DATE: 10 December 1987

TO: R D & E Personnel

FROM: David Spector

SUBJECT: The DEREMER Parser Generator

REFERENCE: None

KEYWORDS: LANGUAGES

ABSTRACT

This document describes the DEREMER Parser Generator. DEREMER is used to create parsers (recognizers) for programming or control languages, given a context-free grammar in the form of BNF productions. It uses Frank DeRemer's original Not-Quite LALR(1) algorithm.

DEREMER is compatible with and produces output for the PLP, PL1G, PL1, SPL, and various C compilers.

DEREMER is compatible with all supported PRIMOS revisions.

DEREMER was written by Robert Schwartz with modifications by Louis Tsien, and more recently by David Spector and Garth Conboy.

This PE-T replaces PE-T-418.

This PE-T describes Rev. 21.0 (30 NOV 87) of DEREMER.

Table of Contents

I Introd

What is D

Stated a

Impact a

of other

a Para

ication

arrang

ich

y pric

Ex

.ter

ample

ionary

1 Introduction

What is DEREMER?

Stated as simply as possible, DEREMER is a program which takes as its input a file containing

(1) A grammar for an assembler, compiler, or interpreter, or other kind of language.

(2) Other information which specifies semantic processing required by the application.

and produces as output a source code file containing a procedure to parse and/or interpret input strings or files according to the supplied grammar, performing the specified semantic actions. This procedure, which we call the parser, can be compiled as part of a larger system. The parser may be produced in a format appropriate for PLP, PL1G, PL1, SPL, C, or CC.

You, the reader, are assumed to have a nodding acquaintance with context free grammars and the Backus-Naur notation (BNF) in which such grammars are expressed. To refresh your memory, a context free language is specified by a context free grammar consisting of

(1) a set of terminal symbols, or tokens, which are the basic elements forming an input string,

(2) a set of nonterminal symbols, which represent the syntactic classes defined by the grammar, and

(3) a set of rules, or productions, which define each nonterminal in terms of instances of terminal and nonterminal symbols. Each rule contains a left hand side, the nonterminal being defined, the BNF metasymbol '::=', and a right hand side, consisting of a string of terminals and nonterminals from which the left hand can be formed.

One specific nonterminal, the start symbol, represents the class of strings which is defined by the entire grammar.

The usage of DEREMER is best explained "backwards", beginning with a brief overview of how the parser operates. The parser reads input by calling a programmer-supplied lexical procedure, or lexer. The lexer's task is to read the actual input stream and produce tokens which are returned to the parser. Tokens consist of a syntactic type, coded as an integer, and a semantic value, which is a pointer carried along by the parser but used only by the user application in whatever way it desires. These tokens are shifted, or pushed onto an internal stack, until the entire right hand side of some grammar rule appears on the stack. At this time a reduction is performed, whereby the symbols forming the right hand side of the production are removed from the stack and replaced by the nonterminal on the left hand side of the production. This operation is also known as recognizing the

production. When a reduction is performed, user application code is executed to calculate the semantic value (generally a function of the semantic values of the symbols on the right hand side of the production) associated with the resulting nonterminal.

This process continues until the parser reaches a final state where the start symbol is the only symbol on the parser stack and no further shift is possible. The parser then accepts the input string, and returns the value associated with the start symbol.

If a point is reached where the current lookahead symbol is illegal, then the parser has detected a syntax error. Error recovery is described in detail under Parser Error Handling below.

2 Input Format

This section describes the format of DEREMER input files.

In order to distinguish between concepts and literal character strings, the notation used throughout this document follows these general rules:

(1) An UPPER CASE identifier is a literal, i.e. stands for itself. This does not necessarily mean that the identifier must actually be written in upper case.

(2) A non-alphabetic character usually stands for itself. In the text, it usually is enclosed in single quotes (').

(3) A symbol given in lower case letters often stands for a class of objects. Usually the symbol itself is suggestive of this.

With these preliminaries taken care of, a short example of a DEREMER input file is given on the next page.

Note that the SPL language is used in all examples. Note that nonterminals are enclosed in angle brackets ('<>').

2.1 Example: CALC1.SPL.DEREMER

```
/* An interpreter to do arithmetic calculations on single digits */
%PARSER Calc1 (input_string);

%DECLARATIONS
declare input_string char (*) varying;
     /* Parameter to the parser procedure */
declare input_position fixed bin;
declare based_integer fixed bin (31) based;
declare rank builtin;
     /* A function to return the ASCII code for the character */
%Include 'Calc1.ins.pl1';
%END_DECLARATIONS;

%TOKEN plus_ , star_ , left_paren_ , right_paren_ , digit_, end_;

%RULES;
<input_line>  ::=  <expression>  end_
;
<expression>  ::=  <expression>  plus_  <term>
                   %ACTION
                   $$ -> based_integer = $1 -> based_integer +
                                         $3 -> based_integer;
                   free $3 -> based_integer;
                   %END_ACTION
                |  <term>    /* Default action is '$$ = $1;' */
;
<term>  ::=  <term>  star_  <primary>
                   %ACTION
                   $$ -> based_integer = $1 -> based_integer *
                                         $3 -> based_integer;
                   free $3 -> based_integer;
                   %END_ACTION
          |  <primary>
;
<primary>  ::=  digit_
             |  left_paren_  <expression>  right_paren_
                   %ACTION
                   $$ = $2;
                   %END_ACTION
;

%INIT
input_position = 1;
%END_INIT;

%PROGRAMS
lexer:
   procedure (type, value_ptr);
      dcl type fixed bin (15);
      dcl value_ptr pointer;
      dcl current_char char (1);
```

```
      /* Value returned will be undefined by default */
      value_ptr = null ();

      /* Return 'end_' token at end of string */
      if input_position > length (input_string)
      then
      do;
         type = end_;
         return;
      end;

      /* Get next character */
      current_char = substr (input_string, input_position, 1);
      input_position = input_position + 1;

      /* Return proper token for current character */
      select (current_char);
         when ('+') type = plus_;
         when ('*') type = star_;
         when ('(') type = left_paren_;
         when (')') type = right_paren_;
         when ('0','1','2','3','4','5','6','7','8','9')
         do;
            type = digit_;
            allocate based_integer set (value_ptr);
            value_ptr -> based_integer = rank (current_char)
                                       - rank ('0');
         end;
         otherwise type = 0;       /* To detect illegal characters */
      end;      /* end of select statement */
   end lexer;  /* end of PROGRAMS section */
%END_PROGRAMS;

%END_PARSER;
```

This example is the DEREMER input file defining a calculator-like interpreter for expressions involving single numeric digits, '+' and '*' operators, and parentheses.

A DEREMER input file consists of sections of text delimited by keywords. Each keyword begins with a '%'. (Although DEREMER does not require keywords to be spelled in all capital letters, the example has them capitalized in order to make them more visible.) Keywords do not have to start in column one: the input is completely free-format. PL/1 style comments, delimited by /* and */, may be interspersed freely within the text. The semicolons following keywords are significant, although somewhat arbitrary.

The non-comment text of the file is bracketed by the keywords %PARSER and %END_PARSER;.

The parser is produced as a function that returns a pointer. The desired external procedure name for the parser follows the keyword %PARSER. If the parser is to take arguments, the argument list follows the parser name; otherwise the argument list is omitted. In either case, a semicolon is required. For instance, the input line

```
%PARSER Calc1 (input_string);
```

results in the statement

```
Calc1: procedure (input_string) returns (ptr);
```

whereas

```
%PARSER Calc1;
```

produces

```
Calc1: procedure returns (ptr);
```

The returned pointer is not used by DEREMER, but is useful for implementing the semantics (meaning) of the application. In particular, a calculator (such as our example above) might use it to return a pointer to a numeric value, while a compiler might use to return a pointer to a parse tree.

Note that the you must supply a procedure called 'lexer' having two arguments. The first argument is the returned token type value, a fixed bin (15). This represents the token for parsing purposes. The second argument is the returned extended token value, a pointer. This represents any value associated with the token to be used in semantic actions. The lexer is described later in this document.

Between the %PARSER and %END_PARSER keywords, there appear a number of sections, each headed by a unique keyword--here we see the keywords %DECLARATIONS, %TOKEN, %RULES, %INIT, and %PROGRAMS. These sections can occur in any order. Any unneeded sections may be omitted.

The %DECLARATIONS, %INIT, and %PROGRAMS sections are very similar in function. Each has a matching end keyword (%END_DECLARATIONS;, %END_INIT;, and %END_PROGRAMS;) which includes a semicolon. Each pair of keywords encloses a section of literal text which is copied directly to the parser. The %DECLARATIONS and %INIT texts are copied to the beginning of the parser, and are most suitable for declarations of variables to be used in semantic processing, %Replace and %Include constructs, initialization statements, and so forth. The %PROGRAMS text is copied toward the end of the parser, and is intended mainly for internal procedures used by your application (semantic) code.

The %TOKEN section consists of one or more statements headed by the keyword %TOKEN. Other keywords (%LEFT, %RIGHT, and %NONASSOC) can be used to specify precedence and associativity for given tokens--this will be explained later under Advanced Features. The keyword %TOKEN is followed by a list of one or more token names, separated by commas and

terminated by a semicolon. An error message results from using an undeclared token name in a grammar rule. You may assign your own token type values, as in the following example:

```
%TOKEN plus_=6, minus_, identifier_=42, constant_=107;
```

If you don't assign a value, DEREMER assigns the previous value plus one. Thus, 'minus_' is given the value 7. The default starting value is one, and numeric values less than one are illegal, as are repeated values.

