions of lex_string_, and adding some simple-to-program action subroutines which perform the semantic analysis function, it is possible to write a moderately sophisticated compiler in one or two man days. The basic portion of the reduction_compiler itself was written in two man days. The reduction statements defining the reduction language are attached to show how simply and clearly a compiler language can be defined by reductions.

You comments on possible extensions or modifications of the reduction_compiler or of lex_string_ would be appreciated. Please mail your comments to GDixon.PDO on the MIT Multics.

Names: reduction_compiler, rdc

Often in the course of programming, it becomes necessary to define a new language, and to write a compiler, interpreter, or other form of translator for the language. Examples of such languages in the Multics system include exec_com control language, runoff control language, the language used in the input segments for set_search_rules, the input language for the error_table_compiler, the binding control language used in bind segments, and of course the programming languages (PL/I, Fortran, ALM, etc). Some of these newly-developed languages will be used heavily, and thus deserve specially-designed translators which are optimized for that particular language. However, many new languages are developed as part of tools which will be used infrequently. For such languages, there is more need for simple translators which are easy to write, to understand and maintain, and to extend than there is a need for optimal, special-purpose translators. The reduction_compiler provides a facility for converting the syntax and semantics of a new language, as defined by a set of reductions, into a simple, standardized, easy to understand, and moderately efficient piece of PL/I code.

Usage

```
reduction_compiler  segment_name  -ctl_arg-
```

1) segment_name — is the path name of the translator source segment containing the reductions to be compiled. If the final entry of this path name does not end with a suffix of .rd, then .rd is assumed.

2) ctl_arg — may be one of the following optional control arguments.

-long, -lg — all error messages will include a detailed description of the error which has occurred. The default is to print the detailed description the first time an error occurs, and brief descriptions thereafter.

-brief, -bf all error message will include only a brief description of the error which has occurred. The default is to print a detailed description the first time each error occurs, and brief descriptions thereafter.

## The Translator

A translator source segment which is to be compiled by the reduction_compiler (rdc) should be organized as shown in Figure 1. The translator source segment (hereinafter called translator.rd) contains: a copyright notice or other PL/I comments (optional); a set of reduction statements and reduction attribute declarations - the delimiter /*++ opens the set of reductions, which are closed by the delimiter ++*/; a PL/I procedure statement for the translator; PL/I declarations for the translator's variables; a PL/I declaration for an error_control_table, containing the text of error messages to be generated by the translator (optional); a PL/I call statement invoking the lex_string_ subroutine to parse the translator's character string input into tokens; a PL/I call statement invoking the SEMANTIC_ANALYSIS subroutine which contains the translation code generated by rdc; a PL/I return statement; one or more PL/I function subprograms which are relative syntax functions (optional); and one or more PL/I subroutines which assign semantic meaning to the legal phrases in the input. Each of these parts of a translator is described further in the sections which follow.

The translator is compiled by a two-step process, as illustrated in Figure 2. First, translator.rd is compiled by rdc to generate a PL/I source segment (hereinafter called translator.pl1). translator.pl1 contains: a heading which identifies the translator source segment, the version of rdc used to compile that source segment into the PL/I segment, and the date and time of compilation; followed by the contents of translator.rd; followed by the translation code generated by rdc from the reductions (including the SEMANTIC_ANALYSIS subroutine); and concluding with a PL/I end statement for the translator. translator.pl1 is then compiled by the PL/I compiler to produce the translator object segment.

Note that, since PL/I code is inserted in translator.pl1 after the contents of translator.rd, care must be taken when coding translator.rd to insure that all of the semant subroutines and relative syntax functions are ended correctly, and that no end statement is included for the main procedure of the translator.

```
/* ********************                 -|
   * c Copyright ...    *               | copyright notice
   ******************** */              _|

/*++                                     -
      MAX_DEPTH 5 \                      |
      BEGIN                              | reduction statements and
        /    /    /           \          | attribute declarations
        /    /    / RETURN \             _|
                             ++*/

translator: procedure;

      dcl .... ,                         -|
          .... ,                         | translator's
          .... ;                         _| declarations

      dcl error_control_table...;

      call lex_string_$lex(..);          -| calls to parse translator
      Pthis_token = ...;                 | input into tokens,
      call SEMANTIC_ANALYSIS();          | translate these tokens,
      return;                            _| & return

      fcn: procedure returns             -|
        (bit(1) aligned);                | relative syntax
      end fcn;                           _| functions

      semant: proc(...);                 -| semantic
      ...                                | subroutines
      end semant;                        _|
```

Figure 1: Organization of a Translator

```
                                              translator.pl1
                                             | /* * * * * * * * * * */ |
                                             | /* heading          */  |
                                             | /* * * * * * * * * * */ |
   translator.rd                             | - - - - - - - - - - - - |
  | /* ***************          |            | /* ***************      |
  |     * c Copyright *         |            |     * c Copyright *     |
  |     *************** */      |            |     *************** */  |
  |                             |            |                         |
  | /*++  reductions ++*/       |            | /*++  reductions ++*/   |
  |                             |  1. rdc    |                         |
  | translator: proc(...);      |=========>  | translator: proc(...);  |
  |                             |            |                         |
  | dcl ...,                    |            | dcl ...,                |
  | error_control_table...;     |            | error_control_table...; |
  |                             |            |                         |
  | call lex_string_$lex..;     |            | call lex_string_$lex..; |
  | call SEMANTIC_ANALYSIS;     |            | call SEMANTIC_ANALYSIS; |
  | return;                     |            | return;                 |
  |                             |            |                         |
  | fcn: proc returns           |            | fcn: proc returns       |
  |        (bit(1) aligned);    |            |        (bit(1) aligned);|
  |    ...                      |            |    ...                  |
  | end fcn;                    |            | end fcn;                |
  |                             |            |                         |
  | semant: proc(...);          |            | semant: proc(...);      |
  |    ...                      |            |    ...                  |
  | end semant;                 |            | end semant;             |
  |_____________________________|            | - - - - - - - - - - - - |
                                             | SEMANTIC_ANALYSIS:      |
                                             |    procedure();         |
                                             |    ...                  |
   translator                                | end SEMANTIC_ANALYSIS;  |
  |                             |            | ERROR: procedure(n);    |
  |                             |  2. pl1    |    ...                  |
  |                             |<========== | end ERROR;              |
  |                             |            | NEXT_STMT: proc();      |
  |_____________________________|            |    ...                  |
                                             | end NEXT_STMT;          |
                                             | PUSH: proc(label);      |
                                             |    ...                  |
                                             | end PUSH;               |
                                             | - - - - - - - - - - - - |
                                             | end translator;         |
                                             |_________________________|
```

Figure 2: Two Steps of Compiling a Translator

## Translation - Part I: Parsing the Input Into Tokens

The translator receives a character string as its input. It must perform some transformation on this string, as defined by the syntax and semantics of the translation language. The translation begins by parsing the character string into a series of tokens (i.e., character strings separated by the translation language's delimiters). These tokens are the atoms of the translation language.

The lex_string_ external subroutine can be called, as shown in Figure 5, to parse the input characters into tokens. lex_string_ generates a chained list of token descriptors in an area provided by the translator. Each descriptor describes one of the tokens in the input. The token descriptors are chained together (forward and backward) in the order in which their respective tokens appear in the input string. The translator then has a chain of tokens which it can process, as shown in Figure 3.

```
     __      __      __      __      __      __      __      __      __
-->|   |-->|   |-->|   |-->|   |-->|   |-->|   |-->|   |-->|   |-->|   |
   |   |<--|   |<--|   |<--|   |<--|   |<--|   |<--|   |<--|   |<--|   |
   |___|   |___|   |___|   |___|   |___|   |___|   |___|   |___|   |___|
     |       |       |       |       |       |       |       |       |
     V       V       V       V       V       V       V       V       V
  Volume     :     70092     ;     Write     ;     File      4       ;
```

Figure 3: Input Tokens and their Descriptors

lex_string_ can optionally be invoked with a statement delimiter character string. lex_string_ uses this delimiter to group the tokens into statements. It can also create statement descriptors which point to the first and last token descriptors of each statement. Each token descriptor in turn points to its respective statement descriptor. The statement descriptors are chained together (forward and backward) in the order in which statements appear in the input string. Thus, with statement delimiters, the input to the translator is of the form shown in Figure 4.

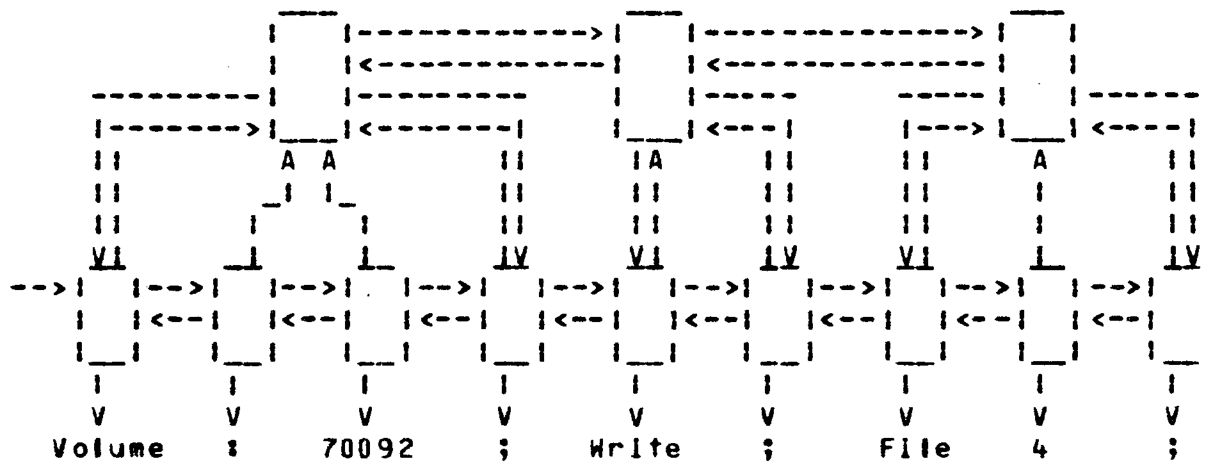


Figure 4: Tokens, Token Descriptors, and Statement Descriptors

Figure 5 shows lex_string_ being invoked first to initialize the lex_delims and lex_control_chars break definition strings, and then to parse the translator's input character string (described by Pinput and Linput) into tokens. In this example: a double quote (") character is used to open and close quoted strings; the characters /* open comments, which are closed by */; a semi-colon (;) is the statement delimiter; and the colon (:), comma (,), space ( ), and all of the ASCII control characters including the PAD character operate as delimiters, of which the space character and all control characters except backspace are ignored delimiters which are not returned as tokens themselves, even though they separate tokens. Both token descriptors and statement descriptors are generated by lex_string_ in this example. No descriptors are generated for the double quotes which enclose quoted strings, although descriptors are generated for the quoted strings themselves. Refer to the writeup on lex_string_ for more details on its calling sequence, as well as for a complete declaration of token and statement descriptors.

```
breaks = substr(collate,1,33) || ":," ||
     substr(collate,128,1);
ignored_breaks = substr(collate,1,8) ||
     substr(collate,10,24) || subst~(collate,128,1);
call lex_string_$init_lex_delims("""", """", "/*", "*/",
     ";", "10"b, breaks, ignored_breaks, lex_delims,
     lex_control_chars);
call lex_string_$lex(Pinput, Linput, Linput_ignore, Parea,
     "100"b, """", """", "/*", "*/", ";", breaks,
     ignored_breaks, lex_delims, lex_control_chars,
     Pfirst_stmt_descriptor, Pfirst_token_descriptor, code);
Pthis_token = Pfirst_token_descriptor;
call SEMANTIC_ANALYSIS();
return;
```

