To: MTB Distribution

From: P. Prange, Benson I. Margulies

Date: October 4, 1982

Subject: LALR, a Translator Construction System

This Technical Bulletin describes the LALR system. LALR translates a BNF-like language description into a parser for the language. The output from LALR is a set of tables that control the operation of a parser procedure. Because these tables are lists of signed integers they can be easily transported to computers other than Multics. The parser procedure is a simple routine and versions of it have been coded in PL/I, COBOL and Assembly language. LALR has options which allow the control tables to be generated as a Multics object segment, an ALM source segment, a GMAP source segment or a DPS 6 (or Level 6) Multics Host Resident System object segment.

The parser created by LALR (the tables along with the parser procedure) is a "bottom-up" LALR(k) algorithm that examines the input symbols in a left to right manner, looks no more than k symbols ahead, does no backtracking and halts immediately if an input symbol is not acceptable. The size of the control table and the code for the parser procedure is competetive with hand-coded methods. LALR is an expedient means to provide parsers for computer languages.

The attribute of immediate error detection is accompanied by facilities for error recovery. Because error recovery is language related, no particular scheme is imposed. The tabular form of parser provides for a variety of error analyses.

LALR requires that the user provide a description (a grammar) of the language for which a parser is desired. This also serves as a document to describe the syntax (allowable symbol arrangements) to people who will use the language. LALR assures the correspondence between what a language is published to be and the parser that "says" what the language "is".

Because of LALR's speed of operation, frequent adjustment can be made to the language description until the user is satisified. Immediate test parses can be performed to observe the operation of the parser. LALR assures that a compiler or translator will be constructed in a modular fashion (unless the user goes out of his way to do otherwise). First the parser can be developed and checked, next the scanner and finally the semantic routines. Each can be tested before being incorporated in the translator.

For comparison purposes, a version of calc was developed using LALR. The compilation and generation listings are included at the end of this Technical Bulletin. This version was run against the installed one for a few cases. The execution time of the LALR version was from 98% to 144% of that of the installed calc. The bound object size of the LALR version was 64% of that of the installed one. It took 7 1/2 hours to complete.

Glossary

grammar - a formal set of rules that define a language. In general, a grammar involves four quanities: terminals, non-terminals, a start symbol, and productions.

terminals - the basic symbols of which strings in the language are composed.

non-terminals - special symbols that denote sets of strings.

variables - another name for non-terminals.

start symbol - a selected non-terminal which denotes the language we are truly interested in. The other non-terminals are used to define other sets of strings, and these help define the language.

sentence - a string of terminal symbols that may be derived from the grammar's start symbol in one or more steps.

complicated terminal - a pseudo-symbol of a language. It is treated like a terminal in a grammar, but it lexically is one of a set of symbols; e.g., <integer>.

rule - a description of a valid combination of symbols in a language. There may be alternatives.

production - a single valid combination of symbols. Equivalent to a rule if there are no alternatives. If a rule has n alternatives, it then represents n productions.

DPDA - Deterministic Push-Down Automata

EOI - end of information. This is the final terminal of an input.

Overview

This document contains information describing Multics commands comprising the LALR system. The LALR system was originally created by J. Falksen and Dave Ward of LISD. It has been extensively modified to improve its performance and to add functionality needed by the Ada/SIL project. You do not have to master all of this information to attempt a use of LALR. Various parts are of interest only after you have tried LALR and are selecting among different approaches in using LALR to aid in the implementation of a translator.

The following are typical steps taken to examine the use of lalr:

1. Prepare a sample grammar, the input to lalr. (See Source format, page 6 and Grammar format, page 13).

2. Execute lalr. (See lalr, page 20).

3. Repair the grammar if it is not acceptable (scratch head). (e.g., use the ted text editor). See Non-LALR (k) Grammars, page 71, for information on the interpretation of certain diagnostics.

4. Test the parser by executing lalrp, after the grammar is accepted by lalr. (See lalr_parse, page 49).

5. If the facilities of lalr_parse are sufficient, you then supply your semantics for that environment. If desired, write a scanner following the lalr_parse interface requirements.

6. Otherwise, you supply your semantics and scanner to match whatever interface requirements you decide on. You then generate your parser procedure with the macro (See Parser macro, page 56).

Consideration will be needed to accommodate error reporting and recovery. (See Error Recovery, page 16). Recovery can not be guaranteed to work under all circumstances or for all languages. You can anticipate a need for trade-offs and compromises.

If you require unreserved keywords, realization of the limitations of the provision from them by LALR must be understood. (See Unreserved Keywords, page 16).

Both error recovery and unreserved keywords are an extension to the context free parsing that lalr is limited to. Use of these facilities "breaks the rules".

Processor functions

An LALR language processor is made up of three parts:

scanner **parser** **semantics**

The **scanner** recognizes symbols in the input. It must know what the encoding of each symbol is to be, but it does not need to know the format of the parse tables.

The **parser** recognizes rules, i.e., valid combinations of symbols as defined by the grammar. It needs to know the format of the parse tables and the encoding of symbols, but it does not need to know anything about the form of these symbols.

The **semantics** represent the action to be taken when a rule has been recognized. It needs to know nothing about the format of the parse tables. It probably needs to know nothing about what makes up symbols.

Division of labor

The job to be done, processing a source input of a language, can be broken up in several different ways. The user makes his own decision as to which he likes.

Certain types of recognition processes can be described in the grammar (parsed) or done by the scanner.
A user could write a grammar like this:

```
<letter> ::= a | b | ...  | z | A | ...  | Z !
<digit> ::= 0 | 1 | ...  | 9 !
<symbol> ::= <letter> | <symbol> <letter> | <symbol> <digit> !
```

Then his scanner would be very simple, and would encode values for the letters and digits. This would, however, be very slow because of many rules being processed for each symbol.

Or the user could drop the first two rules and have the scanner smart enough to recognize <letter> and <digit>. This would parse more quickly.

Or the user could drop all three rules and have the scanner implement this directly and return an encoding for <symbol>. This is usually the best way to do it. It shortens the grammar, making it more readable. It speeds up the parse by having many less rules to works its way through.

If a scanner recognizes a symbol <integer>, for example, there is still the choice of whether the scanner or semantics actually converts the integer string to binary.

Source Format

The source segment can be in one of two forms:

1) grammar only
2) control lines followed by grammar

If the first character of the segment is a "-" it contains control lines. If not, then the grammar begins with the first character. The control arguments contained in the source segment must begin in the first character position of the line.

When control lines are present, they are selected from this set:

-ada_sil
-alm
-asm
-controls, -ctl
-count, -ct
-dpda
-dpda_xref, -dx
-embedded_semantics
-end_of_information {X}, -end_of_info {X}, -eoi {X}
: spaces and/or horizontal tabs separate the keyword from the X.
-gmap
-hash N
: spaces and/or horizontal tabs separate the keyword *from* the N.
-line_length N, -ll N
: spaces and/or horizontal tabs separate the keyword from the value N.
-list, -ls
-long_source, -lgsc
-mark X
: spaces and/or horizontal tabs separate the keyword from the X.
-no_ada_sil
-no_alm
-no_asm
-no_controls, -nctl
-no_count, -nct
-no_dpda_xref, -ndx
-no_end_of_information, -no_end_of_info, -neoi
-no_gmap
-no_list, -nls
-no_long_source, -nlgsc
-no_mark
-no_optimize, -not
-no_optimize_applies
-no_optimize_looks
-no_optimize_reads
-no_production_names, -npn
-no_semantics, -nsem
-no_semantics_header, -nsemhe
-no_source, -nsc
-no_symbols, -nsb
-no_table, -ntb
-no_terminals, -no_terms, -no_term

-no_terminals_hash_list, -nthl
-no_terminals_list, -ntl
-no_time, -ntm
-no_variables_list, -nvl
-nss
-nssl
-optimize, -ot
-optimize_applies
-optimize_looks
-optimize_reads
-production, -prod
-production_names, -pn
-rule
-semantics X, -sem X
spaces and/or horizontal tabs separate the keyword from the X.
-semantics_header, -semhe
-separate_semantics, -sep_sem
-source, -sc
-ss
-ssl
-symbols, -sb
-table X, -tb X spaces and/or horizontal tabs separate the keyword from the X.
-terminals, -terms, -term
-terminals_hash_list, -thl
-terminals_list, -tl
-time, -tm
-variables_list, -vl

For a description of the above control lines see the description of the corresponding control *arguments* of the lalr command beginning on page 20.

-order t t ... This specifies the order which should be used when assigning encodings to terminals. The first terminal will receive 1, the second 2, etc. White space or comments (see page 13) separates the keyword from the first terminal. Thereafter, each terminal is separated from the next by white space or comments. This control lasts up until the next line which begins with a "-". If the order control is present, all terminals are expected to be listed in it. A diagnostic is issued for each symbol not listed in the order control which is contextually determined to be a terminal symbol.

-synonyms list — This specifies sets of terminals that the scanner is to consider to be synonyms. Each set of synonymous terminals is given on a separate line with the synonyms being separated from each other by white space (other than NL) or by comments (see page 13). The first may be preceded by white space and comments and the last may be followed by white space and comments.

The first symbol in each set of synonyms is nominally considered to be the terminal. This symbol may, but is not required to be named in the -order control. If this symbol appears in any prior line of the -synonyms control, the entire current line is treated as a continuation of that prior line.

The second and succeeding symbols in each line are considered to be synonyms of the first symbol on the line. None of these symbols may be named in the -order control nor may they have appeared earlier in the -synonyms control.

Unless all of the terminals, excluding the synonyms, have been named in the -order control, use of the -synonyms control will cause gaps to exist in the sequence of integers encoding the terminals.

-recover t t ... — This specifies terminals for skip-recovery. See Error Recovery. The format is like -order.

-prelude text — This specifies a "standard prelude" that is to be scanned before scanning the normal source segment when parsing a source segment.

-parse — This specifies that everything following the keyword in the segment is the grammar. This must occur last in the control portion of the segment.

The source segment may be in a format called the embedded semantics format or in another format called the separate semantics format. The -embedded_semantics and -separate_semantics controls are used to specify which of these formats is in use.

In the embedded semantics format, the source segment is really a PL/I procedure, a Ada/SIL program unit, or a DPS 6 (or Level 6) Assembly Language program. The following paragraphs describe the creation of the semantics segment from an embedded semantics source segment.

If the source segment is a PL/I procedure (as indicated by the -semantics control argument), LALR will create the compileable semantics segment from it by the following steps.

1) Begin the semantics segment with a <procedure statement> naming the procedure. If the semantics segment is named X.pl1, the following <procedure statement> is generated:

```
X:  proc (rule_no, alt_no, lex_stack_ptr, ls_top);
```

If the semantics segment is named X.incl.pl1, the following <procedure statement> is generated:

```
X:  proc (rule_no, alt_no);
```

If the -production control (see the lalr command on page 20) has been given, the parameters rule_no and alt_no in the above <procedure statement>s are replaced by the single parameter prod_no.

2) Append a <comment> giving the name of the input grammar segment, the date and time it was translated, the version of LALR that was used to translate it, and the user_id of the user who translated it.

3) Append a <declare statement> declaring the formal parameters. If the semantics segment is named X.pl1, the <declare statement> is as follows:

```
dcl (rule_no fixed bin,
     alt_no fixed bin,
     lex_stack_ptr ptr,
     ls_top fixed bin) parameter;
```

If the semantics segment is named X.incl.pl1, the <declare statement> is as follows:

```
dcl (rule_no, alt_no) fixed bin parameter;
```

If the -production control has been given, the declaration of the formal parameters rule_no and alt_no in the above <declare statement>s are replaced by a declaration of a single fixed bin parameter "prod_no".

4) Append a <goto statement> to the semantics segment. If the -production control has not been given, the <goto statement> is as follows:

```
go to rule (rule_no);
```

If the -production control has been given, the <goto statement> is as follows:

```
go to prod (prod_no);
```

5) Append the source segment to the semantics segment making the following changes:

   a) Put /* and */ around the control portion, if present.
   b) Put /* and */ around each LALR rule.
   c) If the -production control (see the lalr command described on page 20) has not been given, each %%%% in the semantics is replaced

with the zero suppressed number of the rule which it represents. If the -production control has been given, each %%%% immediately followed by an unsigned decimal number representing an alternative number is replaced with the zero suppressed number of the production which they represent.

6) Append the following <end statement> to the semantics segment:

```
end X;
```

If the -no_semantics_header control (see the lalr command described on page 20) has been given, only steps 2 and 5 above are performed.

If the source segment is a Ada/SIL program unit (as indicated by the -semantics control argument), LALR will create the compileable semantics segment from it by the following steps.