A legal token name is any symbol which would also be a legal PLP, PL1G, PL1, SPL, C, or CC variable name: a string of 32 or fewer characters, beginning with an alphabetic, where the succeeding characters, if any, come from the set {A-Z, a-z, 0-9, _, $}, except that "$" is not a legal identifier character in standard C compilers.

The %RULES; section contains the rules for the grammar and their associated semantic actions. Each rule consists of a left hand side, which is a nonterminal symbol, followed by the BNF metasymbol '::=', followed by 1 or more alternative right hand sides separated by either vertical bar '|' or exclamation '!'. The last alternative is terminated by a semicolon. Each alternative right hand side for a given rule consists of a (possibly empty) sequence of terminal and nonterminal symbols. A nonterminal symbol is one which is enclosed in angle brackets '<>' or square brackets '[]'. Additional brackets may appear nested within the outermost brackets, as long as each left bracket has a matching right bracket. The square brackets are useful for suggesting an optional item, although DEREMER treats square brackets and angle brackets alike in generating a parser. Note that upper and lower case are distinguished for symbols, unlike keywords.

There may be more than one production having the same left hand side. This is equivalent to using alternative right hand sides.

Each right hand side of a grammar rule may have an action associated with it. The action is a piece of text delimited by the keywords %ACTION and %END_ACTION which follows the terminal and nonterminal symbols of the right hand side. Within the action, the string '$$' is used to denote the value to be associated with the left hand side of the grammar rule when this reduction takes place. A string of the form '$n', where n is a digit between 1 and 9 inclusive, denotes the value associated with the nth component of the right hand side. Each of these values is of datatype pointer. In the 'Calc1' grammar above, the values point to allocated fixed bin (31) variables used to store integers. Note that the $$ and $n constructs are not available in any of the literal (code) sections other than %ACTION sections, because they would be inefficient to support and/or conceptually difficult to understand outside of the semantic action source code associated with an individual production.

When a given right hand side of a grammar rule has no explicit action, the left hand side inherits the value of its first right hand side component. This is equivalent to the action

```
%ACTION
$$ = $1;
%END_ACTION
```

This is particularly convenient when the right hand side contains exactly one symbol.

The assignment '$$ = $1;' takes place before any action is executed.

DEREMER takes the nonterminal which is the left hand side of the first rule to be the start symbol for the grammar.

2.2 DEREMER Directives

Here is a complete list of DEREMER directives. Square brackets indicate optional syntax.

/* ... */ : A comment. Comments may be used freely throughout DEREMER input files and are ignored by DEREMER, except that comments in literal code sections are copied intact to the generated parser.

%PARSER parser_name [(argument_list)]; ... %END_PARSER; : Defines a DEREMER input file. This directive must be present and must bracket any and all other directives (it may be preceded and followed by comments). See explanation just after the Calc1 example above.

%COMMENTS ... %END_COMMENTS; : Defines a section containing comments to be placed at the beginning of the generated parser. Note that all text between the 'S' in %COMMENTS and the '%' in %END_COMMENTS;, including Newlines, is copied to the generated parser. This section allows the Prime Standard File Header to be specified for the generated parser.

%DECLARATIONS ... %END_DECLARATIONS; : Defines the Declarations section. Contains declarations needed by your application.

%INIT ... %END_INIT; : Defines the Initialization section. Contains application code to be executed when the parser is called, prior to parsing.

%PROGRAMS ... %END_PROGRAMS; : Defines a code section containing procedures called by application code anywhere in the parser.

%ACCEPT ... %END_ACCEPT; : Defines a section executed when the parser accepts its input (recognizes the start symbol).

%CHECK_STACK; : Generates extra parser code to check for stack overflow. See the section Parse Stack Overflow below.

%ERROR ... %END_ERROR; : Defines a section executed when the parser encounters an error (a token is found that is not correct according to the grammar rules). This section executes before any SYNTAX_ERROR semantic action. See Parser Error Handling for more details.

%SHIFT ... %END_SHIFT; : Defines a section executed when the parser shifts each input token onto the parse stack.

%REDUCE ... %END_REDUCE; : Defines a section executed when the parser makes each reduction (recognizes each grammar rule).

%TOKEN token_name [= number], ... ; : Declares a list of tokens (terminal symbols) not having a precedence or associativity.

%LEFT ... ; : Declares a list of token (terminal symbol) names with the left-associative attribute and a higher precedence than all previous declarations (see Advanced Features below).

%RIGHT ... ; : Declares a list of token (terminal symbol) names with the right-associative attribute and a higher precedence than all previous declarations (see Advanced Features below).

%NONASSOC ... ; : Declares a list of token (terminal symbol) names having no associativity but a higher precedence than all previous declarations (see Advanced Features below).

%RULES; ... : Defines the section containing the grammar rules, expressed as BNF productions.

%ACTION ... %END_ACTION : Defines a semantic action associated with the preceding grammar rule. The action will be executed when the rule is recognized.

%PREC token_name : Overrides the default precedence of a grammar rule (that of its rightmost terminal symbol) with the precedence of the specified token (see Advanced Features below).

%TABLE table_name; : Creates a file 'name.TABLES' containing the parser tables in .BIN format. The name "table_name" is used as an external static structure name in connection with these tables. The standard "%TABLE table_name;" form of the %TABLE directive is not supported in the C language modes.

The "%TABLE" directive no longer truncates the name of the external symbol to 8 characters when emitting the parse tables to a binary file.

The variant "%TABLE external;" causes the parse tables to be emitted normally (in the generated parser), but their storage class will be "static external" rather than "static internal". For this option to work correctly the generated parser must be loaded with "BIND" to allow resolution of long external-symbol

names.

The variant "%TABLE constant;" causes SPL language parse tables to be declared using "options(constant)". This causes the parse tables to be allocated in the procedure frame rather than in the linkage frame, thus allowing easy sharing of these large data areas.

%NO_DEFAULT_ACTION; : Disables the default actions "$$ = $1" (when there exists at least one right-hand-side symbol) or "$$ = null()" ("$$ = 0" for C). This slightly increases the efficiency of generated parsers at the expense of not allowing simple %ACTION clauses to be omitted. If this option is used then an action must be specified for each reduction.

%ARGUMENT_DCLS ... %END_ARGUMENT_DCLS; : Specifies argument declarations for the generated parser. Applies only to the C language; see the "Notes for C Language Support" section.

%STACK stack_length [EXTERNAL]; : Specifies a length for the parser stacks (the default is 200). If the 'EXTERNAL' option is specified, the stacks are declared with storage class 'external static'. The declaration corresponding to the directive '%STACK 300 external;' would be:

```
dcl 1 dp$_stacks external static,
        2 dp$_state_stack(300) bin,
        2 dp$_state_stack_ptr  bin,
        2 dp$_symbol_stack(300) ptr,
        2 dp$_symbol_stack_ptr  bin;
```

$Insert pathname : Includes the named file at this point in the DEREMER input file. If an entryname is given, the file is found by using the INCLUDE$ PRIMOS search rules. See PE-T-1204 for a preliminary description of search rules. $Insert must begin in column 1 and only the characters '$I' are significant (the 'I' must be upper case). $Inserted files must not be nested. $Insert occurring in any of the literal (code) sections is not expanded, but is copied verbatim to the parser. This supports programming languages in which $Insert is recognized and is to be expanded at compile time.

%SYSTEM option; : Used in specifying multiple parsers. Described under Advanced Features below.

Unneeded directives or sections may be omitted.

3 Invoking DEREMER

DEREMER is invoked by a command line of the form

```
DEREMER input_pathname {command_options}
```

where the input file has a name of the form 'name.lang.DEREMER', of which only 'name.lang' need be specified as the "input_pathname" above. The 'lang' indicates the desired language and file name suffix for the generated parser.

The command options can be chosen from the following list (where "name" is the entryname portion of "input_pathname" with any 'lang.DEREMER' or 'DEREMER' suffix removed, and "lang" is the desired language--PLP, PL1G, PL1, SPL, C, or CC):

-grammar, -grm: Create a file 'name.GRAMMAR' containing a formatted listing of the input grammar (rules section) without the actions.

-fsa: Create a file 'name.FSA' containing a listing of the parser (Finite-State Automaton) states. The format of FSA files is described under Debugging a Parser: How It Works below.

-debug: Create the parser output file with the name 'name.DEBUG.lang' instead of 'name.lang' and provide a run-time trace of parsing via calls to system subroutine IOA$ (see PE-T-364 for a description of IOA$).

-externals, -ext: Remove the 'dp$_token_name' function (returns the character string representation of a token, given its token value) from the parser ('name.lang' file) and place it in a file called 'name.EXTERNALS' instead.

-pl1, -pl1g, -spl, -plp, -c, -cc: Specify the compiler that will be used to compile the parser. This affects the use of certain language features, such as 'select' vs. 'goto', as well as determining the file suffix ("lang") used.

The default language is PLP.

-C produces a C language parser with a .C suffix.

-CC produces a C language parser with a .CC suffix (content is otherwise the same as for -C).

Note that these options are not necessary, because the input file name can be of the form 'name.lang.DEREMER' to specify the desired output language automatically.

-no_parser, -npar: Suppress creation of the parser ('name.lang' file).

-no_actions, -nact: Delete semantic action code from the parser ('name.lang' file).

-no_sr_conflicts, -nsrc: Suppress warning messages for all shift-reduce conflicts.

-noerrtty: will cause most error messages (other than those for fatal errors) to be written in a file named 'name.ERROR' in the current directory, instead of being displayed on the terminal. This option is unsafe to use; it is meant only for in-house use when building COBOL (which has an erroneous grammar), using a previously-stored "norm" file to detect unexpected errors, by doing an explicit file comparison operation after invoking DEREMER.

Further discussion of many of the features mentioned above is made throughout this document.