Figure 5: Parsing Translator Input Into Tokens, Semantically Analyzing Those Tokens, and Returning

## Translation - Part II: Analyzing the Tokens

The translation continues by analyzing the syntax of the input tokens to identify token phrases which are legal in the translation language. Legal token phrases must be assigned some semantic meaning, according to the specifications of the translation language, and illegal phrases must be diagnosed to the user. The syntax and semantics of the translation language are coded in a set of reduction statements. The reductions should specify the syntax of all possible sequences of input tokens, identifying legal sequences explicitly, and illegal sequences by default.

The translation of the input tokens is carried out by calling SEMANTIC_ANALYSIS, an internal procedure generated by rdc. SEMANTIC_ANALYSIS compares a sequence of input tokens (called a token phrase) with the syntax specifications defined in the reductions. If a token phrase matches the syntax requirements of a given reduction, the action routines associated with the reduction are invoked to assign some semantic meaning to the token phrase. The translation is complete when each of the input token phrases: either has been designated as a legal token phrase, and has been assigned a semantic meaning; or has been diagnosed as an illegal phrase.

The translator.pl1 segment generated by rdc contains declarations for many variables used by SEMANTIC_ANALYSIS. In particular, all of the variables defining the structure of tokens and their descriptors are declared in the main procedure. Several of these variables are declared in the main procedure of the translator so that they can be accessed by the translator's subroutines, as well as by SEMANTIC_ANALYSIS. Two such variables are the pointers, Ptoken and Pthis_token, which are used in processing the input tokens, as follows.

At any point in the translation process, some token phrase is being compared with the reductions. This phrase is called the "current" token phrase and its first token is called the "current" token. Pthis_token points to the descriptor of the "current" token, and hence identifies the beginning of the "current" token phrase. Ptoken points to the descriptor of the token within the "current" token phrase which is being compared with one of the syntax specifications of a reduction. Figure 6 illustrates the use of these two pointers.

```
       Pthis_token    Ptoken
            |            |
           _V_    __    _V_    __     __     __     __     __     __
      -->|   |-->|  |-->|   |-->|  |-->|  |-->|  |-->|  |-->|  |-->|  |
         |   |<--|  |<--|   |<--|  |<--|  |<--|  |<--|  |<--|  |<--|  |
         |__|    |__|   |__|    |__|   |__|   |__|   |__|   |__|   |__|
          |       |      |       |      |      |      |      |      |
          V       V      V       V      V      V      V      V      V
       Volume     :    70092     ;    Write    ;    File     4      ;

          A              A
          :              :
          :       TOKEN BEING EXAMINED
          :
      "CURRENT" TOKEN

     |_"CURRENT" TOKEN PHRASE__|
```

Figure 6: Pthis_token Identifies "Current" Token, Ptoken Identifies Token Being Examined

Note that the "current" token phrase does not contain a fixed number of tokens. Instead, the length of the "current" phrase varies to accommodate the number of language syntax specifications in each reduction. Of course, if fewer tokens remain to be translated than are required by the syntax specifications of a reduction, then the "current" token phrase cannot match that reduction.

## Reduction Language

The input language for rdc contains two kinds of statements: reduction statements (or simply reductions), and attribute declarations. Reduction statements specify the syntax of phrases in the translation language, and they assign semantic meaning to these phrases. Attribute declarations control the size of fixed-length tables which the translator will use. If any attribute declarations are given, they must precede the reduction statements.

The sections below describe the reduction language. Section headers have been numbered to provide easy cross-references between sections.

### 1. Reduction Statements

A reduction is a statement which contains four parts: a reduction label field; a syntax specification field; an action specification field; and a next-reduction field. It has the form:

```
optional labels / syntax / actions / next-reduction \
```

All of the fields must appear in each reduction, in the order given above. These fields are separated from one another by a right slant (/) character, and the final field is terminated by a left slant (\) statement delimiter character. The fields of the reduction statement may span any number of lines. A double quote (") character is used as the quoting delimiter, and left parenthesis ((), right parenthesis ()), less than (<), greater than (>), left bracket ([), right bracket (]), and backspace characters delimit tokens within reductions, and are tokens themselves. Spaces, tabs, new-line, new-page, and other ASCII control characters also delimit tokens, but are ignored by rdc, unless they are enclosed in quotes.

The left slant (\) character is used as a statement delimiter for reductions to facilitate writing a set of reductions for a language which uses the more common semi-colon statement delimiter.

## 2. Label Field

The label field of each reduction may contain zero, one, or more labels by which the reduction may be referenced. A label is a character string which begins with an alphabetic character, and contains 32 or fewer alphanumeric or underline (_) characters. Each of the labels defined in any set of reductions must be unique.

We will see (in Section 4) that labels can be used in the next-reduction field and in some action specifications to reference a particular reduction (in Section 11.1). In addition, the first reduction must have a label of BEGIN to distinguish it from any attribute declarations which may precede the reductions.

## 3. Syntax Field

The syntax specification field of each reduction identifies a token phrase by placing requirements on the tokens in the "current" token phrase. The tokens in the "current" phrase are compared consecutively with corresponding specifications in the syntax field. If each of the tokens matches its corresponding syntax specification, then the token phrase matches the requirements of the reduction. It is possible to classify all of the token phrases in the input by writing a set of reductions whose syntax fields identify all of the legal phrases in the language to be translated, and by including one or more reductions which match all other (illegal) token phrases.

There are three types of syntax specifications: absolute syntax specifications; relative syntax functions; and built-in syntax functions. They are discussed in the sections below.

### 3.1 Absolute Syntax Specifications

Absolute syntax specifications require that their corresponding input token equal a particular character string. Absolute specifications are represented by their character string value in the syntax specification field. If a set of reductions were written to translate the tokens in Figure 6, "Volume", ":", ";", "Write", and "File" would probably be identified by absolute syntax specifications.

rdc's delimiter characters may be used in absolute specifications by enclosing the entire specification in quotes (e.g., "and/or", ">udd>proj>prog", """""", "(", ")", "<", ">", "/",

"\", "[", "]", or "  (i.e., "<backspace>")). In addition, the delimiters which have special meaning within the syntax specification field of a reduction (/, <, and >) may be used as one-character absolute specifications by underlining the characters. That is, $\underline{/}$, $\underline{<}$, and $\underline{>}$ are interpreted by rdc as the single-character absolute syntax specifications, /, <, and >.

## 3.2 Relative Syntax Functions

Relative syntax functions are a second type of syntax specification. A relative syntax function requires that its corresponding input token meet some special requirements that are defined by a PL/I function. The requirements defined by such functions may be quite specific or very general in nature, according to the needs of the translation language. The translator must supply the relative syntax functions which it needs to identify phrases in the translation language. Zero, one, or more PL/I functions may be created and referenced as relative syntax functions. Relative syntax functions are represented in the syntax specification field by enclosing the name of the function in angle brackets (e.g., <fcn_name>).

Typical relative syntax functions might be described as follows: <relative_pathname> requires that the token value be a relative path name, and calls expand_path_ to associate an absolute path name as the semantic value of this relative path name; <positive_integer> requires that the token value be a character string representation of a positive integer, and stores the numeric value of the integer in the token.Nvalue element of the token's descriptor; <volume_id> requires that the token value be a 6-character tape volume identifier; and <time_of_day> requires that the token value be convertible to a time of the day.

### 3.2.1 Relative Syntax Functions - Calling Sequence

The calling sequence of a relative syntax function is shown below:

```
dcl  fcn_name: entry returns (bit(1) aligned);

token_matches = fcn_name();
```

The function should return a value of "1"b if the input token matches the requirements of the function, and "0"b otherwise.

The function must be an internal procedure of the translator. It can have any valid PL/I function name which is 32 or fewer alphanumeric or underline characters in length, which contains at least one lower-case alphabetic letter.

### 3.2.2 Relative Syntax Functions - Referencing the Token

By being an internal function of the translator, the relative syntax function can reference its corresponding token in the "current" token phrase to see if that token meets the requirements of the function. To do this, the function references token_value, a variable declared by rdc in the main procedure of the translator. token_value is based on the information in the token's descriptor. This descriptor is pointed to by Ptoken, another variable declared by rdc which is set before the relative syntax function is invoked. (See Figure 6.)

### 3.2.3 Relative Syntax Functions - Assigning Semantic Values

The relative syntax function may associate a semantic value with the token being examined in one of two ways. It can set a variable which has been declared in the main procedure of the translator. Or it can allocate a semantic value structure in the area used for token descriptors, and can then chain this structure onto the token descriptor using the token.Psemant pointer. Refer to the lex_string_ writeup for a complete declaration of the token descriptor's structure.

### 3.3 Built-in Syntax Functions

The third type of syntax specification is the built-in syntax function. These are relative syntax functions which have been pre-defined by rdc. Although several of the built-in syntax functions make requirements on the input token string that would be difficult to implement as relative syntax functions, most built-in syntax functions were defined merely to facilitate the implementation of rdc, itself. Below is a list of the built-in syntax functions which have been defined.