1) Begin the semantics segment with a <subprogram specification> naming the subprogram. If the semantics segment is named X.ada, the following <subprogram specification> is generated:

```
procedure X
          (rule_no: in natural;
           alt_no: in natural;
           lex_stack_ptr: in access;
           ls_top: in integer) is
```

If the semantics segment is named X.incl.ada, the following <subprogram specification> is generated:

```
procedure X
          (rule_no: in natural;
           alt_no: in natural) is
```

If the -production control (see the lalr command on page 20) has been given, the formal parameters rule_no and alt_no in the above <subprogram specification>s are replaced by a single input formal parameter "prod_no" of type natural.

2) Append a sequence of <comment> lines giving the name of the input grammar segment, the date and time it was translated, the version of LALR that was used to translate it, and the user_id of the user who translated it.

3) Append the source segment to the semantics segment making the following changes:

   a) Put -- in front of each line of the control portion, if present.
   b) Put -- in front of each line of each LALR rule. If a rule does not begin at the beginning of a line or end at the end of a line, lines are split as necessary to make each rule do so.
   c) If the -production control (see the lalr command described on page 20) has not been given, each %%%% in the semantics is replaced with the zero suppressed number of the rule which it represents. If the -production control has been given, each %%%% immediately followed by an unsigned decimal number representing an alternative number is replaced with the zero suppressed number of the production which they represent.

4) End the <subprogram body> with the following text:

```
end X;
```

If the -no_semantics_header control (see the lalr command described on page 20) has been given, steps 1 and 4 above are skipped.

If the source segment is a DPS 6 (or Level 6) Assembly Language program unit (as indicated by the -semantics control argument), LALR will create the assembable semantics segment from it by the following steps.

1) If the semantics segment is named X.nml or X.nml.MAC, it begins with the title statement:

```
title X,'yymmdd00'
```

   where yymmdd is the current date. If the semantics segment is named X.incl.nml, it begins with the comment lines mentioned in step 2 below.

2) Comments lines giving the name of the input grammar segment, the date and time it was translated, the version of LALR that was used to translate it, and the user_id of user who translated it are placed in the output semantics segment.

3) Append the following statements defining the semantics procedure's entry point and transfering control to the semantics for the current rule.

```
        xdef    X
X       lab     $B4,jtable-1
        ldr     $R1,$B4.$R1
        jmp     $B4.$R1
```

These statements assume the parser passes the rule number or production number, as appropriate, by value in register R1. (See the -production control argument of the lalr command beginning on page 20 for information regarding use of rule numbers and production numbers.)

4) Append the source segment to the semantics segment making the following changes:

   a) Put a * in front of each line of the control portion, if present.
   b) Put a * in front of each line of each rule. If a rule does not begin at the beginning of a line or end at the end of a line, lines are split as necessary to make the rule do so.
   c) If the -production control (see the lalr command described on page 20) has not been given, each %%%% in the semantics is replaced with the 4-digit number of the rule which it represents. If the -production control has been given, each %%%% immediately followed by an unsigned decimal number representing an alternative number is replaced with the 4-digit number of the production which they represent.

5) Append a DC statement defining the jump table used by the statements shown in step 3 above. If the -production control has not been given the jump table is as follows:

```
jtable    dc        R0001-jtable+1;
                    R0002-jtable+1;
          ...
                    Rn-jtable+1
```

The jump table contains an entry for each rule of the grammar. If the i-th rule has a significant semantic, Ri used in the i-th line of the DC statement is the letter "r" followed by the value of i as a 4-digit decimal number. Otherwise, Ri is "no_sem". (The user is assumed to have defined the tag "no_sem" somewhere in the semantics segment.)

If the -production control has been given the jump table is as follows:

```
jtable    dc        P0001-jtable+1;
                    P0002-jtable+1;
          ...
                    Pn-jtable+1
```

The jump table contains an entry for each production of the grammar. If the i-th production has a significant semantic, Pi used in the i-th line of the DC statement is the letter "p" followed by the value of i as a 4-digit decimal number. Otherwise, Pi is "no_sem". (The user is assumed to have defined the tag "no_sem" somewhere in the semantics segment.)

6) Append the following end statement to the semantics segment if it is named X.nml or X.nml.MAC.

```
end X
```

In the separate semantics format, the semantics are not present in the source segment. In this format the grammar merely names an external entry to be called to perform the required semantic action.

Grammar Format

A grammar consists of rules written in a BNF-like notation. Each rule can have associated semantics. The semantics represent coding which is to be executed when a production of the rule described has been recognized. In embedded semantics source segments, the rules have this basic form:

```
<var> ::= <prod list> ! <semantics>
```

<var> — represents a "variable" (non-terminal). It must be the first non-white-space on a line. It begins with a "<" and ends with a ">".

::= — represents "is defined as". It must be on the same line as the <var>.

<prod list> — represents a production list. A production is a sequence of terminals and variables. If there is a list of them, they are separated by "|". The production list may be empty.

! — represents "end of production list". Everything following it is semantics. This must always be present.

<semantics> — represents the coding which is to be executed if the rule is parsed; it may be null. This cannot contain the string "::=".

(* ... *) — represents a "comment" within the grammar, it must be between the ::= and ! of a rule or within "-order", "-synonyms", or "-recover" control lines.

?include X — represents an "include macro". The include macro is processed as if it were replaced by the segment named X.incl.lalr found using the translator (trans) search paths. LALR allows the translator search paths to specify archives as well as the usual directories. An archive is specified to the search path commands by giving the pathname of the archive, including the suffix archive. Include macros may be nested. They may not appear in the control lines of the source segment nor may they appear between the <var> and ::= of a rule.

In separate semantics source segments, the rules have this basic form:

<var> ::= <prod list> ! <rule semantics>

<var> represents a "variable" (non-terminal). It begins with a "<" and ends with a ">".

::= represents "is defined as".

<prod list> represents a production list. A production is a sequence of terminals and variables. If there is a list of them, they are separated by "|". The production list may be empty. If the -production control is in effect, a production may end with the symbols => t : p$e, where t is an identifier tagging the production and p$e identifies an entry point in an external procedure to be called to perform the semantic action. If no tag is needed, t and the : following it may be omitted. There may not be any white-space between p and the dollar sign nor between the dollar sign and e. If p and e are the same, the $e may be omitted.

When the parser tables are produced as a Multics object segment or an ALM source segment, p is taken to be a segment name and e is considered an entryname. Each t generates an external static variable initialized with the corresponding production number.

When the tables are produced as a GMAP source segment, p is ignored and e is taken to be an external symbol; i.e., it has been SYMDEF'ed. Each t generates a word, tagged with t, containing the corresponding production number. Each t is also SYMDEF'ed.

When the tables are produced as a DPS 6 object unit, p is taken to be the name of an object unit and e is considered to be an entry point defined within that object unit. If the -asm control is used to request the object unit, each t names an external value equal to the corresponding production number. If the -ada_sil control is used to request the object unit, each t generates a variable of type integer which is initialize with the corresponding production number.

! represents "end of production list". This must always be present. If the -rule control is in effect, the ! of each rule may be followed by the symbols => t : p$e, where t, p, and e are as described above except that they pertain to rules instead of productions.

(* ... *) represents a "comment" within the grammar. If control lines are present, it may only appear within the -order, -synonyms, or -recover control lines or after the -parse control. If control lines are not present, it may appear anywhere.

?include X represents an "include macro". The include macro is processed as if it were replaced by the segment named X.incl.lalr found using the translator (trans) search paths. LALR allows the translator search paths to specify archives as well as the usual directories. An archive is specified to the search path commands by giving the pathname of the archive, including the suffix archive. If control lines are present, an include macro cannot appear befor the -parse control. If control lines are not present, include macros can appear anywhere.

Observe some LALR detail:

1. Rule ordering is unimportant, except that the rule that defines the "start symbol" must be physically first.

2. Ordering of productions (rule alternatives) is unimportant.

3. Each rule must be terminated by an exclaimation mark, "!". It is after this mark that semantic code is placed when using the embedded semantics format.

4. LALR reserves the use of the symbols, "<", "::=", "|", "'", "!" and "?include". When processing a separate semantics source segment, the symbol => is also reserved. Spaces are not required except between adjacent terminal symbols, i.e., "<0>::=+|-!" is acceptable.

5. To specify symbols involving these reserved characters and "space" characters the following escape character convention is implemented. The apostrophe, "'", signals an escaped character. It may be followed by an octal number up to three digits long, whose value specifies the Multics ASCII character desired, or if not followed (immediately) by an octal digit whatever character does follow is the character being escaped, i.e., "' ", "'40", and "'040" all indicate one blank character. This escape convention causes the restriction of the use of the apostrophe character, i.e., '' is required (or '047) to specify the "'" character itself.

6. Variables are "normalized" in the following manner: Any spaces immediately after the "<" bracket and immediately preceding the ">" bracket are deleted. Any internal strings of spaces are each replaced by a single space. This removes space sensitivity from variable names. "space" in this context refers to SP, HT, NL, NP, or VT.

The parsing of the LALR input treats all occurances of <...> as a variable as far as normalization is concerned. However, this is not what determines its being a variable; this is done only by appearing at the beginning of a rule. Any others may be considered as "complicated terminals". This means that you intend to have your scanner smart enough to know what <integer> is, for example.

Unreserved keywords

LALR parsing can handle unreserved keywords in a context-free setting. In general, if each statement has an initial keyword to insure proper recognition of statements, then <identifiers> can include symbols which are identical to keywords.

A read state contains a list of terminal encodings in increasing order which are valid in the input at this point. When keywords are to be unreserved, you must specify one terminal as an alternative to the keywords. This is done with the -mark option. Then all keywords which are to have this as their alternative must be given encodings which are higher than the alternative.

Suppose you said:

```
-order + - <integer> = <symbol> let if
-mark <symbol>
```

Then you could recognize the statement:

```
let let = let + 1
```

The lookup procedure in a read table when there are unreserved keywords is this:

> While doing a linear search of the read table, note whether a negative terminal exists. If there is one, compare its absolute value against the current terminal. Also remember what this one is. If the search fails, but a negative (marked) terminal was found, use it.

Error recovery

Error recovery is, in general, a very specific thing which is highly dependent on your language. It is not usually an easy thing to take care of.

One simple case is in an interactive interpreter. It can just discard the rest of the line and start in fresh on the next line. It is usually not that easy.

Two approaches have been developed along with the LALR compiler; local recovery and skip recovery. The "quality" of these recoveries is affected by optimization of the DPDA (see the lalr command, page 20) and use of deferred actions in the parser (see Parser macro, page 56). Generally, the ordering from highest quality to lowest quality is:

optimized looks, deferred actions
no optimized looks, deferred actions
no optimized looks, no deferred actions
optimized looks, no deferred actions

Local recovery

Local recovery uses the previous input symbol (when it is known), the current (unacceptable) input symbol, and the next two or three input symbols. First, all possible parses from the current state are simulated. These trial parses are true simulations of what can happen, apply states are chosen according to the simulated top of the parse stack. After the parses beginning in the current state are exhausted, several parses beginning in the state that read the previous input symbol are simulated if that state is known and the parser has been generated with the "deferred actions" feature (see Parser macro, page 56).

Given:

A is an alternate symbol
P is the previous symbol
B is the current (bad) symbol
N is the next symbol
T is the second next input symbol
U is the third next input symbol
H is the previous read state
C is the current state
R is a "next" read state
F is a "next" read state following R
G is a "next" read state following F

The following table indicates the recoveries that are possible if the states named in the column headings can accept the indicated symbols.

H	C	R	F	G	
P	N	B	T	x	Reverse B and N
P	N	T	x	x	Delete B
P	A	B	N	T	Insert A before B
P	A	N	T	x	Replace B with A
B	P	N	T	x	Reverse P and B
B	N	T	x	x	Delete P
A	P	B	N	T	Insert A before P
A	B	N	T	x	Replace P with A
N	T	U	x	x	Delete P and B
P	T	u	x	x	Delete B and N
P	A	T	U	x	Replace B and N with A
A	N	T	U	x	Replace P and B with A

The recovery tries to find a useable combination among the first four types of repair. If one exists, it is remembered but the search does not stop. If a second one is found, the search stops, a message is generated which says the choice is not unique, and then the first combination is used. If only one useable combination is found, it is used with a message indicating it to be unique. If no usable combination is found, the parser has been generated with the "deferred actions" feature, the parse did not fail in a multiple look ahead state, and the last input symbol read is known, the remaining combinations are tried. (The parse will never fail in a multiple look ahead state if the grammar was processed with optimized looks, see the lalr command described on page 20. The last input symbol read is known if the state on the top of the parse stack is a read state as opposed to an apply state.) If a useable combination is found, the search continues as above, however it is restricted to repairs of the same type.

Only terminals whose encoding is less than that of the nil symbol (see skip recovery below) are considered as alternate symbols by local recovery.

There is a special precedence rule for the delete B and insert A before B repairs. If both repairs are possible (reverse B and N is not), delete B is performed if the encoded value of B is less than the smallest encoded value of A; otherwise insert A before B is performed.