DEREMER produces the following two main output files, where "name" again is the entryname portion of "input_pathname" with any '.lang.DEREMER' or '.DEREMER' suffix removed, and "lang" is the desired language--PLP, PL1G, PL1, SPL, C, or CC (C and CC differ only in the suffix; CC is provided for historical reasons).

name.lang: The parser, including the contents of %PROGRAMS and other literal text sections.

name.INS.PL1: An %Include file containing %Replace statements to define token names, for use by the lexer and other application code.

The file 'name.lang' is the output source file described in the Introduction. The file 'name.INS.PL1' contains definitions of the integer codes assigned by DEREMER (or the user) for the various tokens. In our calculator example, the token 'plus_' is given type code 1 (because it is the first token declared in CALC1.SPL.DEREMER). The token definition file CALC1.INS.PL1 contains a line

```
%replace plus_ by 1;
```

This format allows the programmer to %Include the name.INS.PL1 file within the lexer, or within the %DECLARATIONS section, causing all token type codes to be defined consistently between the lexer and parser.

The following additional output files may be created, when specified by command options or DEREMER directives:

name.GRAMMAR: A formatted listing of the input grammar (rules section) without the actions.

name.FSA: A listing of the parser (Finite-State Automaton) states. See Debugging a Parser: How It Works below for a description of the format used.

name.EXTERNALS: The 'dp$_token_name' function (returns the character string representation of a token, given its token value).

name.TABLES: The parser tables (in .BIN format), when the %TABLE directive appears in the input file. See the description of the %TABLE directive under DEREMER Directives above.

All output files are created in the current (attached) directory.

It cannot be emphasized too strongly that only the latest revision of DEREMER is to be used (it is always extended compatibly). Do not copy CMDNC0>DEREMER.SAVE (or DEREMER.RUN if a changeover is made to EPFs) for local use unless you are sure to recopy it when new versions are installed.

Prime Engineering users must accept responsibility for checking that the latest version of DEREMER is reinstalled after a system upgrade. This is necessary because a system upgrade replaces DEREMER by the latest Master Disk version, which is not always the latest version available in-house.

4 Notes for C Language Support

The generation of parsers in the C language (and the following documentation) has been provided by Garth Conboy of Pacer Software, Inc. At the present time, this feature is available on an "as-is" basis. Potential users must evaluate the suitability of the generated C parsers for their applications.

Both DEREMER language modes C and CC produce parsers in the C language. The difference is that -C (or an input file suffix .C.DEREMER) specifies that the suffix of the generated files is to be .C, while -CC (or an input file suffix .CC.DEREMER) specifies that the suffix of the generated files is to be .CC.

Because standard C compilers do not recognize "$" as a legal character in identifiers, the DEREMER internal identifier prefix "dp$_" is changed for C (and CC) language to "dp__".

The lexer that will be called by the C parser has an additional level of indirection for both of its two arguments to account for C's pass-by-value nature. The DEREMER generated call would be:

```
short dp__tkntyp;
int *dp__tknptr;
        .
        .
lexer(&dp__tkntyp, &dp__tknptr);
```

All returned lexical values must be word-aligned pointers because dp__tknptr is declared "int *" rather than "char *" for efficiency.

Thus, part of a correct C lexer would be:

```
void lexer(lexcode_ptr, lexval_ptr)
  short *lexcode_ptr;
  int   **lexval_ptr;
{
        .
        .
  *lexcode_ptr = Integer_constant;
  *lexval_ptr  = (int *)malloc(sizeof(int));
  **lexval_ptr = Integer_value;
  return;
}
```

The generated parser will be declared to return an "int *" (pointer to integer); this will be the lexical value of the final reduction. On error returns from the parser, the C null pointer will be returned (0).

The semantic stack (the values of $$, $1, etc.) in C generated parsers is declared as an array of "int *"s. In the general case, however, the actual values that are to be put on the semantic stack will be pointers to complex structures rather than pointers to integers. Thus, casts will often be needed to access the semantic stack. For example, assuming the following declarations:

```
typedef struct expTag {short expressionType;
                       short expressionOperation;
                       long pointedToSize;
                       int isAnLvalue:1,
                           parenthesized:1;
                       struct expTag *son1;
                       struct expTag *son2;
                       struct expTag *son3;
                      } *Expression;

Expression Parenthesized();
```

The following excerpt of DEREMER source will correctly deal with a semantic stack containing these nodes:

```
<Expression> ::= LParen <Expression> RParen
   %action
        (Expression)$$ = Parenthesized($2);
        (Expression)$$ -> son1 = (Expression)$2;
        (Expression)$$ -> parenthesized = True;
        (Expression)$$ -> isAnLvalue = False;
   %end_action ;
```

If a C parser takes no arguments then the "%PARSER" directive must be:

```
%PARSER parser_name();
```

Note that there is no test for the presence of the empty parentheses, but they must be there!

If arguments are expected then they should be placed in the "%PARSER" directive, as with other languages. However the argument declarations must appear between "%ARGUMENT_DCLS" and "%END_ARGUMENT_DCLS;" clauses. For example:

```
%PARSER parser_name(string, len);

%ARGUMENT_DCLS
  char *string;
  int len;
%END_ARGUMENT_DCLS;
```

The "%ARGUMENT_DCLS" directive may only be used when generating a C parser.

5 Parser Error Handling

DEREMER provides three distinct error handling facilities:

1. Returning a null pointer (the default).
2. Executing an %ERROR section.
3. Processing a SYNTAX_ERROR symbol and executing its %ACTION, if any. These facilities are described below.

5.1 Default Error Handling

In the absence of any grammar rules having SYNTAX_ERROR on the right hand side, the parser returns a null() after having read as far as is necessary to determine that a syntax error exists. No attempts are made to recover from the error, or to continue the parse. The procedure(s) which call the parser must be prepared for null() to be returned, and must do the error reporting, if any. For interactive systems, some means of determining the point of failure within the input string may be sufficient.

5.2 The %ERROR Section

The directives %ERROR and %END_ERROR; define a section executed when the parser encounters an error (a token occurs that is not correct according to the grammar rules). This section executes before any SYNTAX_ERROR semantic action (these are described below).

The following variables and functions are available within this section to aid in handling errors:

Identifier	Declaration	Description
dp$_tkntyp	fixed bin (15)	Current token type
dp$_tynptr	pointer	Current token value
dp$_current_state	fixed bin (15)	Current state number

```
dp$_number_of_actions entry (bin)        Converts state number to
                      returns (bin)        number of possible shifts
dp$_nth_action       entry (bin, bin)    Converts state number and
                 returns (char(32)varying)    n to nth shift's token
```

IMPORTANT NOTE: For the C (and CC) languages, the DEREMER internal identifier prefix "dp$_" in all these names is changed to "dp__", since "$" is not a legal identifier character in standard C compilers.

Here is a typical error handler using these variables and functions:

```
%ERROR
   /* Display message upon a syntax error */

   dcl er_action_index fixed bin (15);
   dcl er_action fixed bin (15);
   dcl er_message char (1024) varying;

   er_message = '**** Found '
                || dp$_token_name (dp$_tkntyp)
                || ' instead of:  ';
   /* Find expected symbols */
   do er_action_index = 1 to dp$_number_of_actions (dp$_current_state);
      er_action = dp$_nth_action (dp$_current_state, er_action_index);
      er_message = er_message
                || dp$_token_name (er_action)
                || ' ';
   end;
   call display_message (er_message);
%END_ERROR;
```

This error handler works well in conjunction with the SYNTAX_ERROR mechanism described next.

5.3 The SYNTAX_ERROR Symbol

DEREMER parsers feature a controlled error detection and recovery method, using a special, reserved token name SYNTAX_ERROR.

Note that the identifier SYNTAX_ERROR must not be used in any sense other than the one used here. SYNTAX_ERROR can appear in upper or lower case.

Traditionally, parsers recover from syntax errors by scanning the input stream for a synchronizing token, and continuing from there. For example, a semicolon makes a good synchronizing token for PL/I-like languages because it indicates the end of a statement, which is a good place to recover to. DEREMER provides the special token SYNTAX_ERROR to allow you to declare your own synchronizing tokens so recovery can be to points of your own choosing.

We use the 'Calc1' example above to show how SYNTAX_ERROR is used.

We want to recover to an <expression> if there is a syntax error while parsing an <expression>. Just add the rule

```
<expression>  ::=  SYNTAX_ERROR
;
```

This declares that all tokens (terminal symbols) that can legally follow an <expression> (namely the tokens plus_, star_, right_paren_, and end_) are now synchronizing tokens. When a syntax error is detected, the the following things will happen:

1. The %ERROR ... %END_ERROR; section, if any, is executed.

2. Input tokens are examined and discarded until one of the synchronizing tokens is found.

3. The rule containing the SYNTAX_ERROR token is recognized, and any associated %ACTION ... %END_ACTION section is executed. Typical actions might be to display a specific error message, or to set a flag to inhibit further code generation.

4. The parse continues normally with the token following the synchronizing token.

Since the input tokens are discarded until a synchronizing token is found, you must insure that a synchronizing token can always be found, otherwise an infinite loop can occur. Including the rule

```
<start_symbol> ::= SYNTAX_ERROR end_of_stream_token ;
```

(where "<start_symbol>" is your start symbol, and "end_of_stream_token" is your end-of-file or end-of-line token) fulfills this requirement. This rule ensures that there will always be at least one rule pending on the stack which contains the token SYNTAX_ERROR. Thus, in the worst case, recovery from a syntax error will read tokens till the end of the input, and then stop.