<no-token> requires that there are no more tokens remaining in the input token string. All of the tokens have been translated, and the chain of token descriptors is exhausted.

<any-token> requires that its corresponding token exist in the input token string. Any token value is accepted as part of the legal syntax of the language being translated.

<name> requires that the input token be a character string which begins with an alphabetic character and contains 32 or fewer alphanumeric, underline (_), or dollar sign ($) characters.

<decimal-integer> requires that the input token be a valid, optionally-signed decimal integer (as defined by the cv_dec_check_ subroutine). The numeric value of the token is stored as its semantic value in the token.Nvalue element of the token descriptor structure.

<quoted-string> requires that the token.S.quoted_string bit be turned on in the input token's descriptor. This bit is turned on by lex_string_ if the token was enclosed within quoting delimiters when lex_string_ parsed the translator input.

<BS> requires that the input token be a single backspace character.

## 4. Next-Reduction Field

Before discussing the assignment of semantic meaning to the token phrase which matches a reduction, the flow of control between reductions will be described.

When the translator calls the SEMANTIC_ANALYSIS procedure, control passes to the reduction whose label is BEGIN. The first of the "current" token phrases is compared with this beginning reduction and those which follow until it matches the syntax requirements of one of the reductions. The action specifications of that reduction are then performed to assign semantic meaning to the "current" token phrase, and to make the next token phrase "current".

After performing the action specifications, the next-reduction field of the matched reduction controls which reduction the new "current" token phrase is compared with. The next-reduction field may be blank, or it may contain a reduction label. If it is blank, then the reduction immediately following the matched reduction is used in the next comparison. If a reduction label is specified, then the reduction identified by that label is used in the next comparison. In either case, comparison of the new "current" token phrase with reductions continues until a matching reduction is found. This process is repeated until all of the input tokens have been translated.

Each set of reductions must contain one or more reductions which use the <no-token> built-in syntax function to detect when all the input tokens have been translated. When such a <no-token> reduction is invoked, its next-reduction field usually contains the RETURN keyword, instead of a reduction label, to specify that the flow of control should return to the caller of the SEMANTIC_ANALYSIS procedure. On return from SEMANTIC_ANALYSIS, the translation is complete.

Often if several <no-token> reductions appear in a set of reductions, a reduction label is used in their next-reduction field (rather than a RETURN keyword) to branch to a final <no-token> reduction which performs epilogue actions and then returns via a RETURN keyword. Having only one of the <no-token> reductions perform the epilogue actions reduces the amount of translation code generated by rdc.

```
<spec>  ::=   Volume : <volume-id>[,{9track|7track}] ;
              {Read|Write} ;
              File <number> ;
              Records : <number>[, <number>]... ;
              Format : {F|FB|FBS|V|VB|VBS|U} ;
```

Figure 7: BNF Syntax for a Tape Language

```
BEGIN
stmt
   / Volume : <volume_id>        /                    / vol     \
   / Read ;                      /                    / stmt    \
   / Write ;                     /                    / stmt    \
   / File <positive_integer> ;/                       / stmt    \
   / Records :                   /                    / numbers\
   / Format :                    /                    / format  \
   / <any-token>                 /                    / stmt    \
   / <no-token>                  /                    / RETURN  \

vol
   / ;                           /                    / stmt    \
   / , 9track ;                  /                    / stmt    \
   / , 7track ;                  /                    / stmt    \
   / <any-token>                 /                    / stmt    \
   / <no-token>                  /                    / RETURN  \

numbers
   / <positive_integer>          /                    / punct   \
   / <any-token>                 /                    / punct   \
   / <no-token>                  /                    / RETURN  \

punct
   / ,                           /                    / numbers\
   / ;                           /                    / stmt    \
   / <any-token>                 /                    / numbers\
   / <no-token>                  /                    / RETURN  \

format
   / F ;                         /                    / stmt    \
   / FB ;                        /                    / stmt    \
   / FBS ;                       /                    / stmt    \
   / V ;                         /                    / stmt    \
   / VB ;                        /                    / stmt    \
   / VBS ;                       /                    / stmt    \
   / U ;                         /                    / stmt    \
   / <any-token>                 /                    / stmt    \
   / <no-token>                  /                    / RETURN  \
```

Figure 8: Reductions for the Tape Language
(Action Specifications Omitted)

## 5. Sample Reductions #1

Figure 7 shows the Backus-Naur Form (BNF) for the syntax of a language which identifies records to be read or written from a tape file on a particular volume, using a given record format. Several examples below will employ this language to illustrate the use of reductions.

Figure 8 shows how the reduction fields described so far can be used to define the syntax of the tape language shown in Figure 7. <positive_integer> and <volume_id> are the relative syntax functions described in Section 3.2. Note that an <any-token> reduction is included in each group of reductions, in addition to a <no-token> reduction, in order to detect errors in the use of the tape language. An <any-token> reduction (one containing only the <any-token> built-in syntax specification) matches any token phrase except null token phrases (those which match a <no-token> reduction).

## 6. Action Field - Semantic Subroutines

When a legal token phrase is identified by the syntax field of a reduction, the translator must assign some semantic meaning to that phrase, according to the specifications of the translation language. It does this by invoking the semantic subroutines and other action routines which are specified in the action field of the matching reduction. These subroutines are invoked in the order of their appearance in the action field.

The translator must supply semantic subroutines which assign some semantic meaning to the matched token phrase. Semantic subroutines can construct and fill in tables, build compiler trees, generate object code, or do any other functions which are required to perform the translation. rdc supplies other action routines which can make another token phrase the "current" token phrase and perform other functions. These are described in Sections 7, 9, and 11 below.

Often the semantic subroutines must reference one of the tokens in the matching token phrase, or it must reference the semantic value structure attached to the descriptor of one of these tokens. Because it is easiest for a semantic subroutine to reference the "current" token, a semantic subroutine is often preceded in the action field by a lexing routine, an action routine supplied by rdc which makes the token of interest to the semantic subroutine be the "current" token. Lexing routines are described in Section 7.

### 6.1 Semantic Subroutines - Calling Sequence

A semantic subroutine may have any calling sequence accepted by PL/I. If the subroutine would normally be invoked by a PL/I call statement of the form:

```
call semantic_sub (1, "1"b, "able", token_value, (c1+c2));
```

then the semantic subroutine appears in the action specification field as:

```
semantic_sub (1, "1"b, "able", token_value, (c1+c2))
```

A semantic subroutine which requires no input arguments would be invoked by a PL/I call statement of the form:

```
call semant();
```

It can appear in the action specification as:

```
semant
```

An example of a reduction containing semantic subroutines is:

```
/ File <positive_integer> ; / LEX set_file
                              open_file(token.Nvalue,"r")
                              LEX(2)          / stmt \
```

It is often useful to define a single semantic subroutine which performs a group of related functions. This semantic subroutine can then be invoked from many different reductions with a constant argument specifying which of the functions should be performed. Since semantic subroutines may have a different argument list each time they appear in a reduction action field, it is easy to create and use such a multi-function semantic subroutine in a translator.

## 6.2 Semantic Subroutines - Passing Variables As Arguments

Several facts must be considered when passing variables as the arguments to a semantic subroutine. First, the semantic subroutine is actually called from within the SEMANTIC_ANALYSIS procedure. Therefore, the subroutine itself and any variables passed to the subroutine must be known within the scope of SEMANTIC_ANALYSIS. This can be accomplished by defining internal semantic subroutines, and by declaring external subroutines and their variable arguments, within the main procedure of the translator. (See Figure 2.)

Second, care must be taken to avoid name conflicts between the variables declared within SEMANTIC_ANALYSIS, and the semantic subroutines and their arguments. The variables declared by SEMANTIC_ANALYSIS have all been declared with names formed of upper-case letters, with a few exceptions described below. Therefore, name conflicts can generally be avoided by declaring names of translator variables and semantic subroutines which have one or more lower-case letters or digits.

There are three types of exceptions to the upper-case naming convention used within SEMANTIC_ANALYSIS. These exceptions must be considered when naming the translator's semantic subroutines and variables. First, SEMANTIC_ANALYSIS uses and has declared the following PL/I built-in functions: addr, max, null, search, substr, and verify. Second, SEMANTIC_ANALYSIS uses and has declared cv_dec_check_ to be the Multics number conversion function documented in the MPM. Third, the variables and structures required to reference tokens and their descriptors have been declared by rdc in the main procedure of the translator. These variables and structures are referenced by SEMANTIC_ANALYSIS. They are described in the writeup on lex_string_.

## 6.3 Semantic Subroutines - Referencing the "Current" Token

If the semantic subroutine is an internal procedure, it can access the character string value of the "current" token by referencing the token_value variable, just as a relative syntax function does. It can also reference the token descriptor for the "current" token (via Ptoken), and any semantic value structure attached to that descriptor.

If the semantic subroutine is an external procedure, then token_value, Ptoken, or the semantic value of the "current" token can be passed to the subroutine as an argument.

### 6.4 Semantic Subroutines - Examining Other Tokens

Tokens other than the "current" token may be examined from a semantic subroutine by obtaining a pointer to the descriptor for the desired token, assigning this pointer to Ptoken, and referencing the token_value variable. Pointers to the desired token descriptor structures may be stored by other semantic subroutines (for example, in a token push down stack used to process polish strings). Alternatively, by using the forward and backward pointers in the token descriptors, the semantic subroutine can obtain a pointer to the descriptor of a token which precedes or follows the "current" token by some known number of tokens. For example,

```
Ptoken = Pthis_token -> token.Pnext -> token.Pnext;
```

causes the token_value variable to reference the 2nd token following the "current" token. Remember that Pthis_token points to the descriptor of the "current" token.

Before invoking the subroutines in the action field of a reduction, SEMANTIC_ANALYSIS sets Ptoken equal to Pthis_token. SEMANTIC_ANALYSIS does not use or depend upon the value of Ptoken until the action field has been completely executed. It then resets Ptoken to equal Pthis_token. Therefore, Ptoken can be changed by one or more of the subroutines in the action field, as long as the change has no ill effects on the subroutines which follow.

The best coding practice is for a semantic subroutine to assume that Ptoken equals Pthis_token. If the subroutine changes Ptoken, it should reset Ptoken to equal Pthis_token before returning to its caller. (Note that the lexing routines described in Section 7 below change the value of Pthis_token, and then set Ptoken equal to the new value of Pthis_token.)