Local recovery operates as described above when the parser is generated with 2 for the local_reads parameter (see Parser macro described on page 56). The local_reads parameter specifies the number of symbols beyond the bad symbol that must be accepted for a particular recovery to be considered successful. If, for example, 1 is given for local_reads, the following table is used.

H	C	R	F	
P	N	B	x	Reverse B and N
P	N	x	x	Delete B
P	A	B	N	Insert A before B
P	A	N	x	Replace B with A
B	P	N	x	Reverse P and B
B	N	x	x	Delete P
A	P	B	N	Insert A before P
A	B	N	x	Replace P with A
N	T	x	x	Delete P and B
P	T	x	x	Delete B and N
P	A	T	x	Replace B and N with A
A	N	T	x	Replace P and B with A Skip recovery

Skip recovery requires that the user define one or more recovery terminal symbols by means of the

```
-recover <nil> st1 st2 ...
```

control included in the lalr source. st1 st2 etc. are skip terminals. They are terminals which can end statements. They cause a table to be built for skip recovery. This table is a list of read and look ahead states which can follow the reading of a skip terminal or can be the first state to read a terminal. These states correspond to the beginnings of new statements.

Skip recovery is done when an error has occurred and local recovery (if used) was not successful. Basically what it does is to skip forward in the source by calling the scanner until it encounters one of the skip terminals. It then looks backward in the parse stack for a read state or a state applying an empty production which could have followed a state that read a previous occurrence of the skip symbol just found. If one is found, it tentatively adjusts the lexical stack top (which is also the parse stack top) and then proceeds with a trial parse. If the path from the state which could have read the skip terminal to the read or empty apply found above has a sequence of look ahead states (with no intervening non-empty apply states) leading to its ending state, the trial parse starts in the first of these look ahead states, otherwise it starts in the path's ending state.

Effectively a bad "statement" has been discarded. In this case "statement" means an input string ending in a skip terminal which could have followed the *identical* skip terminal (such as ";" for example). It includes the boundary terminal on the right. If the language is such that the discarded statement is optional (syntactically) the rest of the input can be checked for syntax errors. Note that two identical statements need not be parsed beginning in the same read state; e.g., the first of a sequence of statements could be parsed beginning in one read state while the remaining statements could be parsed beginning in some other read state.

When a bad "statement" is discarded the parser is restarted in the state in which it began to process the statement. If the next N input symbols encountered are not acceptable from that state, the parser makes another attempt at error recovery by replacing the bad "statement" with the <nil> symbol defined by the -recover control and starting a second trial parse from this symbol. If neither trial parse is able to accept the next N input symbols and M pairs of trial parses have not yet been attempted for the current symbol, skip recovery looks further backward in the parse stack for a different read state which could have followed a state that read a previous occurrence of the skip symbol found above. The trial parsing described above is then repeated.

If none of the trial parses is able to accept the next N input symbols or all states on the parse stack are exhausted, skip recovery starts over without having made an adjustment to the stacks. To appreciate the effect of looking deeper in the parse stack consider the situation where the first trial parse attempts to accept a <simple statement> and fails. Now assume M > 1 and the second trial parse attempts to accept a <compound statement>. It is possible to obtain better recoverys with M = 2 than with M = 1 when such situations can occur. When one of the trial parses accepts the next N input symbols, the lexical and parse stack adjustment is made final and normal parsing resumes.

Before starting the recovery process described above the parser pushes the current state, or a read state following it if it makes only look transitions, onto the parse stack. This serves two purposes. First, it ensures that the parse can restart in the current state when the error occurs on a terminal immediately following a skip terminal. Second, it allows skip recovery to be done when the parse fails before reading any terminals.

The <nil> symbol is one which the scanner must NEVER return. It is needed because some languages do not allow all statements to occur at every point. This means that when you back up to the last statement beginning point, you may not be allowed to have the statement you find next. As an example, take this grammar:

```
<g> ::= <i> | <g> <i> !
<i> ::= <a> | <b> !
<a> ::= a ; <rd> !
<rd> ::= r ; | <rd> r ; !
<b> ::= b ; <sd> !
<sd> ::= s ; | <sd> s ; !
```

Then suppose that you intended to have an input like line (1) below, but instead you got (2):

```
(1)   a ; r ; r ; b ; s ; s ; s ; a ; r ; r ; r ;
(2)   a ; r ; r ; b ; s ; s ; s   a ; r ; r ; r ;
```

When the "s" "a" ";" is encountered, local recovery will decide that "a" is extraneous and drop it. But this then means that it will miss the fact that it should be entering the <a> rule. It will then get to the "r" and local recovery will fail, necessitating another skip. In this example, skipping will occur, one statement at a time, until EOI is reached.

If the grammar had specified

```
-recover <nil> ;
```

skip recovery would skip to the next ";" and pick up where it was. But the only thing it finds in the stack is a state which can read either an "a", "b", or "s". So it will have to skip again. This means that no syntax checking is done in all of the "r"'s which are skipped. This is not highly desireable.

However, if you add a rule like this:

```
<a> ::= <nil> <rd> !
```

the generated <nil> from skip recovery will allow the <rd> to be correctly parsed, reducing the number of useless error messages by quite a bit, usually.

These <nil> rules can help parse through misplaced statements during error recovery, but will never accept these statements under normal circumstances. The semantics on these <nil> rules must then report an error.

Name: lalr, lrk

The lalr command invokes the LALR compiler to translate a segment containing the text of the LALR source into a set of tables. A listing segment is optionally produced. Packaged forms of the tables may be requested. These results are placed in the user's working directory.

Usage: lalr path {list_args} {ctl_args}

1) path — is the pathname of the LALR source segment containing the grammar to be processed. If path does not have a suffix of lalr, one is assumed. However, the suffix lalr must be the last component of the name of the source segment. This argument may be an archive component pathname.

2) list_args — may be one or more of the following optional arguments. If the source segment is named X.lalr, the list segment will be named X.g.list. This is done so that if the user choses to have his semantics file named X.pl1, the generation listing and compilation listing will not be in conflict.

-line_length N, -ll N
: causes the listing to be prepared with lines no longer than N characters. If this control argument is not specified, a line length of 136 characters is assumed.

-page_length N, -pl N
: prints the "machine" listing (see the -list control argument below) so that no more than N lines are on a page. If this control argument is not specified, a page length of 60 lines is assumed.

-source, -sc
: produces a line-numbered listing of the rules of the grammar. No semantics are listed, only the rules.

-long_source, -lgsc
: produces a line-numbered listing of the rules of the grammar and the associated semantics. This control is meaningless with separate semantics format source segments, that is it has the same effect as -source.

-symbols, -sb
: produces a cross reference listing of the terminals and variables used in the grammar. If the source segment is in the separate semantics format, a cross reference listing of the semantic actions used is also produced.

-list, -ls
: produces a "machine" listing of the DPDA resulting from the LALR execution.

-controls, -ctl
: includes the grammar's control lines, if any, in the output list. This control argument implies -source.

-count, -ct
: produces a list of statistics about the tables. This will go to user_output if no other option is present which provides a list segment.

-terminals, -terms, -term
: produces a listing of the terminals in encoding order, showing the encoding. If the source segment is in the separate semantics format, a listing of semantic actions indexed by the using rule number or production number, as appropriate, is also produced.

-ss
: produces source and symbols.

-ssl
: produces source, symbols, and list.

-dpda_xref, -dx
: includes cross reference lists of states, terminals, and variables in the "machine" listing of the DPDA. If the source segment is in the separate semantics format, the semantic actions are also cross referenced. In the first two of these lists, each referencing state number is immediately followed by the letter "R", "L", "A", "B", or "D" indicating a read transition, look transition, transition from an apply state, a look back reference by an apply state, or a look back reference implied by the default transition of an apply state, respectively. In the lists for variables and semantic actions each state number is immediately followed by the letter "S", "T", or "U" indicating an apply single, apply with look back table, or apply using shared look back table, respectively. This control argument implies the -dpda control argument.

-time, -tm
: prints a table after translation giving the time (in CPU seconds), the number of page faults, measures of other resources used by each phase of the translator. This information is also available from the command lalr$times invoked immediately after a translation.

-no_source, -nsc
: does not produce a listing of the grammar or the associated semantics. This is the default.

-no_symbols, -nsb
: does not produce a listing of the terminals and variables used in the grammar. This is the default.

-no_long_source, -nlgsc
: does not include the semantics in the source listing. This is the default. Note that -long_source -no_long_source is equivalent to -source.

-no_list, -nls
: does not produce a "machine" listing of the DPDA resulting from the LALR execution. This is the default.

-no_controls, -nctl
does not include the grammar's control lines, if any, in the output list if one is produced. This is the default. Note that -controls -no_controls is equivalent to -source.

-no_count, -nct does not produce a list of statistics about the tables. This is the default.

-no_terminals, -no_terms, -no_term
does not produce a listing of the terminals in encoded order. This is the default.

-nss is the same as -no_source -no_symbols

-nssl is the same as -no_source -no_symbols -no_list

-no_dpda_xref, -ndx
does not include any DPDA cross reference lists in the "machine" listing of the DPDA. This is the default. Note that -dpda_xref -no_dpda_xref is equivalent to -dpda.

-no_time, -ntm does not print a table after translation giving the amounts of CPU time and other resources used by each of the phases of the translator. This is the default.

3) ctl_args may be one or more of the following optional arguments.

-end_of_information {X}, -end_of_info {X}, -eoi {X}
Uses a production whose right hand side is the user's start symbol followed by an end-of-information symbol to create the augmented grammar. This is the default. If the optional argument X is present, it is made a synonym of the anonymous end-of-information terminal.

-no_end_of_information, -no_end_of_info, -neoi
Uses a production whose right hand side is simply the user's start symbol to create the augmented grammar.

-production, -prod
causes the DPDA to be generated with apply state tables that contain the production number but not the rule and alternative numbers. If this control argument is not given or is over ridden by a later -rule control argument, the apply state tables will contain the rule number and alternative number in addition to the production number. This control argument also affects the generation of the semantics segment (see Source Format on page 6).

-rule causes the DPDA to be generated with apply state tables that contain the rule and alternative numbers in addition to the production number. This is the default. (This control argument may be over ridden by a later -production control argument.)

-optimize_reads performs certain optimizations on the generated DPDA that primarly affect read states. The first of these optimizations eliminates all read transitions that serve only to read a looked ahead at terminal. Such read transitions are contained in read states that are not referenced in any apply state's look back table. If this optimization causes all of the transitions of a read state to be eliminated, the read state itself is also eliminated. The second optimization eliminates read states that read (only) the terminals looked at by a single look state and which are referenced in one or more apply states' look back table. This optimization is performed when only one (look) state makes a transition to the read state involved, that look state looks at all of the terminals read by the read state, and the look state is not already referenced by an apply state's look back table due to an earlier elimination of a looked back at state that read one or more terminals looked at by the look state. Other less significant optimizations are also performed.

Use of a DPDA with optimized reads requires a parser designed (or generated) according to the requirements given in the June 13, 1981 or later version of this specification (see Parser macro on page 56).

-optimize_applies

performs certain optimizations on the generated DPDA that primarly affect apply states. The most significant optimization performed is the elimination of apply states that do not apply an empty production, do not have a significant semantic action, do not do a look back, and do not delete any entries from the parse and lexical stacks.

-optimize_looks

performs certain optimizations on the generated DPDA that primarly affect look states. This optimization moves marked symbol transitions (see Unreserved keywords on page 16) to the beginning of the look-up table to allow a non-linear look-up and creates a default look transition in lieu of several look transitions to the same next state when possible. It also arranges for read/look tables to be truncated and continued at a similar state. Use of this optimization tends to cause errors to be detected later in the parse than is the case when the DPDA is not optimized.

Use of a DPDA with optimized looks requires a parser designed (or generated) according to the requirements given in the September 18, 1982 or later version of this specification (see Parser macro on page 56).

-optimize, -ot
: is the same as -optimize_reads -optimize_applies -optimize_looks.

-no_optimize_reads
: does not perform the optimizations primarly affecting read states. This is the default.

-no_optimize_applies
: does not perform the optimizations primarly affecting apply states. This is the default.

-no_optimize_looks
: does not perform the optimizations primarly affecting apply states. This is the default.

-no_optimize, -not
: is the same as -no_optimize_reads -no_optimize_applies -no_optimize_looks.

-embedded_semantics
: indicates that the source segment is in the embedded semantics format (see Source Format, page 6 and Grammar Format, page 13). This is the default.

-separate_semantics, -sep_sem
: indicates that the source segment is in the separate semantics format (see Source Format, page 6 and Grammar Format, page 13).

-semantics X, -sem X
: produces a semantics file named X. (X is any pathname other than an archive component pathname.) The suffix(s) must be pl1, incl.pl1, nml, incl.nml, nml.MAC, ada, or incl.ada. If no suffix is given, incl.pl1 is assumed. If incl is given, it is treated as incl.pl1. Note: this control argument is meaningless with a separate semantics format source segment.