SYNTAX_ERROR is used in grammar rules just like a nonterminal symbol representing one or more tokens, other than those specified by other rules. For example, the rule

```
<start>  ::=  a  end_
            |  SYNTAX_ERROR end_
;
```

represents the language consisting of a single 'a' token, where any other case (no token, some token other than 'a') results in recognizing the second alternative.

SYNTAX_ERROR may be used just like any other nonterminal, so the above example could be written

```
<start>  ::=  <indirect>  end_
;
```

```
<indirect>  ::=  a
                 |  SYNTAX_ERROR
;
```

and this would have the same meaning. The important thing to note is that SYNTAX_ERROR eats up (or represents) all otherwise unspecified tokens (terminal symbols) up to the first token which may follow it in the grammar ('end_' in the example above). Multiple uses of SYNTAX_ERROR must obey the same LALR(1) restrictions as any other symbols and, in addition, must be chosen such that the set of tokens indicated by the grammar rules as being legal following the SYNTAX_ERROR symbol are exactly the ones you would like each SYNTAX_ERROR to stop at. When in doubt, try experimenting with small test grammars, running the resulting parsers on test cases or examining the resulting FSA files.

6 Parse Stack Overflow

Stack overflow when running a DEREMER-generated parser is caused by too many shifts prior to a reduce, and is typically the result of specifying right-recursive lists in a grammar, as shown in the following example:

```
<A_list> ::= <A> ;
<A_list> ::= <A> <A_list> ;
```

Such use of right recursion is appropriate for LL(1) grammars but not for the LR(1) grammars DEREMER accepts. Under DEREMER, right recursion causes one additional stack position to be used for each item on the list, whereas left recursion uses only one stack position for the entire list, no matter how long it is.

Right-recursive lists can always be changed to use left recursion, and this will solve the overflow problem:

```
<A_list> ::= <A> ;
<A_list> ::= <A_list> <A> ;
```

The usual reason for using right recursion is that this allows the semantic action code to construct a singly-linked list of structures associated with the symbols in the list in a natural way (the parsing occurs in forward order for left recursion and in backward order, which is most natural for forward-threading, for right recursion). Threading singly-linked lists in forward order using left recursion only requires maintaining a pointer to the previously-parsed structure in the list; this additional slight increase in complexity is well worth doing to eliminate right recursion.

For those rare situations where right recursion or other grammar complexities cannot be eliminated, DEREMER can now produce code in the generated parser to check dynamically for overflow of the state and symbol stack. If an overflow happens, the user-supplied routine "dp$_overflow" will be called; this routine should display an error

message and never return to the parser. The generation of this extra code is enabled by specifying the "%CHECK_STACK;" directive.

7 Practical Hints

7.1 Variable Names

All parser internal variables have names beginning with the characters 'dp$_'. You should avoid using variables beginning with these characters, unless you have a specific reason to interface with internal variables of the parser.

7.2 Semantic Values

As we have seen above, DEREMER associates two different kinds of values with each token returned by the lexer procedure. The first kind, an integer, is used purely for syntactic analysis. This means that it is used to look up the next parser state in the parser tables. The second kind, a pointer, is not used by the parser itself, but is available to semantic actions via the notations '$$' and '$n'. This latter kind of value is called a semantic value.

Prior to executing each semantic action, the default action '$$=$1;' is executed. This action stores the element of the parser symbol stack (semantic value stack) corresponding to the first symbol on the right hand side into an internal variable named dp$_reduce_result. Any user semantic action that assigns a new value to '$$' will actually assign the new value to dp$_reduce_result, overwriting the default value. Finally, after any user semantic action is done, the parse stack is popped (to discard the shifted symbols) and dp$_reduce_result is pushed onto the stack.

Semantic values are represented by pointers because this provides a high degree of generality. A pointer can point to any datatype or structure legal in a PLP, PL1G, PL1, SPL, C, or CC 'based' (or abstract structure) declaration. It can even hold one or two integers in its second and third words, although this usage requires knowledge of the internal format of pointers and is therefore discouraged.

The calculator example, Calc1, showed one way of using semantic values: to allocate and free them as needed. A more efficient mechanism to do this allocation and freeing is shown in the revised example, Calc2, on the next page.

```
/* An interpreter to do arithmetic calculations on single digits */
%PARSER Calc2 (input_string);                    /* CHANGED */

%DECLARATIONS
declare input_string char (*) varying;
     /* Parameter to the parser procedure */
declare input_position fixed bin;
declare free_ptr pointer;                        /* ADDED */
declare 1 value (100) like based_value;          /* ADDED */
declare 1 based_value based,                     /* ADDED */
          2 next_free_value pointer,
          2 integer fixed bin (31);
declare j fixed bin (15);                        /* ADDED */
declare rank builtin;
     /* A function to return the ASCII code for the character */
%Include 'Calc2.ins.pl1';                        /* CHANGED */
%END_DECLARATIONS;

%TOKEN plus_ , star_ , left_paren_ , right_paren_ , digit_, end_;

%RULES;
<input_line>  ::=  <expression>  end_
;
<expression>  ::=  <expression>  plus_  <term>
                       %ACTION                   /* CHANGED */
                       $$ -> based_value.integer =
                          $1 -> based_value.integer +
                          $3 -> based_value.integer;
                       call drop ($3);
                       %END_ACTION
                 |  <term>     /* Default action is '$$ = $1;' */
;
<term>  ::=  <term>  star_  <primary>
                       %ACTION                   /* CHANGED */
                       $$ -> based_value.integer =
                          $1 -> based_value.integer *
                          $3 -> based_value.integer;
                       call drop ($3);
                       %END_ACTION
          |  <primary>
;
<primary>  ::=  digit_
              |  left_paren_  <expression>  right_paren_
                       %ACTION
                       $$ = $2;
                       %END_ACTION
;

%INIT
input_position = 1;
/* ADDED: Initialize list of integer values */
free_ptr = addr (value(1));
do j = 1 to hbound (value, 1) - 1;
   value(j).next_free_value = addr (value(j + 1).next_free_value);
```

```
end;
value(hbound(value,1)).next_free_value = null ();
%END_INIT;

%PROGRAMS
lexer:
   procedure (type, value_ptr);
      dcl type fixed bin (15);
      dcl value_ptr pointer;
      dcl current_char char (1);

      /* Value returned will be undefined by default */
      value_ptr = null ();

      /* Return 'end_' token at end of string */
      if input_position > length (input_string)
      then
      do;
         type = end_;
         return;
      end;

      /* Get next character */
      current_char = substr (input_string, input_position, 1);
      input_position = input_position + 1;

      /* Return proper token for current character */
      select (current_char);
         when ('+') type = plus_;
         when ('*') type = star_;
         when ('(') type = left_paren_;
         when (')') type = right_paren_;
         when ('0','1','2','3','4','5','6','7','8','9')
         do;
            type = digit_;
            call make (value_ptr);              /* CHANGED */
            value_ptr ->                        /* CHANGED */
               based_value.integer = rank (current_char)
                                   - rank ('0');
         end;
         otherwise type = 0;       /* To detect illegal characters */
      end;      /* end of select statement */
   end lexer;  /* end of PROGRAMS section */

make:                                           /* ADDED */
   procedure (value_ptr);
      dcl value_ptr pointer;
      value_ptr = free_ptr;
      free_ptr = free_ptr -> based_value.next_free_value;
   end make;

drop:                                           /* ADDED */
   procedure (value_ptr);
      dcl value_ptr pointer;
```

```
      if value_ptr ^= null ()
      then
      do;
         value_ptr -> based_value.next_free_value = free_ptr;
         free_ptr = value_ptr;
      end;
   end drop;
%END_PROGRAMS;

%END_PARSER;
```

The changes and additions shown above maintain an array of linked structures, each containing one semantic value (in this case, an integer). The allocation procedure 'make' and the deallocation procedure 'drop' must be called explicitly when needed, just as in 'Calc1' above. Failing to deallocate (free or drop) a semantic value may result in running out of free values at run time. This error may be very difficult to debug, since all grammar rules and actions will have to be examined.

Another way to use an array of linked structures is to use them freely in semantic actions without doing any deallocating, and to initialize (link) them at each call to the parser. If this method is chosen, there is no reason actually to use link pointers; it is simpler to allocate the data items sequentially from an array, and initialize by beginning the allocation from the start of the array.

Safer (and more efficient) ways of manipulating semantic values without requiring explicit allocation and deallocation are possible, but DEREMER does not support these very well. One possibility is to change the datatype of a semantic value from pointer to a structure or other declaration appropriate to your application. This can be done by writing a series of ED commands to edit the parser appropriately. Then references to semantic values can look like '$2.integer' (in the case of a structure) or even just '$2' (but note using structures in this way is limited by the fact that you cannot assign structures in all the supported languages; also, passing them as parameters is complicated by having to write separate entry statements lacking the member names).

Another possibility is to ignore the value pointers entirely (it may, however, be necessary to set them to null) and define one or more arrays of items of the desired datatype in parallel with the parser array (dp$_symbol_stack). The new parallel array(s) can be indexed by the appropriate offset from the parser stack pointer dp$_symbol_stack_ptr, which is a fixed bin (15). Note, however, that the constructs '$$' and '$n' will be useless, since these translate into the pointer references 'dp$_reduce_result' and 'dp$_symbol_stack(dp$_symbol_stack_ptr - x)' respectively, where x = length_of_right_hand_side - n. Thus, referencing semantic values in a parallel stack requires a notation such as 'value (dp$_symbol_stack_ptr - x)' instead of '$n'.

7.3 The Parser

The parser is constructed as a function of as many arguments as you indicate in the %PARSER directive. It returns a semantic value (pointer) as its function value. The parser may be called as a main program (in which case the returned pointer is ignored by Primos) or may be called by a function reference from your application program. An example of the latter usage is

```
program_tree = parse ();
```

for a parser declared by the directive '%PARSER parse;'.