### 6.5 Semantic Statements Avoid One-Statement Semantic Subroutines

In many translators, the majority of the semantic subroutines perform very simple operations like turning on a bit or assigning a particular value to a variable. To avoid having to create one-statement semantic subroutines to perform these operations, the reduction language provides a semantic statement facility.

A semantic statement is a PL/I statement (excluding the final semi-colon) which appears, enclosed in square brackets, in the action field of a reduction. For example

```
[file_number = token.Nvalue + 2]
```

is a semantic statement which assigns the numeric value of the "current" token plus 2 to the variable called file_number. token.Nvalue could have been set by the <positive_integer> relative syntax function described in Section 3.2. Care must be taken, as described in Section 6.2, to avoid naming conflicts between the variables used in semantic statements and the variables declared by the SEMANTIC_ANALYSIS procedure. More than one semantic statement may appear within the same pair of square brackets by placing a semi-colon between each pair of statements. For example

```
[if a > 1 then call ERROR(20); else call ERROR(21)]
```

## 7. Action Field - Lexing Routines

Besides invoking semantic subroutines to attach meaning to the "current" token phrase, the action field of a reduction must skip over that phrase so that the next token phrase can be processed by the translator. It does this by making Pthis_token (the pointer to the descriptor of the first token in the "current" token phrase) point to the descriptor of the first token of the next token phrase. This process of moving the pointer to the "current" token is called lexing. Three lexing action routines are provided to perform this function: LEX; LEX(n); and NEXT_STMT.

7.1 Lexing Routines - LEX

The LEX action routine makes Pthis_token point to the descriptor of the token which immediately follows the "current" token. This effectively makes the next token the new "current" token. rdc compiles a LEX action routine into a PL/I statement of the form:

```
Ptoken, Pthis_token = Pthis_token -> token.Pnext;
```

7.2 Lexing Routines - LEX(*n*)

For positive *n*, the LEX(*n*) action routine makes Pthis_token point to the descriptor of the *n*th token which immediately follows the "current" token. This effectively makes the next *n*th token the new "current" token. rdc compiles a LEX(2) action routine into a PL/I statement of the form:

```
Ptoken, Pthis_token = Pthis_token->token.Pnext->token.Pnext;
```

LEX(n) also accepts negative values of *n*. If *n* is negative, LEX(*n*) makes Pthis_token point to the |*n*|th token which *precedes* the "current" token. LEX(-1) action is compiled into:

```
Ptoken, Pthis_token = Pthis_token->token.Plast;
```

Note that care must be taken when writing the reductions to insure that all tokens *skipped over* to reach the new "current" token actually exist. If they do not exist, the code shown above will attempt to reference through a null pointer. The token which will become the new "current" token as the result of a LEX(*n*) operation need not exist, however. If the nth following (or |n|th preceding) token does not exist, Pthis_token and Ptoken are set to null pointers by the code shown above, indicating that the "current" token phrase is a null token phrase (i.e., one containing no tokens and matching a <no-token> reduction) and that all of the input tokens have been translated. In every translation, the last phrase to be translated is such a null phrase.

## 7.3 Lexing Routines - NEXT_STMT

The NEXT_STMT action routine makes the first token of the next statement (after the statement containing the "current" token) the new "current" token. This action routine can only be used when the translator requires lex_string_ to create statement descripters. It can be used to skip over the remainder of a statement when an unrecoverable error has been detected in that statement.

## 7.4 Lexing Routines - Invoked from a Semantic Subroutine

It is sometimes necessary for a semantic subroutine to perform lexing operations, especially to correct an error. It can perform a LEX or LEX(n) operation by executing a PL/I statement like the ones shown in Sections 7.1 and 7.2. It can perform a NEXT_STMT operation by calling the NEXT_STMT internal procedure which is supplied by rdc:

```
call NEXT_STMT();
```

These operations may only be performed by semantic subroutines which are internal procedures, thereby having access to the Ptoken and Pthis_token variables and to the NEXT_STMT procedure, or by external procedures to which these variables or the NEXT_STMT procedure have been passed as arguments.

## 8. Sample Reductions #2

Figure 9 shows the reductions for our tape language, with the action fields filled in. Note that only one of the <no-token> reductions performs epilogue functions, and that this reduction receives control from all other <no-token> reductions. Note too that no semantic subroutines have been specified in the action field of reductions which identify illegal phrases in the input. Section 9 describes a general-purpose error diagnosis semantic subroutine which can be used by any translator to inform the user of errors in the input to the translator.

```
BEGIN
stmt
   / Volume : <volume_id> ;    / LEX(2) [volume=token_value]
                                 [track = 9] LEX    / vol      \
   / Read ;                    / LEX(2) [mode="r"] / stmt     \
   / Write ;                   / LEX(2) [mode="w"] / stmt     \
   / File <positive_integer> ;/ LEX [file_no=token.Nvalue]
                                 LEX(2)             / stmt     \
   / Records :                 / LEX(2)             / numbers\
   / Format :                  / LEX(2)             / format   \
   / <any-token>               / NEXT_STMT          / stmt     \
   / <no-token>                / perform_io         / end      \
vol
   / ;                         / LEX                / stmt     \
   / , 9track ;                / LEX(3)             / stmt     \
   / , 7track ;                / [track = 7] LEX(3)/ stmt     \
   / <any-token>               / NEXT_STMT          / stmt     \
   / <no-token>                /                    / end      \
numbers
   / <positive_integer>        / set_record_no LEX / punct    \
   / <any-token>               / LEX                / punct    \
   / <no-token>                /                    / end      \
punct
   / ,                         / LEX                / numbers\
   / ;                         / LEX                / stmt     \
   / <any-token>               / LEX                / numbers\
   / <no-token>                /                    / end      \
format
   / F ;                       / LEX(2) format(1)   / stmt     \
   / FB ;                      / LEX(2) format(2)   / stmt     \
   / FBS ;                     / LEX(2) format(3)   / stmt     \
   / V ;                       / LEX(2) format(4)   / stmt     \
   / VB ;                      / LEX(2) format(5)   / stmt     \
   / VBS ;                     / LEX(2) format(6)   / stmt     \
   / U ;                       / LEX(2) format(7)   / stmt     \
   / <any-token>               / NEXT_STMT          / stmt     \
   / <no-token>                /                    /          \
end
   / <any-token>               / epilogue           / RETURN \
   / <no-token>                / epilogue           / RETURN \
```

Figure 9: Reductions for the Tape Language
(Error Diagnostic Actions Omitted)

## 9. Action Field - Diagnosing Errors

Besides translating all legal token phrases in the input, most translators identify and report any illegal phrases which may be present. An <any-token> reduction can be used at the end of a group of reductions to identify any non-null token phrase which does not match one of the preceding reductions in the group. Also, specific reductions can be provided to identify predictable errors, such as missing or illegal punctuation, invalid keywords or names in otherwise legal statements, etc. In addition, semantic subroutines may identify inconsistent or invalid input as the translation progresses.

When errors are identified, the user must be notified of the type of error which has occurred, and the location of the error in the input (if known). rdc provides two facilities for printing error messages: the ERROR internal subroutine; and the lex_error_ external procedure.

### 9.1 Error Routines - ERROR(error_number)

The ERROR semantic subroutine can be used by a translator to print error messages. The procedure is invoked from the action field of a reduction by:

```
ERROR(error_number)
```

or from one of the translator's semantic subroutines by:

```
call ERROR(error_number);
```

In order to use the ERROR subroutine, the translator must supply an error_control_table. error_number is an integer index into error_control_table, which is an internal static array of structures declared by the translator in the main procedure of the translator. A declaration for a typical error_control_table is shown below.

```
     dcl  1 error_control_table (2) aligned internal static,
            2 severity          fixed bin(17) unaligned init (2,3),
            2 Soutput_stmt      bit(1) unaligned init ("0"b,"1"b),
            2 message           char(100) varying init (
              "The  reduction source segment does not contain any
valid reductions.",
              "The reduction label  '^a'  is invalid.  This label
has been ignored."),
            2 brief_message    char(24) varying init (
              "No valid reductions.",
              "Label  '^a'  invalid.");
```

Each element of the error_control_table array is a structure which describes one error message. The structure contains: a severity level for the error; a switch which specifies whether the statement containing the "current" token phrase should be output after the error message; a long form of the error message text; and a brief form of the error message text. The error_control_table must be a one-dimensional array, but its upper bound may be declared to suit the needs of the translator.

Note that statement descriptors must be present in order to put the statement containing the "current" token phrase into the error message. Therefore, the Soutput_stmt switch has no effect unless the translator has requested that lex_string_ generate statement descriptors. (See the writeup on lex_string_ to learn how to request statement descriptors.)

The text of the error message is an ioa_ control string. Therefore, although the lengths of the message and brief_message error message texts may be declared according to the needs of the translator, these lengths must not be longer than 256 characters. Up to three occurrences of the ioa_ control characters, ^a, may appear in the message or brief_message character string. The value of the "current" token will replace these control characters in the printed error message. Any number of the ioa_ control characters, ^-, ^/, ^|, ^x, ^R, ^B, and ^^, may appear in the error message text.

The choice of the long message text or the brief text for use in the error message is controlled by the value of a variable, SERROR_CONTROL, which is declared by rdc in the main procedure of the translator. SERROR_CONTROL is a bit string of length 2, which is initialized with a value of "00"b. Table 1 shows how these two bits are interpreted.

Table 1: Interpretation of SERROR_CONTROL Bits

| SERROR_CONTROL | meaning |
|---|---|
| "00"b | the first time a particular error occurs, the long message text is used in the error message; the short message text is used in any subsequent occurrences of that error. |
| "10"b | the long message text is always used in the error message. |
| "11"b | the long message text is always used in the error message. (equivalent to "10"b) |
| "01"b | the brief message text is always used in the error message. |

The error messages which are printed have the form shown below:

<u>prefix</u> <u>error number</u>, SEVERITY <u>severity</u> IN STATEMENT <u>m</u> OF LINE <u>n</u>
<u>text of error message</u>
SOURCE:
<u>statement containing "current" token phrase</u>

The value of <u>prefix</u> is controlled by the severity level associated with the error message, as shown in Table 2.

The statement and line numbers in the message are obtained from the token descriptor of the "current" token or from the statement descriptor of the statement containing the "current" token.