-semantics_header, -semhe
: causes a "program header" to be generated for the semantics file. (See Source format described on page 6.) This is the default.

-no_semantics_header, -nsemhe
caused the "program header" to be omitted from the generated semantics file. This control argument is ignored when generating a DPS 6 Assembly language semantics file.

-no_semantics, -nsem
does not produce a semantics file. This is the default.

-mark X
mark terminal X (see Unreserved keywords, page 16)

-no_mark
generates a parser with no marked terminal. This is the default.

-hash N
set the hash value of the variable and terminal tables to N. The default is 1021.

-no_dpda, -nd
causes only the first pass and the listing passes of LALR to be executed. This allows a new semantics file to be created and/or listings to be produced at considerably less expense than a normal LALR generation. When this option is used, the result file (or a link to it) from a previous LALR generation using the source named by the path argument must exist in the working directory. Also the current grammar must be equivalent to the grammar that the result file was generated from and each rule (or alternative if the -production control was used) must have, or not have, a semantic action as did the same rule (or alternative) in the original grammar.

-dpda
causes the complete LALR procedure to be executed to generate a new result file. This is the default.

-no_table, -ntb
does not produce the table described below. This is the default. This control argument implies then -no_terminals_list, -no_terminals_hash_list, -no_production_names, and -no_variables_list control arguments described below.

-table X{.incl.pl1}, -tb X{.incl.pl1}
produces a table named X and appropriately named source files. (X is any pathname other than an archive component pathname.) The table is produced as a Multics object segment unless otherwise specified by the control described below. This control argument implies the -terminals_list, -variables_list, and -production_names control arguments described below.

-terminals_list, -tl
include the terminals list in the table.

-terminals_hash_list, -thl
: include the terminals list and terminals hash list in the table.

-production_names, -pn
: include the production names in the table. This control argument implies the -variables_list control argument described below.

-variables_list, -vl
: include the variables list in the table.

-no_terminals_list, -ntl
: does not include the terminals list in the table. This is the default. This control argument implies the -no_terminals_hash_list control argument described below.

-no_terminals_hash_list, -nthl
: does not include the terminals hash list in the table. This is the default. Note that -terminals_hash_list -no_terminals_hash_list has the same effect as -terminals_list.

-no_production_names, -npn
: does not include the production names in the table. This is the default. Note that -production_names -no_production_names has the same effect as -variables_list.

-no_variables_list, -nvl
: does not include the variables list in the table. This is the default. This control argument implies the -no_production_names control argument described above.

-no_alm
: does not produce the table in the form described below for the -alm control argument.

-no_gmap
: does not produce the table in the form described below for the -gmap control argument.

-no_asm
: does not produce the table in the form described below for the -asm control argument.

-no_ada_sil
: does not produce the table in the form described below for the -ada_sil control argument.

-alm
: produce the table as an alm segment X.alm and a Multics PL/I include file named X.incl.pl1. X is the name supplied with the -table control argument less all suffixes.

-gmap produce the table as a gmap segment X.gmap and a GCOS III PL/I include file named X.incl.pl1. X is the name supplied with the -table control argument less all suffixes.

-asm produce the table as a DPS 6 (or Level 6) Multics Host Resident System object file named X.object and produce a DPS 6 Assembly Language include file named X.incl.nml. X is the name supplied with the -table control argument less all suffixes.

-ada_sil produce the table as a DPS 6 (or Level 6) Multics Host Resident System object file name X.object and produce a DPS 6 Ada/SIL package specification named X.spec.ada. X is the name supplied with the -table control argument less all suffixes.

Notes: Options -alm, -gmap and -asm or -ada_sil may occur together. (Options -asm and -ada_sil are mutually exclusive.) If -alm, -gmap, -asm or -ada_sil is in effect but the -table parameter is not, the output segments for these parameters use the source segment name with the suffix lalr and the preceding "." replaced with "_t" in lieu of X.

The create_data_segment_ subroutine is used to create the Multics object segment unless a separate semantics format source segment is used. In this case, an alm source segment is created in the process directory and it is automatically assembled if possible. The contents of the Multics object segment produced by the -table X control argument are described by the following PL/I declarations. The generated include file X.incl.pl1 contains a copy of these declarations. When a separate semantics format source segment is used, the object segment also contains a transfer vector with the external name semantics_vector. This vector is used by the parser to call the various semantic actions. The rule number, or production number if the -production control is in effect, must be passed as the first argument in the call to the transfer vector. Any additional arguments desired may be passed. The generated include file does not describe the transfer vector.

```
dcl 1 X$terminals_hash_list external static,
    2 terminals_hash_list_size fixed bin,
    2 terminals_hash_list (0:xx)
        fixed bin (12) unsigned unaligned;
dcl 1 X$terminals_list external static,
    2 terminals_list_size fixed bin,
    2 terminals_list (xx),
      3 link fixed bin (18) unsigned unaligned,
      3 position fixed bin (18) unsigned unaligned,
      3 length fixed bin (18) unsigned unaligned,
      3 code fixed bin (18) unsigned unaligned;
```

```
dcl 1 X$terminal_characters external static,
    2 terminal_characters_length fixed bin,
    2 terminal_characters char (xx);
dcl 1 X$dpda external static,
    2 dpda_size fixed bin,
    2 dpda (xx),
      3 (v1, v2) fixed bin (17) unaligned;
dcl 1 X$skip external static,
    2 skip_size fixed bin,
    2 skip (xx),
      3 (v1, v2) fixed bin (17) unaligned;
dcl 1 X$standard_prelude external static,
    2 standard_prelude_length fixed bin,
    2 standard_prelude char (xx);
dcl 1 X$production_names external static,
    2 production_names_size fixed bin,
    2 production_names (xx) fixed bin (17) unaligned;
dcl 1 X$variables_list external static,
    2 variables_list_size fixed bin,
    2 variables_list (xx),
      3 (position, length)
           fixed bin (18) unsigned unaligned;
dcl 1 X$variable_characters external static,
    2 variable_characters_length fixed bin,
    2 variable_characters char (xx);
```

terminals_hash_list(i) is the terminals_list index of the first terminal symbol whose hash value is i. The function lalr_hash_ (contained in the include file lalr_hash_.incl.pl1), when invoked by lalr_hash_ (T, dim (terminals_hash_list, 1)), returns the hash value of the character string T. The X$terminals_hash_list structure is only generated when the -terminals_hash_list control argument is in effect.

The format shown above is generated when both the -terminals_hash_list and -terminals_list control arguments are in effect and synonyms have been defined. terminals_list(i).link is the terminals_list index of the next terminal symbol having the same hash value as the i-th terminal symbol. substr (terminal_characters, terminals_list(i).position, terminals_list(i).length) is the normalized spelling of the i-th terminal symbol. And finally, terminals_list(i).code is the encoded value of the i-th terminal symbol.

If the -terminals_hash_list and -terminals_list control arguments are both in effect but no synonyms are defined, the following structure is generated for the terminals list instead of the one shown above. When this structure is used, the encoded value of the i-th terminal symbol is i.

```
dcl 1 X$terminals_list external static,
    2 terminals_list_size fixed bin,
    2 terminlas_list (xx),
      3 link fixed bin (11) unsigned unaligned,
      3 position fixed bin (14) unsigned unaligned,
      3 length fixed bin (11) unsigned unaligned;
```

If the -terminals_hash_list control argument is not in effect but the -terminals_list control argument is in effect and synonyms are defined, the following structure is generated for the terminals list instead of one of those shown above.

```
dcl 1 X$terminals_list external static,
    2 terminals_list_size fixed bin,
    2 terminals_list (xx),
      3 position fixed bin (14) unsigned unaligned,
      3 length fixed bin (11) unsigned unaligned,
      3 code fixed bin (11) unsigned unaligned;
```

If the -terminals_hash_list control argument is not in effect but the -terminals_list control argument is in effect and no synonyms are defined, the following structure is generated for the terminals list instead of any of those shown above.

```
dcl 1 X$terminals_list external static,
    2 terminals_list_size fixed bin,
    2 terminals_list (xx),
      3 position fixed bin (18) unsigned unaligned,
      3 length fixed bin (18) unsigned unaligned;
```

If the -terminals_hash_list control argument is not in effect, a trivial structure (with terminals_hash_list_size = 0) is generated for X$terminals_hash_list and no declaration is generated for it. If neither the -terminals_hash_list nor the -terminals_list control argument is in effect, a trivial structure (with terminals_list_size = 0) is generated for X$terminals_list and a zero length string is generated for X$terminal_characters and no declarations are generated for them.

dpda and skip are the Deterministic Push Down Automata implementing the parsing algorithm and its associated error recovery tables. standard_prelude is the Standard Prelude. The X$dpda, X$skip, and X$standard_prelude structures are always generated.

production_names is the production names list. production_names(i) is the negation of the variables_list index for the variable (non-terminal) naming the i-th production. If the -production_names control argument is not in effect, a trivial structure (with production_names_size = 0) is generated for X$production_names.

variables_list is the variables list. substr (variable_characters, variable_list(i).position, variables_list(i).length) is the normalized spelling of the i-th variable. If neither the -production_names nor -variables_list control argument is in effect, a trivial structure (with variables_list_size = 0) is generated for X$variables_list and a zero length string is generated for X$variable_characters.

Each of the level 1 structures described above has two level 2 members, the first being a fixed bin scalar and the second being an array or a character string. In each case, the value of the first member is the upper bound or length, as appropriate, of the second member.

The alm source segment produced by the -alm control argument assembles to produce a Multics object segment as described above except that slack bytes are added between symbols stored in terminal_characters and variable_characters so as to make each symbol start on a word boundary.

The gmap source segment produced by the -gmap control argument is equivalent to the data described by the following PL/I declarations. The generated include file X.incl.pl1 contains a copy of these declarations (unless the -alm control argument is also in effect). When a separate semantics format source segment is used, the gmap source segment also contains a transfer vector with the external name SEMVEC. This vector is used by the parser to call the various semantic actions. The rule number, or production number if the -production control is in effect, must be passed as the first argument in the call to the transfer vector. Any additional arguments desired may be passed. The generated include file does not describe the transfer vector.

```
dcl 1 THL (0:xx) bit (12) unaligned external static;
dcl 1 TL (xx) external static,
    2 lk fixed bin (17) unaligned,
    2 pt fixed bin (17) unaligned,
    2 ln fixed bin (17) unaligned,
    2 cd fixed bin (17) unaligned;
dcl  TC char (xx) external static;
dcl 1 DPDA (xx) external static,
    2 v1 fixed bin (17) unaligned,
    2 v2 fixed bin (17) unaligned,
dcl 1 SKIP (xx) external static),
    2 v1 fixed bin (17) unaligned,
    2 v2 fixed bin (18) unaligned;
```

```
dcl PN fixed bin (17) unaligned external static;
dcl 1 VL (xx) external static,
    2 pt fixed bin (17) unaligned,
    2 ln fixed bin (17) unaligned;
dcl  VC char (xx) external static;
```

binary(THL(i), 12, 0) is the TL index of the first terminal symbol whose hash value is i. The function lalr_hash_ (contained in the include file lalr_hash_.incl.pl1), when invoked by lalr_hash_ (T, dim (THL, 1)), returns the hash value of the character string T. The THL structure is only generated when the -terminals_hash_list control is in effect.

The format shown above is generated when both the -terminals_hash_list and -terminals_list controls are in effect and synonyms have been defined. TL(i).lk is the TL index of the next terminal symbol having the same hash value as the i-th terminal symbol. substr (TC, TL(i).pt, TL(i).ln) is the normalized spelling of the i-th terminal symbol. And finally, TL(i).cd is the encoded value of the i-th terminal symbol.

If the -terminals_hash_list and -terminals_list controls are both in effect but no synonyms are defined, the following structure is generated for the terminals list instead of the one shown above. When this structure is used, the encoded value of the i-th terminal symbol is i.

```
dcl 1 TL external static,
    2 lk fixed bin (10) unaligned,
    2 pt fixed bin (13) unaligned,
    2 ln fixed bin (10) unaligned;
```

If the -terminals_hash_list control is not in effect but the -terminals_list control is in effect and synonyms are defined, the following structure is generated for the terminals list instead of one of those shown above.

```
dcl 1 TL external static,
    2 pt fixed bin (13) unaligned,
    2 ln fixed bin (10) unaligned,
    2 cd fixed bin (10) unaligned;
```

If the -terminals_hash_list control is not in effect but the -terminals_list control is in effect and no synonyms are defined, the following structure is generated for the terminals list instead of any of those shown above.

```
dcl 1 TL external static,
    2 pt fixed bin (17) unaligned,
    2 ln fixed bin (17) unaligned;
```

If the -terminals_hash_list control is not in effect, the THL structure is omitted. If neither the -terminals_hash_list nor the -terminals_list control is in effect, THL, TL, and TC are all omitted.