7.4 The Lexer

The lexer is called by the parser every time it needs the next lexical item (token) appearing in the application's input. Aside from writing the lexer, you must either include it as an internal procedure in the %PROGRAMS section of the input file, declare it as an external entry, or %Include it in the %PROGRAMS section. In any case, the lexer takes two arguments. The first is of type fixed bin(15), and is used to return (that is, the lexer assigns a value to this argument) the integer type code of a token. The second argument is of type pointer, and is used to return the semantic (user) value of a token. You are strongly urged to %Include the name.INS.PL1 file in the lexer or in the %DECLARATIONS section of the input file so that the lexer refers to the symbolic names of the tokens, rather than actual numbers.

When the language defined by your grammar is not clearly terminated (which causes a warning issued by DEREMER), the parser may need to read an extra token beyond those specified by the grammar. This is the "right pad" symbol which signals that the final reduction(s) should take place. In these cases, where there is no explicit end-of-input token, the lexer should return 0 or a negative number as the token type which indicates end of input. To avoid this situation, use an explicit end-of-input token, such as 'end_' in the 'Calc1' example.

7.5 Use of Precedence and Associativity

These are powerful mechanisms, described in Advanced Features below, which should be used sparingly because they can hide deeper problems with a grammar. It is a good idea to begin by declaring all tokens using %TOKEN, and to use the precedence mechanisms only to resolve specific shift-reduce conflicts which come up. Don't try to anticipate a shift-reduce conflict until you've had some practice.

8 The DEREMER Algorithm

The parser generation method used is based upon the Ph.D thesis of Franklin L. DeRemer [DeRemer 69]. This algorithm handles LALR(1) grammars (technically, "Not-Quite-LALR(1)" grammars), which are somewhat restricted subsets of LR(1) grammars, which in turn are very restricted subsets of Context Free Grammars.

The recognizing of grammar rules is done in a strict bottom-up fashion. For example, in the rule

```
<term> ::= <term> star_ <primary>
```

the processing occurs in the following order: (1) All recognitions connected with the non-terminal symbol <term> on the right hand side; (2) the shifting of the token 'star_'; (3) All recognitions connected with the non-terminal symbol <primary>; (4) The recognition of the rule itself.

The original DEREMER was written in FORTRAN and generated an open coded FORTRAN parser. The current version is basically a conversion to PLP having some additional technique inspired by [Aho&Ullman 77], an efficient table-packing scheme, error recovery as described in [Poonen 77], and user and semantic interfacing largely modelled after YACC [Johnson 74].

Some thought has been given to the question of generating lexers automatically. Given regular expressions for each of the token types, it should be possible to do this using the type of technique described by Larry Stabile [PE-T-444, 488].

9 Debugging a Parser: How It Works

You have written a grammar and incorporated it, with semantic actions, into a DEREMER input file. You have run the input file through DEREMER and produced a parser. The next step is to correct any error messages reported by DEREMER. Some help with this is provided under DEREMER Error Messages below. Finally, you are able to obtain a parser, it compiles and links without error, but when it is run it produces the wrong results in that

(1) a syntactically correct input string produces a syntax error,

(2) an incorrect input string produces no error, or

(3) a correct input string produces the wrong semantic result.

Case (3) is usually easy to diagnose when the semantic actions fit naturally with the grammar, and when the parser is producing the correct analysis of an input string. This discussion concentrates on cases (1) and (2) where it is clear that the parser is not analyzing the input string correctly.

First of all, make sure that the grammar is processed without any error messages from DEREMER.

Next, re-run DEREMER with the '-debug' command option to produce a file named 'name.DEBUG.lang'. Compile and link this parser, along with your lexer. The resulting program can be run with various test cases in order to see exactly what is happening wrong. Messages will be displayed showing every action taken by the parser. Note that such DEBUG parsers use the system subroutine IOA$ (see PE-T-364 for a description of IOA$). Do not confuse the '-debug' option for DEREMER with the '-debug' option of PLP, PL1G, PL1, SPL, C, or CC.

The material below explains how to understand the operation of your parser in detail, should you encounter a problem that cannot be solved using the '-debug' option.

Important Note: It is rarely necessary to examine the parser states as described in the remainder of this section. This material is included for the sake of completeness and may be ignored until needed. Examining the grammar and running a parser created using the '-debug' option are the only debugging methods most users will ever need.

To get a detailed listing of the parser states, run DEREMER with the '-fsa' option. Get listings of the 'name.lang.DEREMER' file and of the resulting 'name.FSA' and 'name.INS.PL1' files.

You can simulate the operation of the parser by tracing through the FSA file, as described below.

Alternatively, you can actually run the parser in steps, tracing through its operation using the Primos debugger (DBG). To do this, compile the parser using the '-debug' option of PLP, PL1G, PL1, SPL, CC, or CI. Next, link your application or test using SEG or BIND. Invoke DBG on your runfile, and set a breakpoint at the label 'dp$_loop' within the parser. This is the point within the parser's interpretation loop where it decides which action to perform. The variables 'dp$_current_state' and 'dp$_tkntyp' are the current state number and the token type code respectively. Also, there is a Boolean variable called 'dp$_lookahead_valid' whose usage is explained below. By examining the values of these variables each time around the interpretation loop, you should be able to follow the action of the parser from state to state as it parses a given input. You need the name.INS.PL1 file or the dp$_token_name function to translate token type codes back to token names.

To understand what is happening inside the parser, you need a more detailed explanation of the parser's operation.

Conceptually, the DEREMER-generated parser consists of a set of states, with parsing actions defined for each state which are dependent on the current lookahead symbol.

The parser maintains 2 stacks, a state stack which stores parser state numbers, and a symbol stack which stores the semantic values (pointers)

associated with terminal and nonterminal symbols. When the parser is initialized, both stacks are empty and the current parser state is always state 1.

The bit variable 'dp$_lookahead_valid' is used to remember whether the current token has already been shifted or not. It is initialized to be '0'b.

When the parsing actions defined for the current state consist of one possible reduction and no shift actions, the parser doesn't need to know the lookahead symbol. Otherwise, (if the current state has any shift actions or more than one possible reduction) the parser must see the lookahead symbol in order to decide which action to take. In this situation, if 'dp$_lookahead_valid' is false, then the parser calls the lexer, thus obtaining the next token, and sets 'dp$_lookahead_valid' to be '1'b.

Sample Output Files below contains the CALC1.FSA file describing the example grammar. You may wish to refer to it as you read on.

You can see that the '-fsa' output shows the set of parser states, with the actions listed for each state. Each shift action is denoted in the name.FSA file by a line of the form

```
ON token-name GO TO state-number.
```

When the parser executes this action,

(1) the current state number is pushed onto the state stack,

(2) the token value is pushed onto the symbol stack,

(3) 'dp$_lookahead_valid' is set to '0'b,

and the next state is the "state-number" mentioned in the shift action.

Each reduce action is denoted by a set of three lines of the form

```
ON token1 token2 ...
grammar_rule
(f1,t1) (f2,t2) ...
```

When this reduce action is occurs, the parser performs the following operations:

(1) The semantic action associated with "grammar_rule" is performed.

(2) The top n-1 state numbers on the state stack and the top n values on the symbol stack are popped, where n is the number of symbols on the right hand side of "grammar_rule".

(3) The result of the semantic action generated by step (1) is pushed onto the symbol stack.

(4) The number on the top of the state stack is compared with the first elements of the ordered pairs (f1,t1) (f2,t2) ... When a match is found, the next state of the parser is the second element of the pair whose first element matches.

Note that a reduce action does not invalidate the lookahead symbol.

At this point an example would be helpful. Consider the input string

```
3 + 1 * 5
```

having the token representation

```
digit_ plus_ digit_ star_ digit_ end_
```

The successive actions of the parser can be described by the sequence of snapshots shown on the next page:

Stacks	Current State	Lookahead Symbol
(empty) <-- State Stack (empty) <-- Symbol Stack	1	digit_
1 digit_	7	(invalid)
1 <primary>	6	(invalid)
1 <term>	10	plus_
1 <expression>	13	plus_
1 13 <expression> plus_	3	digit_
1 13 3 <expression> plus_ digit_	7	(invalid)
1 13 3 <expression> plus_ <primary>	6	(invalid)
1 13 3 <expression> plus_ <term>	12	star_
1 13 3 12 <expression> plus_ <term> star_	4	digit_
1 13 3 12 4 <expression> plus_ <term> star_ digit_	7	(invalid)
1 13 3 12 4 <expression> plus_ <term> star_ <primary>	5	(invalid)
1 13 3 <expression> plus_ <term>	12	end_
1 <expression> end_	13	(invalid)
1 <input_line>	2 (accept)	(invalid)

You are encouraged to construct other short examples of input strings and hand-simulate the operation of the parser using the FSA file, as done here.

Two additional notations may occur within an FSA file. The symbol ';' is used to represent the "right pad" character which implicitly terminates every input string. Parsers for languages which are clearly

terminated never need to read this token. Parsers for languages which are not clearly terminated may occasionally read this "extra" token, but they never shift it. It serves to trigger the final reduction(s) of the parser.

The symbol '<' appearing as a nonterminal symbol is used as a DEREMER generated start symbol in cases where the user's start symbol is used on the right hand side of some production. If the start symbol is recognized, there are instances where the symbol is used as a constituent and others where it indicates that the parser should return. DEREMER removes this confusion by adding a production at the head of the grammar (a process called "augmentation") whose form is

```
< ::= <user_start_symbol> ;
```

This operation is performed only if the user's start symbol appears on the right hand side of a rule. The use of an unmatched left angle bracket for this purpose ensures that it can never conflict with a user-defined nonterminal.