"IN STATEMENT <u>m</u> OF LINE <u>n</u>" only appears in the error message if statement descriptors have been provided by lex_string_. <u>n</u> is the line number on which the statement containing the "current" token begins, and <u>m</u> is a number which identifies which statement in line <u>n</u> was in error, if more that one statement appears in line <u>n</u>. If line <u>n</u> contains only one statement, then "STATEMENT <u>m</u> OF" is omitted from the error message.

If no statement descriptors are available, then "STATEMENT <u>m</u> OF" is omitted from the message, and <u>n</u> is the line number on which the "current" token appears. If Pthis_token is null to indicate that the "current" token phrase is null, then "IN STATEMENT <u>m</u> OF LINE <u>n</u>" is omitted altogether.

Table 2: Relationship of Prefix to Error Severity

| sev | prefix | explanation |
|---|---|---|
| 0 | COMMENT | The error message is a comment. |
| 1 | WARNING | The error message warns that a possible error has been detected. The translation will still proceed, however. |
| 2 | ERROR | The error message warns that a probable error has been detected. However, the error is non-fatal and the translation will proceed. |
| 3 | FATAL ERROR | The error message warns that a fatal error has been detected. Processing of the input will continue to diagnose further errors, but no translation will be performed. |
| 4 | TRANSLATOR ERROR | The error message warns that an error has been detected in the operation of the translator. No translation will be performed. |

If Soutput_stmt is off, then "SOURCE" and the statement containing the "current" token phrase are omitted from the error message. If this statement has been printed in a previous error message, then "SOURCE" and the statement are omitted from this error message.

rdc declares two other variables in the main procedure of the translator which are used by the ERROR subroutine. SERROR_PRINTED is an array of bits, with one bit per message in the error_control_table. All bits in the array are initially turned off when the translation begins. Whenever an error message is printed, SERROR_PRINTED(error_number) is turned on. This procedure allows ERROR to detect when subsequent occurrences of the error occur, so that SERROR_CONTROL = "00"b can be implemented.

The second variable declared by rdc in the main procedure of the translator is MERROR_SEVERITY. This is a fixed bin(17) integer which is initialized to zero, and which is used to maintain the severity of the highest-severity error printed during the translation. The translator may reference the value of this variable at the end of the translation to return this highest-severity to its caller, or to determine when to abort the translation due to a fatal error.

The ERROR semantic subroutine and declarations for SERROR_CONTROL, SERROR_PRINTED, and MERROR_SEVERITY are automatically included in every translator which specifies this subroutine in one or more of its reduction action fields. ERROR accesses the appropriate values in the error_control_table, and passes these values and the pointers to the "current" token and its statement descriptor to the lex_error_ external procedure. lex_error_ invokes ioa_ to format the error message, and outputs the message on the error_output I/O stream.

9.2 Error Routines - lex_error_(...)

Although the ERROR procedure described above is very easy to use, the cost of its simplicity comes in its inability to generate highly-specific error messages containing several different variable information fields. ERROR only allows the character string value of the "current" token to be included in the error message.

When more flexible error messages are required, the translator can call the lex_error_ procedure, itself, passing lex_error_ information from the error_control_table (or writing messages not included in the error_control_table), pointers to the statement descriptor for the statement containing the "current" token phrase, and arguments to be substituted into the error message text, according to ioa_ control characters. Refer to the writeup on the lex_error_ external procedure for more information.

Care should be taken to pass lex_error_ elements of the error_control_table by value, rather than by reference. This will enable the PL/I compiler to treat the error_control_table as a constant structure which can be stored in the text of the translator, rather than in its linkage section. error_control_table elements can be passed to lex_error_ by value by surrounding the references to these elements by parentheses in the call to lex_error_.

```
BEGIN
stmt
   / Volume : <volume_id>      / LEX(2) [volume=token_value]
                                 [track = 9] LEX   / vol     \
   / Read ;                    / LEX(2) [mode="r"] / stmt    \
   / Write ;                   / LEX(2) [mode="w"] / stmt    \
   / File <positive_integer> ;/ LEX [file_no=token.Nvalue]
                                 LEX(2)            / stmt    \
   / Records :                 / LEX(2)            / numbers\
   / Format :                  / LEX(2)            / format  \
   / <any-token>               / ERROR(1) NEXT_STMT/ stmt    \
   / <no-token>                / perform_io        / end     \
vol
   / ;                         / LEX               / stmt    \
   / , 9track ;                / LEX(3)            / stmt    \
   / , 7track ;                / [track = 7] LEX(3)/ stmt    \
   / <any-token>               / ERROR(7) NEXT_STMT/ stmt    \
   / <no-token>                / ERROR(3)          / end     \
numbers
   / <positive_integer>        / set_record_no LEX / punct   \
   / <any-token>               / ERROR(2) LEX      / punct   \
   / <no-token>                / ERROR(3)          / end     \
punct
   / ,                         / LEX               / numbers\
   / ;                         / LEX               / stmt    \
   / <any-token>               / ERROR(4) LEX      / numbers\
   / <no-token>                / ERROR(3)          / end     \
format
   / F ;                       / LEX(2) format(1)  / stmt    \
   / FB ;                      / LEX(2) format(2)  / stmt    \
   / FBS ;                     / LEX(2) format(3)  / stmt    \
   / V ;                       / LEX(2) format(4)  / stmt    \
   / VB ;                      / LEX(2) format(5)  / stmt    \
   / VBS ;                     / LEX(2) format(6)  / stmt    \
   / U ;                       / LEX(2) format(7)  / stmt    \
   / <any-token>               / ERROR(5) NEXT_STMT/ stmt    \
   / <no-token>                / ERROR(3)          / end     \
end
   / <any-token>               / ERROR(6) epilogue / RETURN \
   / <no-token>                / epilogue          / RETURN \
```

Figure 10: Reductions for the Tape Language

## 10. Sample Reductions #3

Figure 10 shows the reductions for our tape language, including error diagnostic calls to the ERROR subroutine. The declaration of the error_control_table to be used with these reductions is shown in Figure 11.

```
     dcl  1 error_control_table (7) internal static,
            2 severity fixed bin(17) unaligned
              init (3, 2, 3, 2, 3, 2, 2),
            2 Soutput_stmt bit(1) unaligned
              init ("1"b, "1"b, "0"b, "1"b, "1"b, "1"b, "1"b),
            2 message char(70) varying init(
              "An unknown statement has been encountered.",
              "'^a' is an invalid record number.",
              "Translator   input   ends   with   an   incomplete
statement.",
              "'^a'  is  invalid  punctuation in a list of record
numbers.",
              "'^a' is an invalid record format.",
              "More  input  was  encountered when the end  of
translator input was expected.",
              "A  bad  track  specification was given in a Volume
statement.  9track has been assumed."),
            2 brief_message char(28) varying init(
              "Unknown statement.",
              "Bad record number '^a'.",
              "Incomplete statement.",
              "Invalid punctuation '^a'.",
              "Invalid record format '^a'.",
              "Too much input.",
              "Bad track in Volume.");
```

Figure 11: error_control_table For Reductions in Figure 10

## 11. The Reduction Stack

Often a language to be translated contains syntactic constructs which are similar in form, but which differ in their use of keywords, types of values, etc. The BNF for one such language is shown in Figure 12. The language accepts three different types of statements, each of which includes a list of values.

```
<stmt>  ::=    Name : <name>[, <name>]... ;
             | Attribute : <attr>[, <attr>]... ;
             | Value : <number>[, <number>]... ;

<attr>  ::=    fixed | float | decimal | binary
```

Figure 12: BNF for a Value Space Language

The lists in this language all have the same syntax, and differ only in the keyword at the beginning of the statement and in the type of values included in the list. This suggests that the list punctuation for all three types of lists might be handled by a common group of reductions if there were some way to invoke the group of reductions as a subroutine which would return to some pre-defined reduction after processing the punctuation marks in the list.

rdc implements such a reduction subroutine facility by providing a reduction stack. This stack contains the labels of the reductions which are to be returned to when a reduction subroutine has completed its processing. Appropriate next-reduction keywords are provided to indicate that the reduction identified by the label at the top of the reduction stack should be the next reduction. Action subroutines are provided by rdc for pushing a reduction onto the stack, and later popping it off the stack. These facilities are all described in the next few sections.

### 11.1 Action Field - PUSH(label)

The PUSH action subroutine can be used in the action field of a reduction to push the reduction identified by label onto the reduction stack. PUSH is an internal procedure included automatically by rdc in any translator which uses the PUSH action routine.

If pushing the reduction onto the stack would cause a stack overflow, then the PUSH subroutine writes a special severity 4 error (error number 0) through lex_error_, and calls cu_$cl to invoke a new level of the command processor. The start command cannot be issued after such a stack overflow has occurred, but the translator maintenance personnel can perform debugging operations from the new level of the command processor.

11.2 <u>Attribute Declaration</u> - MAX_DEPTH *n* \

Normally, enough storage is declared (at 1 word per reduction) for 10 reductions to be pushed onto the reduction stack. The translator may increase or decrease the amount of storage which is reserved to meet the needs of its reduction subroutine strategy. The size of the reduction stack can be set by using the MAX_DEPTH attribute declaration, which has the form:

MAX_DEPTH *n* \

where *n* is an integer, such that 0 < *n* < 10000, specifying the maximum number of reductions which can be pushed onto the reduction stack at any given time. If a MAX_DEPTH attribute declaration is given, it must appear before any of the reductions in the input to rdc.

11.3 <u>Action Field</u> - POP

The POP action routine can be used in the action field to pop the top reduction off of the reduction stack. POP is a built-in action routine supplied by rdc. If POP is invoked when there are no reductions on the stack, then no popping operation is performed, and no error is reported either.

11.4 <u>Next-Reduction Field</u> - STACK

The STACK keyword may be used in the next-reduction field of a reduction to transfer to the reduction on top of the reduction stack. If the reduction stack is empty when the STACK keyword is specified, then a blank next-reduction field is assumed and the reduction following the one containing the STACK keyword is used in the next comparison.