DPDA and SKIP are the Deterministic Push Down Automata implementing the parsing algorithm and its associated error recovery tables. The DPDA and SKIP structure are always generated.

PN is the production names list. PN(i) is the negation of the VL index for the variable (non-terminal) naming the i-th production. If the -production_names control is not in effect, the PN structure is not generated.

Vl is the variables list. substr (VC, VL(i).pt, VL(i).ln) is the normalized spelling of the i-th variable. If neither the -production_names control nor the -variables_list control is in effect, PN, VL, and VC are all omitted.

The -terminals_hash_list control argument is treated as if it were the -terminals_list control argument when producing a DPS 6 (or Level 6) object file. The -production_names and -variables_list control arguments are ignored when producing a DPS 6 object file. The DPS 6 object file is produced in LAF mode.

The DPS 6 object file produced by the -asm control argument is equivalent to the data described by the PL/I declarations below. When a separate semantics format source segment is used, the object file also contains a transfer vector with the external name SEMVEC. The rule number, or production number if the -production control is in effect, must be passed to the transfer vector by value in register R1. The transfer vector's code destroys registers R1 and B4, all other registers are unchanged.

```
dcl  OP1C_n fixed binary (15) internal static
     options (constant) initial (xx);
dcl  OP2C_n fixed binary (15) internal static
     options (constant) initial (xx);
dcl  RSWD_n fixed binary (15) internal static
     options (constant) initial (xx);
dcl  LIT_c fixed binary (15) internal static
     options (constant) initial (xx);
dcl  INT_c fixed binary (15) internal static
     options (constant) initial (xx);
dcl  NUMB_c fixed binary (15) internal static
     options (constant) initial (xx);
dcl  REAL_c fixed binary (15) internal static
     options (constant) initial (xx);
dcl  SYMB_c fixed binary (15) internal static
     options (constant) initial (xx);
dcl  EOL_c fixed binary (15) internal static
     options (constant) initial (xx);
dcl  HEXI_c fixed binary (15) internal static
     options (constant) initial (xx);
dcl  BIT_c fixed binary (15) internal static
```

```
     options (constant) initial (xx);
dcl  NIL_c fixed binary (15) internal static
     options (constant) initial (xx);
dcl  OP1C_s (xx) char (1) external static
     initial ("x", "x", ... );
dcl  OP2C_s (xx) char (2) external static
     initial ("xx", "xx", ... );
dcl 1 RSWD (xx) aligned external static,
    2 RSWD_s char (xx) initial ("xx", "xx", ... ),
    2 RSWD_c fixed bin (15) initial (xx, xx, ... );

dcl  DPDA_n fixed binary (15) internal static
     options (constant) initial (xx);
dcl  SKIP_n fixed binary (15) internal static
     options (constant) initial (xx);

dcl 1 DPDA (xx) external static,
    2 v1 fixed binary (15) initial (xx, xx, ... ),
    2 v2 fixed binary (15) initial (xx, xx, ... );
dcl 1 SKIP (xx) external static,
    2 v1 fixed binary (15) initial (xx, xx, ... ),
    2 v2 fixed binary (15) initial (xx, xx, ... );
```

The data with internal static options (constant) attributes are generated as "external value definitions" in the DPS 6 object file. The data with external static attributes are generated as "code section" constants with "external location definitions". OP1C_n and OP1C_s are the number of one character operators (e.g. +) and the one character operators themselves, respectively. OP2C_n and OP2C_s are the number of two character operators (e.g. >=) and the two character operators themselves, respectively. LIT_c is the code for the nonnumeric literal complicated terminal. This terminal may be specified as <character string>, <string>, <quoted string>, or <nonnumeric literal>. INT_c is the code for the integer literal complicated terminal. This terminal may be specified as <integer>. NUMB_c is the code for the fixed-point literal complicated terminal. This terminal may be specified as <number> or <fixed-point literal>. REAL_c is the code for the floating-point literal complicated terminal. This terminal may be specified as <real> or <floating-point literal>. SYMB_c is the code for the identifier complicated terminal. This terminal may be specified as <identifier> or <symbol>. EOL_c is the code for the end of line complicated terminal. This terminal may be specified as <eol>, <end of line>, <nl>, or <newline>. HEXI_c is the code for the hexadecimal integer literal complicated terminal. This terminal may be specified as <hexadecimal integer> or <hex integer>. BIT_c is the code for the bit string literal complicated terminal. This terminal may be specified as <bit string> or <boolean aggregate>. NIL_c is the code for the nil symbol terminal. This terminal may be specified as <nil> or <syntax error>. For any of the above mentioned complicated terminals not used in the grammar, a code of zero is used. RSWD_n, RSWD_k, and RSWD are the number of reserved words, the length of each reserved word, and the reserved words themselves, respectively. All terminal symbols that were not associated with a XXXX_c variable above are considered reserved words.

In RSWD (i), RSWD_s is the i-th reserved word padded with spaces and RSWD_c is the encoding for that reserved word. DPDA_n and DPDA are the number of DPDA entries and the DPDA itself, respectively. SKIP_n and SKIP are the number of SKIP table entries and the skip tables themselves, respectively.

If the -terminals_list control is not in effect, only the declaration of DPDA_n, SKIP_n, DPDA and SKIP are generated.

The DPS 6 object file produced by the -ada_sil control argument is equivalent to the data described by the PL/I declarations below. When a separate semantics format source segment is used, the object file also contains a transfer vector with the external name SEMVEC. The rule number, or production number if the -production control is in effect, must be passed to the transfer vector by value in register R1. The transfer vector's code destroys registers R1 and B4, all other registers are unchanged.

```
dcl  TL_length fixed binary (15) internal static
     options (constant) initial (xx);
dcl  TC_length fixed binary (15) internal static
     options (constant) initial (xx);

dcl 1 Terminal aligned based,
    2 position fixed binary (15),
    2 length fixed binary (15),
    2 code fixed binary (15);

dcl 1 TL (xx) aligned like Terminal external static;
dcl  TC char (xx) external static init ("xxx ... ");

dcl  DPDA_length fixed binary (15) internal static
     options (constant) initial (xx);
dcl  SKIP_length fixed binary (15) internal static
     options (constant) initial (xx);
dcl  DPDAv1 (xx) fixed binary (15) external static
     initial (xx, xx, ... );
dcl  DPDAv2 (xx) fixed binary (15) external static
     initial (xx, xx, ... );
dcl  SKIPv1 (xx) fixed binary (15) external static
     initial (xx, xx, ... );
dcl  SKIPv2 (xx) fixed binary (15) external static
     initial (xx, xx, ... );
```

All of the above external static variables are generated as "code section" constants to allow them to be shared constants. Because of this, this object file must be linked (with a LINKN linker directive) before the object file for any Ada/SIL compilation unit using the generated package specification.

As used in the above declarations, TL_length is the number of terminals (including complicated terminals) and TC_length is the length of the TC variable. The based variable Terminal describes a single entry in the terminal list array TL. The i-th terminal is substring (TC, TL.position (i), TL.length (i)). If the grammar uses synonyms, TL.code (i) gives the code for the i-th terminal. Otherwise, the code component is omitted from the Terminal structure and the code for the i-th terminal is i. DPDA_length and SKIP_length specify the number of entries in the DPDA and SKIP tables, respectively. DPDAv1 and DPDAv2 are the two columns of the DPDA. Similarly, SKIPv1 and SKIPv2 are the two columns of the SKIP tables.

If the -terminals_list control is not in effect, TL_length, TC_length, Terminal, TL, and TC are not generated.

Names: list_dpda

The list_dpda command produces a listing of the DPDA extracted from the result file of a previous LALR generation. This listing is formatted in the same manner as that produced by the -list control argument of the lalr command described above.

Usage: list dpda result file path {ctl args}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the DPDA is to be extracted. If result_file_path does not have a suffix of grammar, one is assumed. However, the suffix grammar must be the last component of the name of the result segment to be used. This argument may be an archive component pathname.

2) ctl_args may be the following optional argument.

-line_length N, -ll N
: causes the listing to be prepared with lines no longer that N characters. If this control argument is not specified, a line length of 136 characters is assumed.

-page_length N, -pl N
: prints the listing so that no more than N lines are on a page. If this control argument is not specified, a page length of 60 lines is assumed.

-dpda_xref, -xref, -dx
: includes cross reference lists of states, terminals, and variables in the listing of the DPDA. If the source segment was in the separate semantics format, the semantic actions are also cross referenced. In

the first two of these lists, each referencing state number is immediately followed by the letter "R", "L", "A", "B", or "D" indicating a read transition, look transition, transition from an apply state, or a look back reference by an apply state, or a look back reference implied by the default transition of an apply state, respectively. In the lists for variables and semantic actions, each state number is immediately followed by the letter "S", "T", or "U", indicating an apply single, apply with look back table, or apply using shared look back table, respectively.

-no_dpda_xref, -no_xref, -ndx
: does not include any DPDA cross reference lists in the listing of the DPDA. This is the default.

Notes:

If the result file used is named X.grammar, the listing produced will be placed in a segment named X.o.list in the working directory.

Names: plist_dpda

The plist_dpda command produces a listing of the DPDA extracted from the result file of a previous LALR generation. The listing is presented in the notation of Dijkstra [55].

Usage: plist_dpda result_file_path {ctl_args}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the DPDA is to be extracted. If result_file_path does not have a suffix of grammar, one is assumed. However, the suffix grammar must be the last component of the name of the result segment to be used. This argument may be an archive component pathname.

2) ctl_args may be any of the following optional arguments.

-line_length N, -ll N
: causes the listing to be prepared with lines no longer that N characters. If this control argument is not given, a line length of 136 characters is assumed.

Notes:

If the result file used is named X.grammar, the listing produced will be placed in a segment named X.p.list in the working directory.

Names: lalr$rev

The lalr$rev command prints the revision numbers of the major components of LALR on the user_output I/O switch.

Usage: lalr$rev

Names: print_parser_info, ppi

The print_parser_info command prints selected items of information for the specified result segment.

Usage: print_parser_info result_file_path {ctl_args}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the information is to be taken. If result_file_path does not have a suffix of grammar, one is assumed. However, the suffix grammar must be the last component of the name of the result segment used. This argument may be an archive component pathname.

2) ctl_args may be any of the following optional arguments.

   -header, -he prints the header. This is the default.

   -no_header suppresses printing of the header.

Names: make_dpda, md

The make_dpda command produces a table containing the DPDA extracted from the result file of a previous LALR generation. This table is the same as the one produced by the lalr command when it is invoked with the -table control argument.

Usage: make_dpda result_file_path {table_path} {ctl_args}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the DPDA is to be extracted. If result_file_path does not have a suffix of grammar, one is assumed. However, the suffix grammar must be the last component of the name of the result segment to be used. This argument may be an archive component pathname.

2) table_path is the pathname of the table to be produced. If this argument is given with the suffix incl.pl1, the suffix is ignored. Any other suffix is retained as given. If this argument is omitted, the entryname (or component name in case of an archive component pathname) portion of the first argument with the suffix grammar and the preceding "." replaced with "_t" is used.

3) ctl_args may be one or more of the following optional arguments. As used below X is the name given, or assumed, for the table.

-terminals_list, -tl
: include the terminals list in the table.

-terminals_hash_list, -thl
: include the terminals list and terminals hash list in the table.

-production_names, -pn
: include the production names (table) in the table. This control argument implies the -variables_list control argument described below.

-variables_list, -vl
: include the variables list in the table.

-synonyms, -syn
: include the terminal encoding as a field in the terminals list instead of using the index to the terminals list as the encoded value. This options is forced if the grammar contained a -synonyms control. The -synonyms control argument is meaningless unless the -terminals_list control argument is also specified.

-no_terminals_list, -ntl
: include neither the terminals list nor terminals hash list in the table. This is the default.

-no_terminals_hash_list, -nthl
: does not include the terminals hash list in the that. This is the default. Note that -terminals_hash_list -no_terminals_hash_list has the same effect as -terminals_list.

-no_production_names, -npn
: does not include the production names in the table. This is the default. Note that -production_names -no_production_names has the same effect as -variables_list.

-no_variables_list, -nvl
: include neither the production names nor the variables list in the table. This is the default.

-no_alm
: does not produce the table in the form described below for the -alm control argument.

-no_gmap
: does not produce the table in the form described below for the -gmap control argument.

-no_asm
: does not produce the table in the form described below for the -asm control argument.

-no_ada_sil
: does not produce the table in the form described below for the -ada_sil control argument.

-alm
: produce the table as an alm segment named X.alm and a Multics PL/I include file named X.incl.pl1.

-gmap
: produce the table as a gmap segment named X.gmap and a GCOS III PL/I include file named X.incl.pl1.