10 Sample Output Files

On the following pages are the files CALC1.FSA, CALC1.INS.PL1, CALC1.GRAMMAR, and CALC1.SPL resulting from processing the input file CALC1.SPL.DEREMER listed above.

10.1 CALC1.FSA

```
state 1:
    on digit_ go to 7.
    on left_paren_ go to 8.

state 2: (final state)
    on ;
     1. <input_line>  ::=  <expression>  end_
        .

state 3:
    on digit_ go to 7.
    on left_paren_ go to 8.

state 4:
    on digit_ go to 7.
    on left_paren_ go to 8.

state 5:
    on star_ right_paren_ plus_ end_
     4. <term>  ::=  <term>  star_  <primary>
        (  1, 10) (  8, 10) (  3, 12) .

state 6:
    on star_ right_paren_ plus_ end_
     5. <term>  ::=  <primary>
        (  1, 10) (  3, 12) (  8, 10) .

state 7:
    on star_ right_paren_ plus_ end_
     6. <primary>  ::=  digit_
        (  1,  6) (  3,  6) (  4,  5) (  8,  6) .

state 8:
    on digit_ go to 7.
    on left_paren_ go to 8.

state 9:
    on star_ right_paren_ plus_ end_
     7. <primary>  ::=  left_paren_  <expression>  right_paren_
        (  1,  6) (  3,  6) (  4,  5) (  8,  6) .

state 10:
    on star_ go to 4.
    on right_paren_ plus_ end_
     3. <expression>  ::=  <term>
        (  1, 13) (  8, 11) .

state 11:
    on right_paren_ go to 9.
    on plus_ go to 3.

state 12:
```

```
    on star_ go to 4.
    on end_ right_paren_ plus_
     2. <expression>  ::=  <expression>  plus_  <term>
        (  8, 11) (  1, 13) .

state 13:
    on plus_ go to 3.
    on end_ go to 2.
```

10.2 CALC1.INS.PL1

```
/* associativities: token(1), left(2), right(3), nonassociating(4). */
/* prec(    1), assoc(    1). */   %replace plus_ by 1;
/* prec(    1), assoc(    1). */   %replace star_ by 2;
/* prec(    1), assoc(    1). */   %replace left_paren_ by 3;
/* prec(    1), assoc(    1). */   %replace right_paren_ by 4;
/* prec(    1), assoc(    1). */   %replace digit_ by 5;
/* prec(    1), assoc(    1). */   %replace end_ by 6;
```

10.3 CALC1.GRAMMAR

```
/* associativities: token(1), left(2), right(3), nonassociating(4). */
/* prec(    1), assoc(    1). */   1. <input_line>  ::=  <expression>
end_
/* prec(    1), assoc(    1). */   2. <expression>  ::=  <expression>
plus_  <term>
/* prec(    0), assoc(    1). */   3. <expression>  ::=  <term>
/* prec(    1), assoc(    1). */   4. <term>  ::=  <term>  star_  <prim
ary>
/* prec(    0), assoc(    1). */   5. <term>  ::=  <primary>
/* prec(    1), assoc(    1). */   6. <primary>  ::=  digit_
/* prec(    1), assoc(    1). */   7. <primary>  ::=  left_paren_  <exp
ression>  right_paren_
```

10.4 CALC1.SPL

```
Calc1: /* deremer rev 19.0c */
  procedure (input_string) returns(ptr);
    dcl dp$_state_stack(200) bin;
    dcl dp$_state_stack_ptr  bin;
    dcl dp$_symbol_stack(200) ptr;
    dcl dp$_symbol_stack_ptr  bin;
    dcl dp$_reduce_result    ptr;
    dcl dp$_current_state    bin;
    dcl dp$_action           bin;
    dcl (dp$_i, dp$_j, dp$_k, dp$_l, dp$_m) bin;
    %replace dp$_base_ by 1;
    %replace dp$_alias_ by 2;
    %replace dp$_otherwise_ by 3;
    %replace dp$_rhslen_ by 4;
    %replace dp$_next_ by 1;
    %replace dp$_check_ by 2;
    dcl dp$_top_state bin;
    dcl dp$_tkntyp bin;
    dcl dp$_tknptr ptr;
    dcl dp$_lookahead_valid bit(1);

dcl dp$_action_list_info(13, 3) bin static init(
/* state     1 */    -2,     1,     0,
/* state     2 */     0,    -1,    -1,
/* state     3 */    -2,     1,     0,
/* state     4 */    -2,     1,     0,
/* state     5 */     0,    -1,    -4,
/* state     6 */     0,    -1,    -5,
/* state     7 */     0,    -1,    -6,
/* state     8 */    -2,     1,     0,
/* state     9 */     0,    -1,    -7,
/* state    10 */     0,    10,    -3,
/* state    11 */     3,    11,     0,
/* state    12 */     0,    10,    -2,
/* state    13 */     4,    13,     0);

%replace dp$_action_list_length by 10;

dcl dp$_action_list(10, 2) bin static init(
/* row     1 */      8,     1,
/* row     2 */      4,    10,
/* row     3 */      7,     1,
/* row     4 */      3,    11,
/* row     5 */      3,    13,
/* row     6 */  (1)0,  (1)0,
/* row     7 */      9,    11,
/* row     9 */  (2)0,  (2)0,
/* row    10 */      2,    13);

dcl dp$_reduce_list_info(7, 4) bin static init(
/* production     1 */     0,    -1,     0,     2,
/* production     2 */    -7,     2,    13,     3,
```

```
/* production     3 */    -7,     2,    13,     1,
/* production     4 */    -1,     4,    10,     3,
/* production     5 */    -1,     4,    10,     1,
/* production     6 */    -1,     6,     6,     1,
/* production     7 */    -1,     6,     6,     3);

%replace dp$_reduce_list_length by 3;

dcl dp$_reduce_list(3, 2) bin static init(
/* row     1 */    11,     2,
/* row     2 */    12,     4,
/* row     3 */     5,     6);

declare input_string char (*) varying;
     /* Parameter to the parser procedure */
declare input_position fixed bin;
declare based_integer fixed bin (31) based;
declare rank builtin;
     /* A function to return the ASCII code for the character */
%Include 'Calc1.ins.pl1';

input_position = 1;
    dp$_state_stack_ptr  = 0;
    dp$_symbol_stack_ptr = 0;
    dp$_lookahead_valid  = '0'b;
    dp$_current_state    = 1;
    do while('1'b);
      if ^dp$_lookahead_valid & /* don't already have next symbol. */
         dp$_action_list_info(dp$_current_state, dp$_alias_) ^= -1 /* f
lag that says to read a symbol. */
        then /* read a symbol. */
          do;
            call lexer(dp$_tkntyp, dp$_tknptr);
            dp$_lookahead_valid = '1'b;
          end;
      dp$_state_stack_ptr = dp$_state_stack_ptr + 1; /* push current st
ate. */
      dp$_state_stack(dp$_state_stack_ptr) = dp$_current_state;
      dp$_action = dp$_access_action_list(dp$_current_state, dp$_tkntyp
);
dp$_loop:
      if dp$_action > 0
        then /* shift. */
          do;
            dp$_symbol_stack_ptr = dp$_symbol_stack_ptr + 1; /* push cu
rrent symbol. */
            dp$_symbol_stack(dp$_symbol_stack_ptr) = dp$_tknptr;
            dp$_lookahead_valid = '0'b;
            dp$_current_state = dp$_action;
          end;
        else
      if dp$_action < 0
        then /* reduce. */
          do;
```

```
            if dp$_reduce_list_info(- dp$_action, dp$_rhslen_) > 0 /* h
ave a rhs. */
              then /* $$ = $1. (default) */
                dp$_reduce_result =
                  dp$_symbol_stack(dp$_symbol_stack_ptr - dp$_reduce_li
st_info(- dp$_action, dp$_rhslen_) + 1);
              else /* $$ = null(). */
                dp$_reduce_result = null();
            select(- dp$_action); /* a "when (rdn£) do; ... end;" for e
ach reduce. */
              when (1)
                do;
                end;
              when (2)
                do;

                  dp$_reduce_result -> based_integer =
dp$_symbol_stack(dp$_symbol_stack_ptr - 2) -> based_integer +

dp$_symbol_stack(dp$_symbol_stack_ptr - 0) -> based_integer;
                  free
dp$_symbol_stack(dp$_symbol_stack_ptr - 0) -> based_integer;

                end;
              when (3)
                do;
                end;
              when (4)
                do;

                  dp$_reduce_result -> based_integer =
dp$_symbol_stack(dp$_symbol_stack_ptr - 2) -> based_integer *

dp$_symbol_stack(dp$_symbol_stack_ptr - 0) -> based_integer;
                  free
dp$_symbol_stack(dp$_symbol_stack_ptr - 0) -> based_integer;

                end;
              when (5)
                do;
                end;
              when (6)
                do;
                end;
              when (7)
                do;

                  dp$_reduce_result =
dp$_symbol_stack(dp$_symbol_stack_ptr - 1);

                end;
              otherwise;
            end;
            dp$_symbol_stack_ptr = dp$_symbol_stack_ptr - dp$_reduce_li
```

```
st_info(- dp$_action, dp$_rhslen_) + 1;
            dp$_state_stack_ptr = dp$_state_stack_ptr - dp$_reduce_list
_info(- dp$_action, dp$_rhslen_);
            if dp$_reduce_list_info(- dp$_action, dp$_otherwise_) = 0
              then /* accept. */
                do;
                  return(dp$_reduce_result);
                end;
            dp$_symbol_stack(dp$_symbol_stack_ptr) = dp$_reduce_result;
            dp$_top_state = dp$_state_stack(dp$_state_stack_ptr);
            dp$_current_state = dp$_access_reduce_list(- dp$_action, dp
$_top_state);
          end;
        else
      if dp$_action = 0 /* only option left! */
        then /* error. */
          do;
            return(null());
          end;
dp$_next:
    end;
dp$_number_of_actions:
  procedure(state_number) returns(bin);
    dcl state_number bin;
    dcl (i, j, k, l) bin;
    k = 0; /* count of actions. */
    i = dp$_action_list_info(state_number, dp$_alias_);
    do j = 1 to dp$_action_list_length;
      l = j - dp$_action_list_info(state_number, dp$_base_);
      if dp$_action_list(j, dp$_check_) = i
        then
          if l ^= -1
            then
              k = k + 1;
    end;
    return(k);
  end /* of dp$_number_of_actions */;
dp$_nth_action:
  procedure(state_number, n) returns(bin);
    dcl (state_number, n) bin;
    dcl (i, j, k, l) bin;
    k = 1;
    i = dp$_action_list_info(state_number, dp$_alias_);
    do j = 1 to dp$_action_list_length;
      l = j - dp$_action_list_info(state_number, dp$_base_);
      if dp$_action_list(j, dp$_check_) = i
        then
          if l ^= -1
            then
              if k = n
                then
                  return(l);
                else
                  k = k + 1;
```

```
    end;
    return(0); /* should never get here. */
  end /* of dp$_nth_action */;
dp$_access_action_list:
  procedure(row, offset) returns(bin);
    dcl (row, offset, i) bin;
    i = dp$_action_list_info(row, dp$_base_) + offset; /* base(s) + a *
/
    if i >= 1 & i <= dp$_action_list_length
      then /* inside bounds, consult table. */
        if dp$_action_list(i, dp$_check_) =
           dp$_action_list_info(row, dp$_alias_)
          then /* match! check(base(s) + a) = alias(s) */
            return(dp$_action_list(i, dp$_next_)); /* next(base(s) + a)
 */
    /* get here if outside bounds of table, or no match. */
    return(dp$_action_list_info(row, dp$_otherwise_)); /* otherwise(s)
*/
  end /* of dp$_access_action_list */;
dp$_access_reduce_list:
  procedure(row, offset) returns(bin);
    dcl (row, offset, i) bin;
    i = dp$_reduce_list_info(row, dp$_base_) + offset; /* base(s) + a *
/
    if i >= 1 & i <= dp$_reduce_list_length
      then /* inside bounds, consult table. */
        if dp$_reduce_list(i, dp$_check_) =
           dp$_reduce_list_info(row, dp$_alias_)
          then /* match! check(base(s) + a) = alias(s) */
            return(dp$_reduce_list(i, dp$_next_)); /* next(base(s) + a)
 */
    /* get here if outside bounds of table, or no match. */
    return(dp$_reduce_list_info(row, dp$_otherwise_)); /* otherwise(s)
*/
  end /* of dp$_access_reduce_list */;
dp$_token_name:
  procedure(n) returns(char(32) varying);
    dcl n bin;
    go to name(n);
name(1): return('plus_');
name(2): return('star_');
name(3): return('left_paren_');
name(4): return('right_paren_');
name(5): return('digit_');
name(6): return('end_');
  end /* of dp$_token_name */;

lexer:
   procedure (type, value_ptr);
      dcl type fixed bin (15);
      dcl value_ptr pointer;
      dcl current_char char (1);
      /* Value returned will be undefined by default */
      value_ptr = null ();
```

```
      /* Return 'end_' token at end of string */
      if input_position > length (input_string)
      then
      do;
         type = end_;
         return;
      end;
      /* Get next character */
      current_char = substr (input_string, input_position, 1);
      input_position = input_position + 1;
      /* Return proper token for current character */
      select (current_char);
         when ('+') type = plus_;
         when ('*') type = star_;
         when ('(') type = left_paren_;
         when (')') type = right_paren_;
         when ('0','1','2','3','4','5','6','7','8','9')
         do;
            type = digit_;
            allocate based_integer set (value_ptr);
            value_ptr -> based_integer = rank (current_char)
                                       - rank ('0');
         end;
         otherwise type = 0;      /* To detect illegal characters */
      end;     /* end of select statement */
   end lexer;  /* end of PROGRAMS section */
end /* of Calc1 */;
```