11.5 <u>Next-Reduction Field</u> - STACK_POP

Probably the most useful method of returning from a reduction subroutine is to transfer to the reduction on top of the reduction stack, while at the same time popping that reduction from the stack. This combination of the STACK and POP operations can be performed by specifying the STACK_POP keyword in the next-reduction field of a reduction. As with STACK, if the reduction stack is empty, then a blank next reduction field is assumed and the reduction following the one containing STACK_POP is used in the next comparison.

```
MAX_DEPTH 2 \
BEGIN
stmt
   / Name :          / LEX(2) PUSH(stmt)          / names     \
   / Attribute :     / LEX(2) PUSH(stmt)          / attr      \
   / Value :         / LEX(2) PUSH(stmt)          / values    \
   / <any-token>     / ERROR(1) NEXT_STMT         / stmt      \
   / <no-token>      /                            / RETURN    \

names
   / <name>          / set_name LEX PUSH(names)   / punct     \
   / ;               / ERROR(2) LEX               / STACK_POP \
   / ,               / ERROR(2) LEX               / names     \
   / <any-token>     / ERROR(3) LEX PUSH(names)   / punct     \
   / <no-token>      / ERROR(4)                   / RETURN    \

attr
   / fixed           / attr(1) LEX PUSH(attr)     / punct     \
   / float           / attr(2) LEX PUSH(attr)     / punct     \
   / decimal         / attr(3) LEX PUSH(attr)     / punct     \
   / binary          / attr(4) LEX PUSH(attr)     / punct     \
   / ;               / ERROR(2) LEX               / STACK_POP \
   / ,               / ERROR(2) LEX               / attr      \
   / <any-token>     / ERROR(5) LEX PUSH(attr)    / punct     \
   / <no-token>      / ERROR(4)                   / RETURN    \

values
   / <decimal_number>
                     / set_num  LEX PUSH(values)/ punct     \
   / ;               / ERROR(2) LEX               / STACK_POP \
   / ,               / ERROR(2) LEX               / values    \
   / <any-token>     / ERROR(6) LEX PUSH(values)/ punct     \
   / <no-token>      / ERROR(4)                   / RETURN    \

punct
   / ;               / LEX POP                    / STACK_POP \
   / ,               / LEX                        / STACK_POP \
   / <any-token>     / ERROR(7) NEXT_STMT POP     / STACK_POP \
   / <no-token>      / ERROR(4)                   / RETURN    \
```

Figure 13: Reductions for Value Space Language

11.6 Sample Reductions #4

The reductions for the value space language of Figure 12 are shown in Figure 13. In these reductions, <number> is a relative syntax function which converts a character-format number to floating decimal, and stores the result in a semantic value structure attached to the number's token descriptor.

The error messages generated by the reductions in Figure 13 may be summarized as follows: ERROR(1) - severity 2, unrecognized statement; ERROR(2) - severity 2, unexpected '^a' punctuation mark in a name list; ERROR(3) - severity 2, invalid name '^a' in a Name list; ERROR(4) - severity 3, incomplete statement; ERROR(5) - severity 2, invalid attribute '^a' in an Attribute list; ERROR(6) - severity 2, invalid number '^a' in a Value list; and ERROR(7) - severity 3, unexpected '^a' when a punctuation mark was expected in a name list.

Table 3: Elements of the Reduction Language

### Attribute Declarations

```
MAX_DEPTH n \
```

### Reduction Statements

```
labels  /  syntax           / actions        / next-reduction \

BEGIN   /                   /                /                \
        /                   /                /                \
label
label2  /                   /                /                \

        / absolute spec     /                /                \
        / <relative_fcn>    /                /                \
        /                   /                /                \
        / <no-token>        /                /                \
        / <any-token>       /                /                \
        / <name>            /                /                \
        / <decimal-integer>
                            /                /                \
        / <quoted-string>/                   /                \
        / <BS>              /                /                \

        /                   / semant(...)    /                \
        /                   / [var="1"b]     /                \
        /                   /                /                \
        /                   / LEX            /                \
        /                   / LEX(n)         /                \
        /                   / NEXT_STMT      /                \
        /                   / ERROR(n)       /                \
        /                   / PUSH(label)    /                \
        /                   / POP            /                \

        /                   /                / label          \
        /                   /                /                \
        /                   /                / RETURN         \
        /                   /                / STACK          \
        /                   /                / STACK_POP      \
```

 (END)

Internal Interface
Administrative/User Ring
06/19/74

**Name:** lex_string_

lex_string_ provides a facility for parsing an ASCII character string into tokens (character strings delimited by break characters) and statements (groups of tokens). It supports the parsing of comments and quoted strings. It parses an entire character string during one invocation, creating a chain of descriptors for the tokens and statements in an area. The cost per token of lex_string_ is significantly lower than that of parse_file_ because the overhead of calling parse_file_ to obtain each token is eliminated. It is recommended for translators which deal with moderate to large amounts of input.

The descriptors generated when lex_string_ parses a character string can be used as input to translators generated by the reduction_compiler command, as well as in other applications. In addition, the information in the statement and token descriptors can be used in error messages printed by the lex_error_ facility.

Refer to the writeups for the reduction_compiler and lex_error_ for details on the use of these facilities.

**Entry:** lex_string_$init_lex_delims

This entry constructs two character strings from the set of break characters and comment, quoting, and statement delimiters: one string contains the first character of every delimiter or break character defined by the language to be parsed; the second string contains a character of control information for each character in the first string. These two character strings form the break tables which lex_string_ uses to parse an input string. It is intended that these two (delimiter and control) character strings be internal static variables of the program which calls lex_string_, and that they be initialized only once per process. They can then be used in successive calls to lex_string_$lex, as described below.

__Usage__

declare lex_string_$init_lex_delims entry (char(*), char(*), char(*), char(*), char(*), bit(*), char(*) varying aligned, char(*) varying aligned, char(*) varying aligned, char(*) varying aligned);

call lex_string_$init_lex_delims (quote_open, quote_close, comment_open, comment_close, statement_delim, Sinit, break_chars, ignored_break_chars, lex_delims, lex_control_chars);

1) quote_open — is the character string delimiter which is to indicate the beginning (or opening) of a quoted string. It may be up to four characters in length. If it is a null character string, then quoted strings are not supported during the parsing of a character string. (Input)

2) quote_close — is the character string delimiter which is to indicate the ending (or closing) of a quoted string. It may be the same character string as quote_open, and may be up to four characters in length. (Input)

3) comment_open — is the character string delimiter which is to indicate the opening of a comment. It may be up to four characters in length. If it is a null character string, then comments are not supported during the parsing of a character string. (Input)

4) comment_close — is the character string delimiter which is to indicate the closing of a comment. It may be the same character string as comment_open, and may be up to four characters in length. (Input)

5) statement_delim — is the character string delimiter which is to indicate the closing of a statement. It may be up to four characters in length. If it is a null character string, then statements are not delimited during the parsing of a character string. (Input)

6) Sinit — is a bit string which controls the creation of statement descriptors, and the creation of token descriptors for quoting delimiters. The bit string consists of two bits in the order listed below. (Input)

Ssuppress_quoting_delims
is "1"b if token descriptors for the quote opening and closing delimiters of a quoted string are to be suppressed. A token descriptor is still created for the quoted string itself, and the quoted_string switch in this descriptor is turned on. If Ssuppress_quoting_delims is "0"b, then token descriptors are returned for the quote opening and closing delimiters, as well as for the quoted string.

Ssuppress_stmt_delims
is "1"b if the token descriptor for a statement delimiter is to be suppressed. The end_of_stmt switch in the descriptor of the token which precedes the statement delimiter is turned on, instead. If Ssuppress_stmt_delims is "0"b, then a token descriptor is returned for a statement delimiter, and the end_of_stmt switch in this descriptor is turned on.

7) break_chars — is a character string containing all of the characters which may be used to delimit tokens. The string may include characters used also in the quoting, comment, or statement delimiters, and should include any ASCII control characters which are to be treated as delimiters. (Input)

8) ignored_break_chars
is a character string containing all of the break_chars which may be used to delimit tokens, but which are not tokens themselves. No token descriptors are created for these characters. (Input)

9) lex_delims — is an output character string containing all of the delimiters which lex_string_ will use to parse an input string. This string is constructed by the init_lex_delims entry from the preceding arguments. It must be long enough to contain all of the break_chars, plus the first character of the quote_open delimiter, the comment_open delimiter, and the statement_delim delimiter, plus 30 additional characters. This length will not exceed 128 characters, the number of characters in the ASCII character set. (Output)

10) lex_control_chars — is an output character string containing one character of control information for each character in lex_delims. This string is also constructed by init_lex_delims from the preceding arguments. It must be as long as lex_delims. (Output)

<u>Entry</u>: lex_string_$lex

This entry parses an input string, according to the delimiters, break characters, and control information given as its arguments. The input string consists of two parts: the first part is a set of characters which are to be ignored by the parser, except for the counting of lines; the second part are the characters to be parsed. It is necessary to count lines in the part which is otherwise ignored so that accurate line numbers can be stored in the token and statement descriptors for the parsed section of the string.

<u>Usage</u>

declare lex_string_$lex entry (ptr, fixed bin(21), fixed bin(21), ptr, bit(*), char(*), char(*), char(*), char(*), char(*), char(*) varying aligned, char(*) varying aligned, char(*) varying aligned, char(*) varying aligned, ptr, ptr, fixed bin(35));

call lex_string_$lex entry (Pinput, Linput, Lignored_input, Parea, Slex, quote_open, quote_close, comment_open, comment_close, statement_delim, break_chars, ignored_break_chars, lex_delims, lex_control_chars, Pfirst_stmt_desc, Pfirst_token_desc, code);

1) Pinput — is a pointer to the string to be parsed. (Input)

2) Linput — is the length (in characters) of the second part of the input string, the part which is actually to be parsed. (Input)

3) Lignored_input — is the length (in characters) of the first part of the input string, the part which is ignored except for line counting. This length may be 0 if none of the input characters are to be ignored. (Input)

4) Parea — is a pointer to an area formatted by the area_ subroutine. (Input)

5) Slex — is a bit string which controls the creation of statement and comment descriptors, and the handling of doubled quotes within a quoted string. The bit string consists of three bits in the order listed below. (Input)

Sstatement_desc — is "1"b if statement descriptors are to be created along with the token descriptors. If Sstatement_desc is "0"b, or if the statement delimiter is a null character string, then no statement descriptors are created.

Scomment_desc is "1"b if comment descriptors are to be created for any comments which appear in the input string. If Scomment_desc is "0"b, if comment_open is a null character string, or if no statement descriptors are being created, then no comment descriptors are created.