-asm
: produce the table as a DPS 6 (or Level 6) Multics Host Resident System object file named X.object and produce a DPS 6 Assembly Language include file named X.incl.nml.

-ada_sil
: produce the table as a DPS 6 (or Level 6) Multics Host Resident System object file named X.object and produce a DPS 6 Ada/SIL package specification file named X.spec.ada.

Notes: Options -alm, -gmap and -asm or -ada_sil may occur together. (Options -asm and -ada_sil are mutually exclusive.) If none of the control arguments -alm, -gmap, -asm or -ada_sil are present, the table is produced as a Multics object segment named X and a Multics PL/I include file name X.incl.pl1.

The -terminals_hash_list control argument is treated as if it were the -terminals_list control argument when producing a DPS 6 (Level 6) object file. The -synonyms control argument is meaningless when producing a DPS 6 object file with the -asm control argument. The -production_names and -variables_list control arguments are ignored when producing a DPS 6 object file. The DPS 6 object file is produced in LAF mode.

Name: l6_dpda

The l6_dpda command produces a DPS 6 Multics Host Resident System object file containing the DPDA extracted from the result file of a previous LALR generation. This object file is the same as the one produced by the lalr command when it is invoked with the -table control argument and either the -asm or -ada_sil control argument.

Usage: l6_dpda result_file_path {object_file_path} {ctl_args}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the DPDA is to be extracted. If result_file_path does not have a suffix of grammar, one is assumed. However, the suffix grammar must be the last component of the name of the result segment to be used. This argument may be an archive component pathname.

2) object_file_path is the pathname of the object file to be produced. If object_file_path does not have a suffix of object, one is assumed. If this argument is omitted, the object file is placed in the working directory with an entryname obtained by changing the result suffix of the first argument's entryname (or component name in case of an archive component pathname) to object.

3) ctl_args may be one or more of the following optional arguments.

-terminals_list, -tl
include the terminals list in the object file.

-synonyms, -syn
include the terminal encoding as a field in the terminals list instead of using the index to the terminals list as the encoded value. This options is forced if the grammar contained a -synonyms control. The -synonyms control argument is meaningless unless the -terminals_list control argument is also specified.

-no_terminals_list, -ntl
: does not include the terminals list (TL and TC) in the table. This is the default.

-no_asm
: does not produce the table in the form described below for the -asm control argument.

-no_ada_sil
: does not produce the table in the form described below for the -ada_sil control argument.

-saf
: produce the object file in SAF mode.

-laf
: produce the object file in LAF mode.

-slic
: produce the object file in SLIC mode.

-asm
: produce a DPS 6 (or Level 6) Assembly Language include file describing the external variables defined in the object file. This include file is stored in the same directory as the object file. Its entryname is obtained by changed the object suffix of the object file to incl.nml.

-ada_sil
: produce an Ada/SIL package specification describing the external variables defined in the object file. This package specification is stored in the same directory as the object file. Its entryname is obtained by changing the object suffix of the object file to spec.ada.

Notes: If none of the control arguments -saf, -laf, or -slic are present, the object file is produced in LAF mode. The -saf, -laf, and -slic control argument are mutually exclusive.

The control arguments -asm and -ada_sil are mutually exclusive. If neither is specified, -asm is assumed.

Names: kwsl

The kwsl command produces a Multics PL/I include file that declares an array containing a sorted list of terminal symbols and their encoded values. One or more synonyms may be specified for each terminal symbol.

Usage: kwsl result_file_path {synonyms_path {output_path {structure_name}}}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the encoded values for the terminals are to be taken. If result_file_path is given without the suffix grammar, it is assumed. This argument may be an archive component pathname.

2) synonyms_path is the pathname of an unstructured file naming the terminal symbols and synonyms to be included in the output. Each line of this file begins with the name of a terminal symbol in the first position of the line. The terminal symbol may optionally be followed by a list of synonyms for it. The synonyms are separated from the terminal symbol and from each another by single occurrences of the horizontal tab (HT) character.

   -all (or -a) may be specified instead of synonyms_path. In this case all of the terminals and synonyms defined by the grammar are include in the output.

   If synonyms_path is given without the suffix syn, it is assumed. This argument may be an archive component pathname. If this argument is not given, the first argument (result_file_path) with the suffix grammar changed to syn is used if such a segment exists otherwise -all is assumed.

3) output_path is the pathname of the include file to be produced. If output_path is given without the suffix incl.pl1, it is assumed. This argument may not be an archive component pathname. If this argument is not given, the entryname portion (or component name portion in case of an archive component pathname) of the first argument (result_file_path) with the suffix grammar changed to incl.pl1 is used.

4) structure_name is the name to be used for the level 1 structure in the output include file. If this argument is omitted, the structure will be named keyword.

Names: cobol_kwsl

The cobol_kwsl command produces a COBOL copy file that describes a table containing a sorted list of terminal symbols and a second table containing their encoded values. One or more synonyms may be specified for each terminal symbol.

Usage: cobol_kwsl result_file_path {synonyms_path {output_path}} {-ctl_args}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the encoded values for the terminals are to be taken. If result_file_path is given without the suffix grammar, it is assumed. This argument may be an archive component pathname.

2) synonyms_path is the pathname of an unstructured file naming the terminal symbols and synonyms to be included in the output. Each line of this file begins with the name of a terminal symbol in the first position of the line. The terminal symbol may optionally be followed by a list of synonyms for it. The synonyms are separated from the terminal symbol and from each another by single occurrences of the horizontal tab (HT) character.

-all (or -a) may be specified instead of synonyms_path. In this case all of the terminals and synonyms defined by the grammar are included in the output.

If synonyms_path is given without the suffix syn, it is assumed. This argument may be an archive component pathname. If this argument is not given, the first argument (result_file_path) with the suffix grammar changed to syn is used if such a segment exists otherwise -all is assumed.

3) output_path is the pathname of the copy file to be produced. If output_path is given without the suffix incl.cobol, it is assumed. This argument may not be an archive component pathname. If this argument is not given, the entryname portion (or component name portion in case of an archive component pathname) of the first argument (result_file_path) with the suffix grammar and the period preceding it changed to "-kwsl.incl.cobol" is used.

4) ctl_args may be one or more of the following optional arguments.

-bcd translates any lower case letters in the terminal symbols to upper case letters, sorts the terminal symbols according to the BCD collating sequence, and issues an error message for any terminal symbol containing a "'", "{", "|", "}", or " " (\140, \173, \174, \175, or \176).

-usage X describes the encoded value of the terminal symbols with usage X. The default usage is COMP-1.

-noe X names the elementary 01-level item giving the number of elements in the tables X. The default name for this item is SCAN-TABLE-1-NOE.

-loe X names the elementary 01-level item giving the length of the elements in the terminal symbol table X. The default name for this item is SCAN-TABLE-1-LOE.

-table1 X, -t1 X names the original definition of the terminal symbol table X. This item describes a record consisting of a series of 03-level FILLER items with VALUE clauses specifying the terminal symbols. The default name for this record is SCAN-TABLE-1.

-redef1 X, -r1 X names the redefinition of the terminal symbol table X. This item describes a record consisting of a single 03-level item with an occurs clause. The default name for this item is S-T-1.

-keyword X, -kw X
names the 03-level item in the redefinition of the terminal symbols table X. The default name of this item is KW.

-table2 X, -t2 X names the original definition of the encoded value table X. This item describes a record consisting of a series of 03-level FILLER items with values clauses specifying the encoded value of the terminal symbols. The default name for this item is SCAN-TABLE-2.

-redef2 X, -r2 X names the redefinition of the encoded value table X. This item describes a record consisting of a single 03-level item with an occurs clause. The default name for this record is S-T-2.

-keyvalue X, -kv X
names the 03-level item in the redefinition of the encoded value table X. The default name for this item is KV.

Notes:

If m terminal symbols and synonyms are given and the longest of them is n characters long, and assuming no control arguments are used, the following record descriptions will be produced:

```
01 SCAN-TABLE-1-NOE COMP-1 VALUE m.
01 SCAN-TABLE-1-LOE COMP-1 VALUE n.
01 SCAN-TABLE-1.
    03 FILLER PIC X(m) VALUE "..."
           .
           .
           .
01 S-T-1 REDEFINES SCAN-TABLE-1.
    03 KW PIC X(m) OCCURS n TIMES.
01 SCAN-TABLE-2.
    03 FILLER COMP-1 VALUE ... .
           .
           .
           .
01 S-T-2 REDEFINES SCAN-TABLE-2.
   03  KV COMP-1 OCCURS n TIMES
```

Names: cobol_dpda

The cobol_dpda command produces a COBOL copy file that describes various tables containing the DPDA extracted from the result file of a previous LALR generation.

Usage: cobol_dpda result_file_path {output_path} {ctl_args}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the DPDA is to be extracted. If result_file_path is given without the suffix grammar, it is assumed. This argument may be an archive component pathname.

2) output_path is the pathname of the copy file to be produced. If output_path is given without the suffix incl.cobol, it is assumed. This argument may not be an archive component pathname. If this argument is not given, the entryname portion (or component name portion in case of an archive component pathname) of the first argument (result_file_path) with the suffix grammar changed to incl.cobol is used.

3) ctl_args may be the following optional argument.

-usage X describes the entries in the DPDA tables with usage X.

Notes:

If the DPDA contains m entries and the SKIP table contains n entries, and assuming the usage control argument is not used, the following record descriptions will be produced:

```
*
*
01 DPDA-NOE COMP-1 VALUE m.
01 DPDA-V1-VALS.
    03 FILLER COMP-1 VALUE ... .
          .
          .
          .
01 DPDA-V1-REDF REDEFINES DPDA-V1-VALS.
    03 DPDA-V1 COMP-1 OCCURS m TIMES.
01 DPDA-V2-VALS.
    03 FILLER COMP-1 VALUE ... .
          .
          .
          .
01 DPDA-V2-REDF REDEFINES DPDA-V2-VALS.
    03 DPDA-V2 COMP-1 OCCURS m TIMES.
*
*
01 SKIP-NOE COMP-1 VALUE n.
01 SKIP-V1-VALS.
    03 FILLER COMP-1 VALUE ... .
          .
          .
          .
01 SKIP-V1-REDF REDEFINES SKIP-V1-VALS.
    03 SKIP-V1 COMP-1 OCCURS n TIMES.
01 SKIP-V2-VALS.
    03 FILLER COMP-1 VALUE ... .
          .
          .
          .
01 SKIP-V2-REDF REDEFINES SKIP-V2-VALS.
    03 SKIP-V2 COMP-1 OCCURS n TIMES.
```

Names: make_DPDA_dcl, mdd

The make_DPDA_dcl command produces a Multics PL/I include file containing the DPDA extracted from the result file of a previous LALR generation. This include file declares the DPDA as an internal static constant structure.

Usage: make_DPDA_dcl result_file_path {output_path}

1) result_file_path is the pathname of the result file from a previous LALR generation from which the DPDA is to be extracted. If result_file_path is given without the suffix grammar, it is assumed. This argument may be an archive component pathname.

2) output_path is the pathname of the include file to be produced. If output_path is given without the suffix incl.pl1, it is assumed. This argument may not be an archive component pathname. If this argument is not given, the entryname portion (or component name portion in case of an archive component pathname) of the first argument (result_file_path) with the suffix grammar changed to incl.pl1 is used.

Names: print_parse_info, ppi

The print_parser_info command prints selected items of information for the specified result segment.

Usage: print_parser_info result_file_path {ctl_args}

1) result_file_path is the pathname of the result file. If result_file_path is given without the suffix grammar, it is assumed. This argument may be an archive component pathname.

2) ctl_args may be one or more of the following optional arguments.

-header, -he prints the header. This is the default.

-no_header does not print the header.

-long, -lg prints more information when the header is printed. Additional information includes a listing of source files used to generate the result file. The severity is also printed if it is nonzero.

-short, -sh does not print the extra information described above for the -long control argument. This is the default.

Names: lalr_terms

The lalr_terms command prints the terminal symbols contained in the result file produced when a grammar was previously translated. The encoded value of each terminal symbol is also printed.

Usage: lalr_terms result_file_path

1) result_file_path is the pathname of the result file from which the terminal symbols and their encoded values are to be obtained. If result_file_path is given without the suffix grammar, it is assumed. This argument may be an archive component pathname.

Names: DPDA_sizes, DPDAsizes

The DPDA_sizes command prints a list giving the sizes of the various types of tables comprising the DPDA for a grammar. For each size of read table (1 entry, 2 entries, 3 entries, etc), the total number of read tables of that size, the percentage of total read table storage occupied by read tables of that size, and the percentage of total read table storage occupied by read tables of that size and smaller sizes is listed. The same statistics are also given for look tables. In addition, the number of tables of each type is given.

Usage: DPDA_sizes result_file_path

1) result_file_path is the pathname of the result file from a previous LALR generation containing the DPDA to be examined. If result_file_path is given without the suffix grammar, it is assumed. This argument may be an archive component pathname.