11 Advanced Features

11.1 Ambiguity Resolution using Precedence and Associativity

DEREMER generates parsers for the subset of context free grammars called LALR(1), which stands (roughly) for Look Ahead Left-to-Right using 1 look-ahead symbol. This means that the parsing of input strings takes place left to right, with no backup or indeterminacy, with a lookahead of exactly 1 symbol. Recall that the parser alternates between shifting symbols onto its stack and reducing stack symbols to form nonterminals. At each point in the process, a decision is made whether to shift or reduce. This decision is based upon the current state of the parser and the "next", or lookahead, symbol. For certain grammars, it may not be possible for DEREMER to make a clearcut decision for all combinations of current state and lookahead symbol. This happens because a grammar is inherently ambiguous, or because its parsing requires a lookahead of 2 or more symbols, or for an assortment of other possible reasons. When the choice is between a shift and a reduce, DEREMER reports a shift-reduce conflict. In certain cases, a parser may have a choice of more than one possible reduction. This is called a reduce-reduce conflict.

For each type of conflict, DEREMER has a default resolution. For reduce-reduce conflicts, the default is to reduce by the production which appears first in the grammar. Experience has shown that this type of situation almost always results from ambiguity in the grammar and should be eliminated if possible. Otherwise, it may produce surprising or incorrect results.

For shift-reduce conflicts, in the absence of specific precedence and associativity rules (described below), the default is to shift. This favors longer rules over shorter ones.

The statement that a grammar is ambiguous indicates that there are legal input strings which can be parsed in more than one way. The existence of these alternatives shows up in the parser as parsing conflicts, which represent the points where a choice between alternatives must be made.

Ambiguity in a parser may result from improper arrangement of the rules, or may be unavoidable in that more than one look-ahead symbol is required to recognize the language. In the latter case, doing the look-ahead in the lexer can provide a solution.

When rearrangement of the grammar is indicated, the number of rules added to resolve a shift-reduce or reduce-reduce conflict can be limited by making use of the DEREMER precedence and associativity features.

These terms are explained (and these features are used) most naturally in the context of infix algebraic expressions. In an expression of the form

```
digit1 op1 digit2 op2 digit3
```

the statement that op1 has higher precedence than op2 indicates that the operand digit2 should "bind closer" to op1 than to op2, and therefore that op1 is performed first, with its result used as the left operand of op2. If instead op2 has higher precedence than op1, then op2 is performed first, and its result becomes the right operand of op1.

If op1 and op2 have equal precedence (most notably if op1 and op2 are instances of the same operator), then the choice depends upon the associativity, i.e. whether the operators group to the left, right, or neither side.

Calc1, an unambiguous grammar, enforces a particular order of evaluation by using the nonterminals <term> and <primary>, together with the "single" or "chain" rules

```
<expression>  ::=  <term>      and      <term>  ::=  <primary>
```

to establish a particular constituent relationship between groups of input symbols. If these extra nonterminals and single productions are removed, the result is the set of grammar rules we shall call Calc3:

```
/* Calc3 */
<input_line>  ::=  <expression>  end_
;
<expression>  ::=  <expression>  plus_  <expression>
                 |  <expression>  star_  <expression>
                 |  left_paren_  <expression>  right_paren_
                 |  digit_
;
```

This grammar is more concise than Calc1, and in certain ways it is more natural.

The difficulty with the concise grammar is that it is ambiguous. In particular, there are two possible interpretations for the input string

```
3 + 4 + 5
```

corresponding to

```
(3 + 4) + 5
```

and

```
3 + (4 + 5)
```

The same applies to the input strings

```
3 * 4 * 5
```

```
3 + 4 * 5

3 * 4 + 5
```

In each case, the conflict occurs between the second digit and the second operator. The parser has the choice of reducing two digits and an operator to an <expression>, thus performing the leftmost operation first, or it may shift the second operator, which effectively causes the rightmost operation to be performed first.

One deals with these conflicts by specifying precedences and associativities for 'plus_' and 'star_'. The necessary mechanisms are

(1) an input format for declaring the precedences and associativities to DEREMER, and

(2) an algorithm by which DEREMER translates the precedences and associativities into a specific choice of shift or reduce.

The input format works as follows: Each token which appears in a %LEFT, %RIGHT, or %NONASSOC is assigned the associativity "left", "right", or "nonassociative". Each successive %LEFT, %RIGHT, or %NONASSOC statement causes the tokens appearing within that statement to be assigned a precedence higher than all previously declared tokens. Tokens appearing within %TOKEN statements, and those which are not declared at all, receive no precedence or associativity.

Each production of the grammar normally takes the same precedence and associativity as the precedence and associativity of its rightmost token (terminal symbol). This may be overridden by attaching the keyword %PREC, followed a token name, after the grammar rule (before or after the %ACTION clause). When a %PREC is supplied, the production takes the precedence and associativity of the token which follows %PREC.

Here is an example of a grammar using most of the features just discussed:

```
/* Calc4: Using %LEFT, %RIGHT, and %PREC */
%TOKEN left_paren_ , right_paren_ , digit_;
%LEFT minus_;
%RIGHT expt_;    /* Exponentiation has higher precedence than minus
                    and is right-associative */
%LEFT uminus_;  /* Unary minus has higher precedence than minus */
%RULES;
<expression>  ::=  <expression>  minus_  <expression>
                |  <expression>  expt_  <expression>
                |  minus_  <expression>
                   %PREC uminus_  /* Use precedence of uminus_ */
                |  digit_
                |  left_paren_  <expression>  right_paren_
;
```

Note here that 'uminus_' is never returned by the lexer. It exists

only to indicate the precedence of the unary minus grammar rule.