Sretain_doubled_quotes
is "1"b if doubled quote_close delimiters which appear within a quoted string are to be retained. If Sretain_doubled_quotes is "0"b, then a quoted string containing doubled quote_close delimiters is copied into the area, and the doubled quote_close are changed to single quote_close delimiters.

Sequate_comment_close_stmt_delim
is "1"b if the comment_close and statement_delim character strings are the same, and if the closing of a comment is to be treated as the ending of the statement containing the comment. It could be used when parsing line-oriented languages which have only one statement per line and one comment per statement.

6) - 12) are as above. (Input)

13) lex_delims is the character string initialized by lex_string_$init_lex_delims. (Input)

14) lex_control_chars
is the character string initialized by lex_string_$init_lex_delims. (Input)

15) Pfirst_stmt_desc
is a pointer to the first in the chain of statement descriptors. This is a null pointer on return if no statement descriptors have been created. (Output)

16) Pfirst_token_desc
is a pointer to the first in the chain of token descriptors. This is a null pointer on return if no tokens were found in the input string. (Output)

17) code is one of the following status codes.

0 the parsing was completed successfully.

error_table_$zero_length_seg
no tokens were found in the input string.

error_table_$no_stmt_delim
the input string did not end with a statement delimiter, when statement delimiters were used in the parsing.

error_table_$unbalanced_quotes
the input string ended with a quoted string which was not terminated by a quote_close delimiter.

## Notes

Any character may be used in the quoting, comment, and statement delimiter character strings, including such characters as new line and the space character.

A quoted string is defined in the PL/I sense, as a string of characters which is treated as a single token, even though some of the characters may be delimiters or break characters. The string must begin with a quote_open delimiter, and must end with a quote_close delimiter. Two consecutive quote_close delimiters may be used to represent a quote_close delimiter within the quoted string. lex_string_$lex provides the option of retaining any doubled quote_close delimiters in the quoted string token, or of copying the quoted string into the area, changing double quote_close to single quote_close delimiters, and treating the modified copy as the quoted string token. Switches in the token descriptor of a quoted string are turned on: to indicate that the token was originally a quoted string; to indicate whether any quote_close delimiters appear within the quoted string; and to indicate whether doubled quote_close delimiters have been retained in the token.

Statements are defined as groups of tokens which are terminated by a statement delimiter token. lex_string_$lex can optionally return a token descriptor for the statement delimiter or it can suppress the statement delimiter's token descriptor. It always turns on the end_of_stmt switch in the final token descriptor of each statement, even if the statement delimiter's token descriptor has been suppressed. Also, it can optionally return a statement descriptor which points to the descriptors for the first and last tokens of a statement, contains a pointer to and the length of the statement, and describes various other characteristics of the statement. These descriptors are described in the next section.

Comments are defined in the PL/I sense, as a string of characters which begin with a comment_open delimiter, and which end with a comment_close delimiter. Comments which appear in the input string act as breaks between tokens. lex_string_$lex can optionally create descriptors for each comment which appears in a statement. These descriptors are chained off of the statement descriptor for that statement. Switches are set in each comment descriptor of a given statement to indicate whether the comment appears before any of the tokens in that statement, and whether any tokens intervene between this comment and any previous comments in that statement.

lex_string_ uses the smart_alloc_ subroutine to perform allocations in the PL/I area. When smart_alloc_ signals the area condition, it passes an information structure which describes the allocation which failed, and which can be used to cause the allocation to be reattempted in another area. Refer to the writeup on smart_alloc_ for more details.

## Descriptors

If lex_string_$lex were invoked to parse the input shown in Figure 1, using standard PL/I parsing conventions, then tokens and token descriptors created by lex_string_ would have the form shown in Figure 2.

```
Volume: 70092;
Write;
File 4;         /* Process 4th file on the tape.          */
/* END */
```

Figure 1: Sample Input to lex_string_

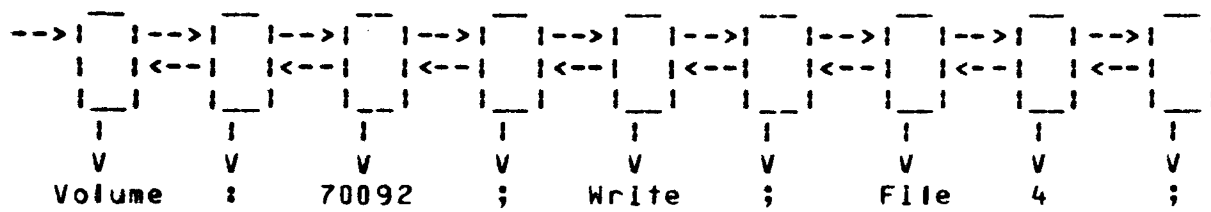


Figure 2: Input Tokens and their Descriptors

If statement descriptors were being created by lex_string_, then the output would have the form shown in Figure 3.

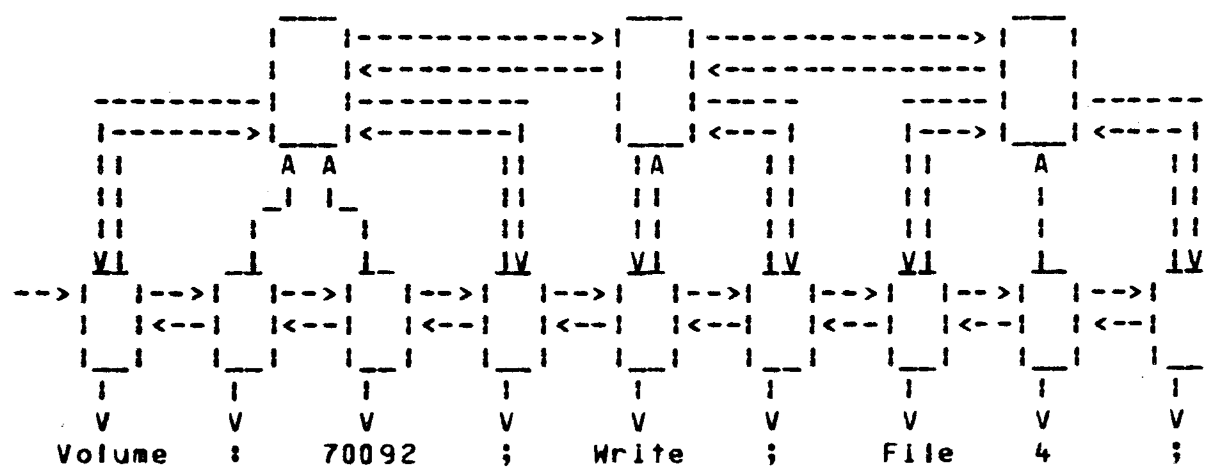


Figure 3: Tokens, Token Descriptors, and Statement Descriptors

Below is a declaration for the token descriptor structure.

```
declare
1 token aligned based (Ptoken),
  2 group1 unaligned,
    3 version fixed bin(17),
    3 size fixed bin(17),
  2 Pnext ptr unal,
  2 Plast ptr unal,
  2 Pvalue ptr unal,
  2 Lvalue fixed bin(18),
  2 Pstmt ptr unal,
  2 Psemant ptr unal,
  2 group2 unaligned,
    3 Itoken_in_stmt fixed bin(17),
    3 line_no fixed bin(17),
    3 Nvalue fixed bin(35),
    3 S,
      4 end_of_stmt bit(1),
      4 quoted_string bit(1),
      4 quotes_in_string bit(1),
      4 quotes_doubled bit(1),
      4 pad2 bit(32),
Ptoken ptr,
token_value char(token.Lvalue) based (token.Pvalue);
```

| | |
|---|---|
| 1) version | is the version number of the structure. The structure shown above is version 1. |
| 2) size | is the size of the structure, in words. |
| 3) Pnext | is a pointer to the descriptor for the next token in the input. If this is the last token descriptor, then the pointer is null. |
| 4) Plast | is a pointer to the descriptor for the previous token in the input. If this is the first token descriptor, then the pointer is null. |
| 5) Pvalue | is a pointer to the token character string. |
| 6) Lvalue | is the length of the token character string, in characters. |

| | |
|---|---|
| 7) Pstmt | is a pointer to the statement descriptor for the statement which contains this token. If statement descriptors are not being created, then this pointer is null. |
| 8) Psemant | is a pointer available for use by lex_string_'s caller. It might be used to chain a structure defining the semantic value of the token to the token's descriptor. |
| 9) Itoken_in_stmt | is the position of the token with respect to the other tokens in the statement containing this token. If no statement delimiters are being used in the parsing, then this is the position of the token with respect to all other tokens in the input. |
| 10) line_no | is the line_no on which this token appears. |
| 11) Nvalue | is a number available for use by lex_string_'s caller. It might be set to the numeric value of a token which is the character string representation of an integer. |
| 12) end_of_stmt | is "1"b if this is the last token of a statement. |
| 13) quoted_string | is "1"b if this token appeared in the input as a quoted string. |
| 14) quotes_in_string | is "1"b is quote_close delimiters appear within this quoted string token. |
| 15) quotes_doubled | is "1"b if quote_close delimiters which appear in a quoted string token are still represented by doubled quote_close delimiters, rather than having been converted to single quote_close delimiters. |
| 16) pad2 | is available for use by lex_string_'s caller. |
| 17) Ptoken | is a pointer to a token descriptor. |

18) token_value — is the character string representation of the token described by the token descriptor pointed to by Ptoken.

Statement descriptors are declared by the structure shown below.

```
declare
1 stmt aligned based (Pstmt),
  2 group1 unaligned,
    3 version fixed bin(17),
    3 size fixed bin(17),
  2 Pnext ptr unal,
  2 Plast ptr unal,
  2 Pvalue ptr unal,
  2 Lvalue fixed bin(18),
  2 Pfirst_token ptr unal,
  2 Plast_token ptr unal,
  2 Pcomments ptr unal,
  2 Puser ptr unal,
  2 group2 unaligned,
    3 Ntokens fixed bin(17),
    3 line_no fixed bin(17),
    3 Istmt_in_line fixed bin(17),
    3 semant_type fixed bin(17),
    3 S,
      4 error_in_stmt bit(1),
      4 output_in_err_msg bit(1),
      4 pad bit(34),
Pstmt ptr,
stmt_value char(stmt.Lvalue) based (stmt.Pvalue);
```

1) version — is the version number of this structure. The structure declared above is version 1.