Names: lalr_parse, lalrp, lrk_parse, lrkp

The lalr_parse command provides a means for testing an lalr produced parser table. This program is an adequate parser in many applications.

Usage: lalr_parse path {source} {ctl_args}

1) path is the pathname of the result segment generated when the grammar was processed. If the path does not have a suffix of grammar, one is assumed. However, the suffix grammar must be the last component of the name of the result segment to be used. This argument may be an archive component pathname.

2) source is the pathname of a source segment to be parsed. If not supplied, lines will be read from user_input. This is true of the default scanner (see below). If a user scanner is supplied, it must provide for reading user_input if no source is specified, or it must report an error. This argument may be an archive component pathname.

3) ctl_arg may be one or more of the following optional arguments. NOTE: P represents a pathname, other than an archive component pathname, with an entryname portion of E; if P is given as a simple name, it is found according to the search rules.

-local_reads N, -lr N
: requires the parser's local recovery facility to accept N symbols beyond the bad symbol in order for a particular recovery to be considered successful. N must be in the range 1 to 9. The default is 2.

-max_recover N, -mr N
: allows the parser to perform at most N local recoveries (see page 17) in succession. If N is zero, local recovery is disabled. The default is 1.

-no_print, -npr
: does not print the source as it is scanned. This is the default.

-no_recovery, -nr
: is the same as -max_recover 0.

-no_semantics, -nsem
: disables calls to the semantics actions. This is the default.

-no_trace
: does not trace the execution of the parsing and error recovery procedures. This is the default.

-print, -pr
: causes each line from source to be printed (with line numbers) as it is scanned. This is true of the default scanner. If a user scanner is supplied, it determines whether or not printing is available.

-recovery
: is the same as -max_recover 1.

-scanner P, -scan P
: is the pathname of a scanner procedure which corresponds to the grammar. The scanner procedure must have entry points named E and init. The default scanner is explained below.

-semantics {P}, -sem {P}
: enables calls to the semantic actions. If the grammar's source is in the embedded semantics format, the pathname of a semantics segment which corresponds to the grammar must be given by P and this segment must have an entry point named E. It is this entry point which is called to perform a semantic action. If the grammar's source is in the separate semantics format, P may be given; however, its only use is to specify an initialization entry point as discussed in the interface description below. (In the separate semantics format the result segment contains the names of any semantic routines to be called.)

-trace
: causes a trace of the parsing and error recovery procedures to be printed.

-skip_depth N, -sd N
: specifies that skip recovery (see page 18) shall not make more than N attempts, each from deeper in the parse stack, to recover after discarding a particular skip symbol.

-skip_reads N, -sr N
: specifies that skip recovery (see page 18) must be able to accept the next N input symbols following a skip symbol in order to recover following the skip symbol. If fewer than N symbols can be accepted, skipping continues until another skip symbol is found. N must be in the range 1 to 9.

Scanner/Semantics

lalr_parse supplies a scanner procedure and a semantics procedure. The user can supply his own. This is how these procedures are used. User routines must have these interfaces.

1) The semantics routine is called each time action is required. The supplied semantics routine does nothing. (It is used to disable calls to the semantic actions.)

If the DPDA was generated without use of the -production control (see the lalr command described beginning on page 20), the following interface is used:

Usage:

```
dcl E entry (fixed bin, fixed bin, ptr, fixed bin);
call E (rule_no, alt_no, lex_stack_ptr, ls_top);
```

rule_no is the number of the rule completed
alt_no is the number of the alternative which was used
lex_stack_ptr
is a pointer to the lexical stack.
ls_top is the location in the lexical stack corresponding to the rightmost rule alternative symbol.

If the DPDA was generated with use of the -production control, the following interface is used:

Usage:

```
dcl E entry (fixed bin, ptr, fixed bin);
call E (prod_no, lex_stack_ptr, ls_top);
```

prod_no is the number of the production which was used
lex_stack_ptr
is a pointer to the lexical stack.
ls_top is the location in the lexical stack corresponding to the rightmost production symbol.

2) The semantics routine may also contain an initialization entry point. If it does contain an initialization entry point, it is called once before the parse begins.

Usage:

```
dcl E$init entry;
call E;
```

3) The scanner contains an initialization entry point. It is called once, to begin the parse. It allows the scanner to get the input information and to do any initialization necessary.

Usage:

```
dcl E$init entry (ptr, fixed bin (21), bit (1), ptr, char (100) varying);
call E$init (input, leng, prsw, result_ptr, opt);
```

input is a pointer to the source segment if leng is non-zero. Otherwise, it points to an empty temporary segment. If the user choses to read from user_input when source is not supplied, he should append each line read to this segment (values in the lex_stack may reference more than the current line).

prsw is "1"b if the -print option was specified, otherwise it is "0"b.

leng is the length in bytes of the source segment or is zero if source was not specified.

result_ptr
is a pointer to the input result segment. This segment contains, among other things, the grammar's terminal list and the corresponding terminal codes.

opt contains a list of control arguments given in the lalr_parse command line.

4) The scanner also contains a get-next-symbol entry. It is called each time another symbol is needed. It must return an encoding of zero when end-of-information (EOI) is reached.

Usage:

```
dcl E$E entry (ptr, fixed bin);
call E$E (stkp, putl);
```

stkp is a pointer to the lexical stack. The stack declaration is in lalr_stk.incl.pl1. It specifies that the stack is based on a variable named "stkp".

putl is the location in the stack to put the symbol information.

The scanner must set these fields:

stk.symptr (putl) points to the beginning of the found symbol.

stk.symlen (putl) length in bytes of the symbol found (may be zero).

stk.file (putl) the include file number of the segment containing the symbol. The source segment is include file number zero, the first include file requested is include file number one, the second include file requested is include file number two, etc.

stk.line (putl) line number where symbol begins. The symbol is assumed to be contained entirely within a single include file.

stk.symbol (putl) encoding for the symbol found.

These fields may be set:

stk.ptr1 (putl) pointer to user data

stk.ptr2 (putl) pointer to user data

5) The scanner may also contain a termination entry point. If it does contain a termination entry point, it is called once at the end of the parse.

Usage:

```
dcl E$finish entry;
call E$finish;
```

The default scanner algorithm is this:

1. During initialization, the terminals are separated into 2 lists. One list contains all the terminals that consist only of alphanumeric characters. The other contains all the rest, sorted by decreasing length.

However, the special terminals <string>, <integer>, <fixed-point literal>, <floating-point literal>, <symbol>, and <EOL> are looked for. These are built in complicated terminals.

2. At get-next-symbol time, if an alphanumeric string exists, then it is taken as a single token. This token is compared against the list of alphanumeric terminals in the grammar. If one is found, that encoding value is returned. The fact that the whole alphanumeric string is compared against the terminal list means, for example, that a label "dclnam" will not be mistakenly taken as the terminal "dcl".

If no terminal in the list matches, then if the token is all numeric characters and at least one of the terminals <integer>, <fixed-point literal>, or <floating-point literal> exists in the grammar, the token is extended as necessary if it contains a decimal point and one of these complicated terminals is returned. These complicated terminals are defined by the following grammar.

```
<floating-point literal ::=
   <decimal number>e<exponent> !
<fixed-point literal> ::=
   <decimal number>f<exponent> !
<integer> ::=
   <decimal integer> !
<decimal number> ::=
   <decimal integer>.<decimal integer> |
   <decimal integer>. |
   .<decimal integer> !
<exponent> ::=
   -<decimal integer> |
   +<decimal integer> |
   <decimal integer> !
<decimal integer> ::=
   <decimal integer><digit> |
   <digit> !
<digit> ::=
   0|1|2|3|4|5|6|7|8|9 !
```

If a token conforming to the syntax of <decimal number> is found but it is not followed by an "e" or "f", it is considered a <fixed-point literal> if it exists in the grammar. If <fixed-point literal> does not exist in the grammar but <floating-point literal> does, the <decimal number> is considered a <floating-point literal>. If neither <fixed-point literal> nor <floating-point literal> exists in the grammar, the <decimal number> is considered to be two <integer>s separated by a dot.

If a token conforming to the syntax of <fixed-point literal> is found but <fixed-point literal> is not a terminal of the grammar, the token is tentatively split into two tokens, a <decimal number> followed by some token beginning with the letter "f". If <floating-point literal> is a terminal of the grammar, the <decimal number> is considered a <floating-point literal> otherwise it is considered two <integer>s separated by a dot.

If a token conforming to the syntax of <floating-point literal> is found but <floating-point literal> is not a terminal of the grammar, the token is tentatively split into two tokens, a <decimal number> followed by some token beginning with the letter "e". If <fixed-point literal> is a terminal of the grammar, the <decimal number> is considered a <fixed-point literal> otherwise it is considered two <integer>s separated by a dot.

If none of the above apply and the terminal "<symbol>" exists in the grammar, this encoding is returned.

If an alphanumeric string is not present in the input, then if the first character is a " and the terminal <string> is present in the grammar, a PL/I style string is scanned off and the proper encoding is returned. Otherwise, the second list of terminals is searched, taking the length of each terminal to determine the amount of input to look at. If a match is found, the encoding for it is returned. Remember that this list is ordered by decreasing length. This method of comparison means, for example, that if both ">=" and ">" are terminals, the first will always be found if it exists in the input.

If neither of the lists contained a match at this point in the input, the scanner moves ahead one character. If the character skipped is NL (\012) and the terminal <EOL> exists in the grammar, this encoding is returned. Otherwise, the scanner tries again. In this case, if the character skipped is not greater than SP (\040), it is dropped without comment.

stk.symptr (putl) is always set to point to the first character of the symbol which satisfied the scan. If <symbol>, <integer>, <fixed-point literal>, <floating-point literal>, or <string> is processed, stk.symlen (putl) is set to the length of the input string which was used; otherwise stk.symlen (putl) is set to zero.

EOI is returned when the end of an input segment is reached, or when a line is read from user_input consisting of "EOI" only.

Parser macro

The LALR system has available a macro which can generate a skeleton parser. Once this parser is obtained, it may be tailored to the individual application. The tailoring actually begins during the generation, at which time many options are available to dictate what will be obtained. This "macro" is processed by runoff.

Figure 1 shows what a parse procedure generally looks like. However, it fleshes out quite a bit when you add things like look ahead processing, error recovery of one or two kinds, and error reporting. The macro helps in this process. To generate a parser, you must create a segment X.runoff. It has this form:

```
   .if lalr_skel
 [ .sr XXX YYY ]
   ...
   .if lalr_skel
```

The first call to lalr_skel sets the default values in some variables. Then you adjust any of these values you wish. The second call to lalr_skel generates the parser, directed by values in the variables.

If the segment is named X.runoff the output segment will be named X.incl.pl1 and the parse procedure therein will be named X.

Following are the variables which control the generation; they show the variable name and the default value. Remember that in quoted strings runoff requires:

```
" be entered as *"
* be entered as **
```

```
initialize
do while (^EOI);
     if READ_state then do; /* includes lookahead 1 */
          if lookahead stack empty then
               call scanner; /* puts to lookahead stack */
          look in read-table for first lookahead symbol
          if not found then
               if there is a default look transition then
                    set next state from it
               else if there is a table continuation then
                    change to continuation table
                         and repeat the above search
               else ERROR
          else do;
               if not lookahead transition then
                    remove symbol from lookahead stack
                           and push it onto lex stack
                    and push state number onto parse stack
                set next state from read-table
          end;
     end;
     else if MLOOK_state then do; /* look ahead n */
          do until n symbols in lookahead stack;
               call scanner; /* put to lookahead stack */
          end;
          look in look-table for n'th lookahead symbol
          if not found then
               if there is a default transition then
                    set next state from it
               else if there is a table continuation then
                    change to continuation table
                         and repeat the above search
               else ERROR
          else set next state from look-table
     end;
     else if APPLY_state then do;
          call semantics
          delete necessary symbols from lex stack
          delete necessary states from parse stack
          if empty production then
               push state number onto parse stack
          and push "empty" onto lex stack
          look in apply-table for top stacked state
          set next state from apply-table
     end;
end;
```

Figure 1. Generalized parse procedure.

```
.sr parameters ""
```

The value of this variable is any parameters wanted on the parse procedure. Example: "sptr, slen"

```
.sr code ""
.sr standard_codes %true%
```

These control the reporting of events which cause the parser to prematurely terminate. If "code" is "" such events are reported by signals. If it is not "", it is the name of a parameter or variable which is assigned a non-zero return status code to report such events. The events causing premature termination are: parse_error indicating a recovery failure; logic_error which is caused by an invalid DPDA; and stack_overflow indicating overflow of the parse, lexical, or look ahead stacks. If "code" is not "" and "standard_codes" is %true%, standard status codes from lalr_error_table_ are used and "code" is declared as a fixed bin (35) parameter to the parser. (In this case "code" must be named in "parameters" described above.) If "code" is "" or "standard_codes" is %false%, these conditions or constants are declared before the parser's procedure statement.

```
.sr print_recov_msg "call print_recov_msg"
.sr print_recov_msg_incl ""
.sr gen_print_recov_msg %true%
.sr message_prefix "ERROR"
.sr severity_length "**"
.sr unique_local_recovery_severity ""
.sr ambiguous_local_recovery_severity ""
.sr local_recovery_severity ""
.sr skip_recovery_severity ""
.sr syntax_error_severity
.sr stack_overflow_severity ""
.sr logic_error_severity ""
```

These specify things about printing error recovery messages. "print_recov_msg" is a statement or statements to be used to print the error recovery message. The terminating semi-colon need not be included. At the time this statement is executed the variable recov_msg contains the text of the message. "print_recov_msg_incl" is the name of an include file (without the incl.pl1 suffix) which contains the procedure (or a declaration of an external procedure) to print error recovery messages. If this is specified, an %include statement will be generated inside the parser. If "print_recov_msg_incl" is "" and "gen_print_recov_msg" is %true% the parser macro generates a procedure to print the error recovery message on user_output. "message_prefix" is a character string that each error recovery message is to begin with. The "XXX_severity" variables specify the severity of the various types of errors that may occur. "severity_length" specifies the length of the character strings given for the "XXX_severity" variables or is an asterisk if they are not all the same length. The severity can be words (or phrases) such as "Warning" and "Fatal" or numbers such as "1", "2", and "3". In either case they are treated as character strings for the purpose of fabricating the error recovery message.