When a shift-reduce conflict occurs, the precedences of the conflicting symbol and production are checked. If either lacks a precedence (because of a declaration by %TOKEN), the default action is taken: the conflict is reported, and then is resolved in favor of shift. If both symbol and production have precedences, then the conflict is resolved, without user notification, according to the following table:

Shift-Reduce Conflict Resolution

Associativity:	left	right	nonassoc
prec(production) < prec(symbol)	<------	shift	------->
prec(production) > prec(symbol)	<------	reduce	------->
prec(production) = prec(symbol)	reduce	shift	syntax error

These rules cause resolution of shift-reduce conflicts such that the resulting evaluation order conforms with the precedences and associativities. For instance, if the token plus_ is defined as left associative, then the input string

```
3 + 4 + 5
```

is taken as equivalent to

```
(3 + 4) + 5
```

with the other interpretation enforced by declaring plus_ as right associative. A similar effect occurs for star_. If star_ is given a higher precedence than plus_ (the standard algebraic usage), then the strings

```
3 + 4 * 5
```

and

```
3 * 4 + 5
```

both have their multiplications performed first. If the precedence order is reversed, the evaluation order is also reversed. If plus_ and star_ are defined with the same precedence, then the operations will group to the left or right depending on the associativity. The "nonassoc" category is useful for operators such as comparisons, for which (in the FORTRAN notation) expressions like

```
3 .LT. 4 .LT. 5
```

are illegal.

11.2 Multiple Parsers

Sometimes, although rarely, it may be convenient to break apart one large grammar into several smaller ones, each with its own DEREMER input file. For example, a language may have one parser for a <statement> and another for an <expression>, where the former calls the latter when necessary. Since the parsers may interact in strange ways, a special DEREMER directive, %SYSTEM, has been added to support this case.

The %SYSTEM directive notifies DEREMER whether the current input file is a complete parsing specification or is a part of a larger system, and if a part, whether it is a main (controlling) or subsidiary part. Here are the options:

%SYSTEM INTERNAL; : This is a complete parsing specification (default).

%SYSTEM EXTERNAL; : This is a main, controlling part.

%SYSTEM INHIBIT; : This is a subsidiary part.

This mechanism controls the declarations of some variables (that is, whether they are local or global), and the contents of the name.EXTERNALS and name.INS.PL1 files.

For example, suppose you have the DEREMER input files shown on the next page.

STMT.SPL.DEREMER

```
%PARSER stmt;
%SYSTEM EXTERNAL;
%DECLARATIONS
%Include 'stmt.ins.pl1';
declare expr entry returns (ptr);
%END_DECLARATIONS;
%Include 'stmt.token.directive';
%RULES;
<stmt>  ::=  ...
;
   ...
<expr>  ::=  /* Empty string, since the tokens are read in the
                expr parser */
          %ACTION
          $$ = expr ();  /* Call to subsidiary part */
          %END_ACTION
;
%END_PARSER;
```

EXPR.SPL.DEREMER

```
%PARSER expr;
%SYSTEM INHIBIT;
%DECLARATIONS
%Include 'stmt.ins.pl1';
declare expr entry returns (ptr);
%END_DECLARATIONS;
%Include 'stmt.token.directive';
%RULES;
<expr>  ::=  ...
;
   ...
%END_PARSER;
```

STMT.SPL.DEREMER generates the files STMT.SPL, STMT.INS.PL1, and STMT.EXTERNALS; EXPR.SPL.DEREMER generates only EXPR.SPL.

Since both parsers operate on the same input, they both %Include the file STMT.TOKEN.DIRECTIVE so that they share the same token names. The generated parsers both %Include the file STMT.INS.PL1 for the same reason.

When you compile and link all the files, be sure to remember STMT.EXTERNALS, which contains the debugging function dp$_token_name.

Note that INTERNAL, EXTERNAL, and INHIBIT are reserved identifiers, so that it is illegal to declare them in a %TOKEN directive or use them in any other sense than the one described here. Naturally, these options

can be given in upper or lower case.

11.3 Calling the Recovery Procedure

The internal mechanism for recovering from syntax errors is encapsulated into a procedure named 'dp$_recover'. This procedure may be called from inside %ACTION sections.

This procedure will not normally return - it will do a non-local goto to the label 'dp$_next', in the main parser loop (this is where all semantic actions go when they are done). The exception is if the entire rest of the input is swallowed without having recovered. In this case, dp$_recover will return.

Note - This procedure, along with other mechanisms to support the SYNTAX_ERROR feature, is only included in the parser if you have used SYNTAX_ERROR at least once in your rules.

11.4 Parser Table Format

The parser listing under CALC1.SPL above shows what the parser tables look like. There are four tables, packed in order to make full use of the space. Since they are packed, they are accessed almost exclusively by means of specialized unpacking procedures. The parser tables have the following formats:

1. dp$_action_list_info (state, x): Gives information concerning the list of actions for each state. "X" may be 'dp$_base_', 'dp$_alias_', or 'dp$_otherwise_' and is used by the table packing algorithm.

2. dp$_action_list (x, y): Contains a list of actions for each state. "Y" may be 'dp$_next_' or 'dp$_check_'. Both "x" and "y" are used by the table packing algorithm.

3. dp$_reduce_list_info (production, x): Gives the length of the right hand side of each rule when "x" is 'dp$_rhslen_', and information concerning the list of states used when performing reductions otherwise, in which case "x" is as described for dp$_action_list_info above.

4. dp$_reduce_list (x, y): Contains a list of states used when performing reductions. "X" and "y" are as described for dp$_action_list above.

12 DEREMER Error Messages

The kinds of error messages you may encounter include:

(1) 'EXPECT keyword INSTEAD OF symbol' or 'EXPECT punctuation INSTEAD OF symbol'. Check your grammar against the specification of input format given in section 2.

(2) 'xxxx MUST BE yyyy'. These are usually self-explanatory. For example, the symbols appearing in a %TOKEN statement must be terminal symbols.

(3) 'MAL-FORMED xxxx'. The object referred to is a multi-character punctuation symbol, such as '/*' or '::='. Check your typing.

(4) 'UNDEFINED NONTERMINAL xxxx' or 'UNUSED NONTERMINAL xxxx' You either forgot to define a nonterminal which appears on the right hand side of a rule, or you define a nonterminal which does not appear on the right hand side of a rule and is therefore unnecessary. In either case, no parser is generated.

(5) 'READ ERROR AT LINE nnn WHILE READING xxxx BEGINNING AT LINE iii' You probably forgot a '*/', an %END_DECLARATIONS, or an %END_PROGRAMS, and DEREMER is interpreting the rest of your input file as literal text. No parser is generated.

(6) 'SHIFT-REDUCE CONFLICT ...' These are discussed under Ambiguity Resolution using Precedence and Associativity above. Fix them by some combination of: (a) rearranging the grammar (almost always the preferred method), (b) defining the precedence and/or the associativity of tokens, or (c) using the %PREC mechanism to override the default precedence of a production.

(7) 'REDUCE-REDUCE CONFLICT ...' Sometimes this is unavoidable, resulting from the need for more than 1 lookahead symbol. Sometimes, however, you can obtain the effect of more lookahead by delaying the point at which the reduction occurs. For example, consider the grammar

```
<S> ::= <X> b x
      | <Y> b y
;
<X> ::= a
;
<Y> ::= a
;
```

This grammar allows two legal input strings, 'a b x' and 'a b y'. Presumably, the semantics attached to the two reductions '<X> ::= a' and '<Y> ::= a' are different enough to require separate definitions. After the token 'a' has been read and shifted, it is not possible to decide which reduction to perform--the token which determines the correct reduction is hidden. One way to rearrange the grammar is to write

```
<S> ::= <X> x
      | <Y> y
;
<X> ::= a b
;
<Y> ::= a b
;
```

which defers the reduction to <X> or <Y> until either x or y is the lookahead symbol. The semantics must be adjusted accordingly.

(8) 'WARNING: LANGUAGE NOT CLEARLY TERMINATED. This parser may need to read an extra symbol.' The language defined by your grammar (that is, by the start symbol) contains at least one pair of legal strings such that the first string is a prefix of the second string. Calc1 would be such a grammar if the 'end_' symbol were omitted. Consider the input strings '3' and '3+4'. When the parser for this grammar has read the symbol '3', it must decide whether to accept (reduce) or to read on (shift). It makes the decision by reading one extra symbol to see whether it is a '+' (or a '*'). If it is neither, the parser accepts '3'. However, the extra symbol (which may be the first symbol of the next input) has already been read.

You can eliminate this problem by defining an end-of-input token, either as part of the input language itself or as something the lexer supplies. This is illustrated by the token 'end_' in the example grammar Calc1 above.

13 References

[Aho&Ullman 77] Alfred V. Aho and Jeffry D. Ullman, Principles of Compiler Design, Addison-Wesley 1977 (especially chapter 6).

[DeRemer 69] Franklin L. DeRemer, "Practical Translation for LR(k) Languages", Ph.D thesis, MIT 9/69

[DeRemer 71] Franklin L. DeRemer, "Simple LR(k) Grammars", CACM 7/71 p. 453

[Johnson 74] Stephen C. Johnson, "YACC - Yet Another Compiler Compiler", Bell Laboratories Computer Science Technical Report 32

[Poonen 77] George Poonen, "Error Recovery for LR(k) Parsers", submitted for publication in "Computer Software".