2) size — is the size of this structure, in words.

3) Pnext — is a pointer to the statement descriptor for the next statement. If this is the descriptor for the last statement, then this pointer is null.

4) Plast — is a pointer to the descriptor for the previous statement. If this is the descriptor for the first statement, then the pointer is null.

5) Pvalue — is a pointer to the character string representation of the statement as it appears in the input, excluding any leading new line characters or leading comments.

6) Lvalue — is the length of the character string represenation of the statement, in characters.

7) Pfirst_token — is a pointer to the descriptor of the first token in the statement.

8) Plast_token — is a pointer to the descriptor of the last token in the statement.

9) Pcomments — is a pointer to a chain of comment descriptors associated with this statement.

10) Puser — is a pointer available for use by lex_string_'s caller.

11) Ntokens — is a count of the tokens in this statement.

12) line_no — is the line number on which the first token of this statement appears in the input.

13) semant_type — is a number available for use by lex_string_'s caller. It might be used to classify the statement by its semantic type.

14) error_in_stmt — is "1"b if an error has occurred while processing this statement. This switch is never set by lex_string_, but it is set by lex_error_ when a statement descriptor is used to generate an error message.

15) output_in_err_msg — is "1"b if the statement has already been output in another error message. This switch is referenced and set by lex_error_ to prevent a statement from being printed in more than one error message.

16) pad — is available for use by lex_string_'s caller.

17) Pstmt — is a pointer to a statement descriptor.

18) stmt_value — is the character string value of the statement, as it appears in the input, excluding any leading new line characters or leading comments.

Comment descriptors are declared as follows.

```
declare
1 comment aligned based (Pcomment),
  2 group1 unaligned,
    3 version fixed bin(17),
    3 size fixed bin(17),
  2 Pnext ptr unal,
  2 Plast ptr unal,
  2 Pvalue ptr unal,
  2 Lvalue fixed bin(18),
  2 group2 unaligned,
    3 line_no fixed bin(17),
    3 S,
      4 before_stmt bit(1),
      4 contiguous bit(1),
      4 pad bit(16),
Pcomment ptr,
comment_value char(comment.Lvalue) based (comment.Pvalue);
```

1) version — is the version number of this structure. The structure declared above is version 1.

2) size — is the size of this structure, in words.

3) Pnext — is a pointer to the descriptor for the next comment associated with the statement containing this comment. If there are no more comments associated with that statement, then the pointer is null.

4) Plast — is a pointer to the descriptor for the previous comment associated with the statement containing this comment. If this is the first comment associated with the statement, then the pointer is null.

5) Pvalue — is a pointer to the character string value of the comment string, exactly as it appears in the input, excluding the comment_open and comment_close delimiters.

6) Lvalue — is the length of the character string value of the comment, in characters.

7) line_no — is the line number on which the comment begins.

8) before_stmt — is "1"b if the comment appears in its statement before any tokens.

9) contiguous — is "1"b if no tokens appear between this comment and the previous comment associated with this statement.

10) pad — is available for use by lex_string_'s caller.

11) Pcomment — is a pointer to a comment descriptor structure.

12) comment_value — is the character string value of a comment.

The above declarations are available for inclusion in PL/I programs in lex_descriptors_.incl.pl1.

```
/*++
MAX_DEPTH 20 \

BEGIN      / <no-token>                        / ERROR(1)                                  / stop       \
           / <any-token>                       / reductions_init                           / attributes\

attributes
           / BEGIN                             / [Psave = Pthis_token]                     / pass1      \
           / MAX_DEPTH <decimal-integer> "\"   / LEX set_depth LEX(2)                      / attributes\
           / <no-token>                        / ERROR(1)                                  / stop       \
           / <any-token>                       / ERROR(2) NEXT_STMT                        / attributes\

pass1
set_label  / ∠                                 / count_reduction LEX                       / count      \
           / <name>                            / set_label LEX                             / set_label\
           / "\"                               / ERROR(22) LEX                             / set_label \
           / <no-token>                        / [Pthis_token = Psave] reductions_begin    / pass2      \
           / <any-token>                       / ERROR(3) LEX                              / set_label\

count      / <quoted-string>                   / count_token(1) LEX(1)                     / count      \
           / ∠ <BS> _                          / count_token(3) LEX(3)                     / count      \
           / ∠                                 / NEXT_STMT                                 / set_label\
           / <any-token>                       / count_token(1) LEX                        / count      \
           / <no-token>                        / ERROR(5)                                  / stop       \

pass2
skip_label
label      / ∠                                 / reduction_begin LEX                       / tokens     \
           / <name>                            / LEX                                       / skip_label\
           / "\"                               / LEX                                       / skip_label\
           / <any-token>                       / LEX                                       / skip_label\
           / <no-token>                        /                                           / stop       \

tokens     / <quoted-string>                   / compile_token(0) LEX                      / tokens     \
           / ∠ <BS> _                          / compile_token(0) LEX(3)                   / tokens     \
           / ∠                                 / action_begin LEX                          / action     \
           / < <BS> _                          / compile_token(0) LEX(3)                   / tokens     \
           / > <BS> _                          / compile_token(0) LEX(3)                   / tokens     \
           / [ <BS> _                          / compile_token(0) LEX(3)                   / tokens     \
           / ] <BS> _                          / compile_token(0) LEX(3)                   / tokens     \
           / ( <BS> _                          / compile_token(0) LEX(3)                   / tokens     \
           / ) <BS> _                          / compile_token(0) LEX(3)                   / tokens     \
           / ≤ no-token ≥ ∠                    / compile_token(1) action_begin LEX(4)      / action     \
           / ≤ no-token ≥ <any-token>          / LEX(3) ERROR(14) [obj_red.Ilast(Nobj_red) = 0]
                                                 reduction_end NEXT_STMT                   / label      \
           / ≤ any-token ≥                     / compile_token(2) LEX(3)                   / tokens     \
           / ≤ name ≥                          / compile_token(3) LEX(3)                   / tokens     \
           / ≤ decimal-integer ≥               / compile_token(4) LEX(3)                   / tokens     \
           / ≤ BS ≥                            / compile_token(5) LEX(3)                   / tokens     \
           / ≤ quoted-string ≥                 / compile_token(6) LEX(3)                   / tokens     \
           / ≤ <name> ≥                        / LEX compile_token(7) LEX(2)               / tokens     \
           / "\"                               / ERROR(22) LEX                             / label      \
           / <any-token>                       / compile_token(0) LEX                      / tokens     \
           / <no-token>                        / ERROR(5)                                  / stop       \
```

```
action     / ∠                                / LEX                                                   / next_red\
           / LEX ( <decimal-integer> )        / LEX(2) rtn(1) LEX(2)                                  / action   \
           / LEX (                            / ERROR(19) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end NEXT_STMT                               / label    \
           / LEX                              / rtn(2) LEX                                            / action   \
           / NEXT_STMT (                      / ERROR(19) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end NEXT_STMT                               / label    \
           / NEXT_STMT                        / rtn(3) LEX                                            / action   \
           / POP (                            / ERROR(19) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end NEXT_STMT                               / label    \
           / POP                              / rtn(4) LEX                                            / action   \
           / PUSH ( <name> )                  / LEX(2) rtn(5) LEX(2)                                  / action   \
           / PUSH                             / ERROR(19) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end NEXT_STMT                               / label    \
           / ERROR ( <decimal-integer> )      / [Sinclude_ERROR = "1"b] set_action_with_args
                                                LEX(2) PUSH(last_paren)                               / args     \
           / ERROR                            / ERROR(19) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end NEXT_STMT                               / label    \
           / [                                / LEX(1) output((6)"          " || (5)" ")              / stmt     \
           / ]                                / ERROR(21) LEX                                         / action   \
           / (                                / ERROR(21) LEX                                         / action   \
           / )                                / ERROR(21) LEX                                         / action   \
           / <quoted-string>                  / ERROR(23) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end NEXT_STMT                               / label    \
           / "\"                              / ERROR(22) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end LEX                                     / label    \
           / <any-token> (                    / set_action_with_args LEX(2) PUSH(last_paren)          / args     \
           / <any-token>                      / set_action LEX                                        / action   \
           / <no-token>                       / ERROR(5)                                              / stop     \

stmt       / <quoted-string>                  / output("""") output(token_value) output("""")
                                                LEX                                                   / stmt     \
           / (                                / output("(") LEX PUSH(stmt)                            / args     \
           / ]                                / LEX                                                   / last_paren \
           / ;                                / output (";" || NL || (6)"    " || (5)" ")
                                                LEX                                                   / stmt     \
           / "\"                              / ERROR(24) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end LEX                                     / label    \
           / <any-token>                      / output(token_value) output(" ") LEX                   / stmt     \
           / <no-token>                       / ERROR(5)                                              / stop     \

args       / <quoted-string>                  / output("""") output(token_value) output  ("""")
                                                LEX                                                   / args     \
           / (                                / output("(") LEX PUSH(args)                            / args     \
           / )                                / output(")") LEX                                       / STACK_POP \
           / "\"                              / ERROR(24) [obj_red.Ilast(Nobj_red) = 0]
                                                reduction_end LEX                                     / label    \
           / <any-token>                      / output(token_value) LEX                               / args     \
           / <no-token>                       / ERROR(5)                                              / stop     \

last_paren/                                   / output(";") output(NL)                                / action   \
```

```
next_red  / "\"                       / next_reduction reduction_end LEX                   / label    \
          / RETURN "\"                / terminal_reduction reduction_end LEX(2)            / label    \
          / STACK "\"                 / stacked_reduction reduction_end LEX(2)             / label    \
          / STACK_POP "\"             / stacked_reduction_with_pop reduction_end LEX(2)    / label    \
          / <name> "\"                / specified_label reduction_end LEX(2)               / label    \
          / <name>                    / specified_label reduction_end ERROR(15)
                                        NEXT_STMT                                          / label    \
          / <any-token> "\"           / ERROR(4) next_reduction reduction_end NEXT_STMT    / label    \
          / <any-token>               / ERROR(15) next_reduction reduction_end NEXT_STMT/  label    \
          / <no-token>                / ERROR(5)                                           / stop     \

stop      / <no-token>                / reductions_end                                     / RETURN   \
          / <any-token>               / ERROR(5) reductions_end                            / RETURN   \
                                                                                                      ++*/
```