The following table shows the message formats resulting from various combinations of "message_prefix" and the "XXX_severity" values:

Prefix	Severity	Message
"XXX"	""	XXX on line ...
"XXX"	"YYY"	XXX YYY error on line ...
""	"YYY"	YYY error on line ...
""	""	Line ...

For example

```
.sr message_prefix "Severity"
.sr local_recovery_severity "2"
```

results in messages (for local recovery) of the form

```
Severity 2 error on line ...
```

```
.sr db_sw "db_sw"
.sr db_sw_param %true%
.sr db_sw_attr "internal static init (*"0*"b)"
.sr clear_residue %false%
```

These control options to aid in debugging a grammar and its semantics procedure. "ds_sw", "db_sw_param", and "db_sw_attr" control the inclusion of the trace coding and generation of the switch to control it. "db_sw" names the switch to control execution of the trace coding. If the value is "" no trace coding is included. If "db_sw_param" is %true%, "db_sw" is generated as a bit (1) parameter to the parser. If "db_sw_param" is %false%, "db_sw" is generated as a global variable with its declaration preceding the parser's procedure statement. In this case, "db_sw_attr" are attributes, in addition to bit (1), wanted on the switch. "clear_residue" controls generation of code to clear the lexical and parse stack entries as they are deleted. It also causes code to be generated (when %true%) to fill in the symbol, symptr, and symlen fields of the new top lexical stack entry after a production is applied so as to indicate the production variable's name. Use of the "clear_residue" option requires the PN (Production Names), VL (Variables List), and VC (Variables Characters) to be available in the parse tables. (See the -production_names and -variables_list control arguments of the lalr command described on page 20.)

```
.sr parse_tables_incl ""
```

This specifies the name of an include file (without the incl.pl1 suffix) containing declarations of the parse tables. If "parse_tables_incl" is not "" an %include statement will be generated to include the named include file in the parser, otherwise no %include is generated with the assumption that the tables will be declared in the parser's containing block. The parse tables are the TL (Terminals List), TC (Terminals Characters), and the DPDA. The PN (Production Names), VL (Variables List), and VC (Variables Characters) may also be included in the parse tables.

```
.sr mla 4
```

This specifies the maximum look ahead the parser is to handle. If "mla" is 1, code for multiple look ahead states is not generated. "mla" is also used to determine the size of the look ahead stack required. If K is specified for the maximum look ahead, then the required size of the look ahead stack is: K if no recovery is requested, K+N if skip recovery is requested but local recovery is not, K+M+R if local recovery is requested but skip recovery is not, or the greater of K+N or K+M+R if both recoveries are requested. See "skip_recovery" and "max_recover" below for further discussion of recovery mechanisms and the definition of the "local reads" value M and the "skip reads" value N. The value of R is one if "deferred_actions" (also described below) is false or two it it is true.

```
.sr check_la %true%
```

This controls generation of code to check for look ahead stack overflow. (The look ahead stack cannot overflow unless "mla" was specified too small.) The overflow checks can be eliminated by setting "check_la" to %false%.

```
.sr lex_stack_incl ""
.sr ls_name ""
.sr ls_attr "based"
```

These specify things about the lexical stack include file. "lex_stack_incl" is the name of the include file to be generated, without the incl.pl1 suffix. "ls_name" is the level 1 name of the structure generated. If "ls_name" is "", the value of "lex_stack_incl" is used as the level 1 name of the structure generated. If the value of "lex_stack_incl" is "" no include file is generated. "ls_attr" are the attributes wanted on the structure in the include file.

```
.sr lex_stack "lex_stack"
.sr lex_stack_ptr "lex_stack_ptr"
.sr ls_dim 50
.sr ls_top "ls_top"
.sr ls_dcl1 ""
.sr ls_dcl2 ""
.sr ls_dcl3 ""
.sr ls_dcl4 ""
.sr ls_dcl5 ""
.sr ls_dcl6 ""
.sr ls_dcl7 ""
.sr ls_dcl8 ""
.sr ls_dcl9 ""
```

These specify things about the lexical stack. "lex_stack" is the name of the lexical stack. "lex_stack_ptr" is the name of a variable to be declared as a pointer and initialized with the address of the lexical stack. If "lex_stack_ptr" is "", no such variable is declared. "ls_dim" is the size (dimension) of the lexical stack. (The parse stack is the same size.) "ls_top" is the name of the variable which tells where the top element currently is. The four fields required to be set by the scanner used by lalr_parse are always in the stack declaration. "ls_dcl1" thru "ls_dcl9" are a way of specifying additional entries needed in the stack. Do not include the level number or comma in the specification. Examples:

```
"value fixed bin (24)"
"(ptr1, ptr2) ptr"
```

```
.sr la_dim 0
```

This can be used to declare the look ahead stack (FIFO) larger than implied by the maximum look ahead value, "mla", described above. The lexical stack and parse stack are declared as

```
lex_stack (-la_dim:ls_dim)
parse_stack (ls_dim)
```

The look ahead stack is the negative elements of the lexical stack; therefore they have identical structure.

```
.sr reserved_kw %false%
.sr binary_lookup %true%
.sr binary_lookback %true%
```

These control the generation of symbol look up and state look back coding. If "reserved_kw" is true, the symbol look up is generated to handle only grammars with reserved keywords. If it is false, the generated symbol look up code can handle both reserved and unreserved keyword grammars. Generally, the coding for unreserved keywords is more time-consuming than that for reserved keywords. Reserved keyword coding will not work when a symbol has been marked (-mark option) for unreserved purposes. If "binary_lookup" is true, a binary search is used for symbol look up (if possible); otherwise, a serial search is used. If "binary_lookback" is true, a binary search is used for state look backs; otherwise, a serial search is used.

```
.sr optimized_looks %false%
.sr nonoptimize_looks %true%
```

These control the generation of code to handle DPDA's with and without optimized looks respectively. If both are %true%, code is generated to handle both types of DPDA's. No significant extra code is required to handle both types of DPDA's.

```
.sr scanner "scanner"
.sr sc_desc ""
.sr sc_args ""
.sr la_put_needed %true%
.sr sc_incl ""
.sr scanner_init ""
.sr scanner_init_desc ""
.sr scanner_init_args ""
.sr scanner_finish ""
.sr scanner_finish_desc ""
.sr scanner_finish_args ""
```

These specify things about the scanner procedure. "scanner" is the name of the scanner to be called. "sc_desc" is a parameter descriptor list (without enclosing parentheses) for the scanner procedure. The values "none" and "any" may be given for "sc_desc", or any of the other "..._desc" variables discussed below, to indicate an entry declaration with no parameter descriptor list or an entry declaration with the options (variable) attribute instead of a parameter descriptor list, respectively, is to be generated. If "sc_desc" is "", no entry declaration is generated for the scanner. "sc_args" are the arguments to be passed to the scanner procedure. Whenever the scanner is called, the variable lookahead_put contains the subscript value for the element of the look ahead stack to be filled by the scanner. If "la_put_needed" is true, a variable named la_put will also exist and will contain the value -lookahead_put whenever the scanner is called. "sc_incl" is the name of an include file (without the incl.pl1 suffix) which contains the scanner. If this is specified, an %include statement will be generated inside the parser. Then the lexical stack will be available without any include file or parameter passing necessary. "scanner_init" specifies an entry point (normally in the scanner procedure) to be called once before the first call to the scanner's "get next terminal" entry point. If "scanner_init" is "" no such call is generated. "scanner_init_desc" is a parameter descriptor list (without enclosing parentheses) for the "scanner_init" entry point. If "scanner_init_desc" is "", no entry declaration is generated for this entry point. "scanner_init_args" are the arguments to be passed to the "scanner_init" entry point. "scanner_finish" specifies an entry point (normally in the scanner procedure) to be called once after the last call to the scanner's "get next terminal" entry point. If "scanner_finish" is "" no such call is generated. "scanner_finish_desc" is a parameter descriptor list (without enclosing parentheses) for the "scanner_finish" entry point. If "scanner_finish_desc" is "", no entry declaration is generated for this entry point. "scanner_finish_args" are the arguments to be passed to the "scanner_finish" entry point.

```
.sr deferred_actions %false%
.sr semantics "semantics"
.sr sem_desc ""
.sr sem_args "rule_number, alternative_number"
.sr semantics_prod ""
.sr desc_prod ""
.sr sem_args_prod "production_number"
.sr sem_incl ""
.sr semantics_sw ""
.sr semantics_sw_param %true%
.sr semantics_init ""
.sr semantics_init_desc ""
.sr semantics_init_args ""
.sr semantics_finish ""
.st semantics_finish_desc ""
.sr semantics_finish_args ""
```

These specify things about the semantics procedure. If "deferred_actions" is %true%, calls to the semantics procedure are deferred until a read transition is about to be made, an empty production is about to be applied, or the final state is reached. This usually improves the behavior of both local and skip recovery. If neither local recovery nor skip recovery is being generated, "deferred_actions" is ignored. "semantics" is the name of the semantics procedure to be called when an apply is done using a DPDA generated without use of the -production control (see the lalr command described beginning on page 20). "sem_desc" is a parameter descriptor list (without enclosing parentheses) for the semantics procedure. If "sem_desc" is "", no entry declaration is generated for the semantics. "sem_args" are the arguments to be passed to the "semantics" procedure. When it is called the variables rule_number, alternative_number, and production_number are valid. The default is to pass the rule number and alternative number of the apply being done. "semantics_prod" is the name of the semantics procedure to be called when an apply is done using a DPDA generated with use of the -production control. "sem_desc_prod" is a parameter descriptor list (without enclosing parentheses) for the "semantics_prod" procedure. If "sem_desc_prod" is "", no entry declaration is generated for this procedure. "sem_args_prod" are the arguments to be passed to the "semantics_prod" procedure. When it is called the variable production_number is valid. The defaults generate a parser which does not support DPDA's generated with the -production control. "sem_incl" is the name of an include file (without the incl.pl1 suffix) which contains the semantics procedure. If this is specified, an %include statement will be generated inside the parser. "semantics_sw" and "semantics_sw_param" control the generation of a switch used to dynamically enable calls to the semantics procedure. They are used in the same manner as described above for "db_sw" and "db_sw_param". "semantics_init" specifies an entry point (normally in the semantics procedure) to be called once before the first call to the semantics' "take semantic action" entry point. If "semantics_init" is "" no such call is generated. "semantics_init_desc" is a parameter descriptor list (without enclosing parentheses) for this entry point. If "semantics_init_desc" is "", no entry declaration is generated for it. "semantics_init_args" are the arguments to be passed to the "semantics_init" entry point. "semantics_finish" specifies an entry point (normally in the semantics procedure) to be called once after the last call to the semantics' "take semantic action" entry point. If "semantics_finish" is "" no such call is generated. "semantics_finish_desc" is a parameter descriptor list (without enclosing parentheses) for the "semantics_finish" entry point. If "semantics_finish_desc" is "", no entry declaration is generated for this entry point. "semantics_finish_args" are the arguments to be passed to the "semantics_finish" entry point. NOTE: If the parse tables used are to be obtained from a separate semantics format source segment, X$semantics_vector must be specified for "semantics" and/or "semantics_prod", as appropriate. (X is the segment name of the parse tables.) Also, rule_number must be the first argument listed in "sem_args" and/or production_number must be the first argument listed in "sem_args_prod".

```
.sr skip_recover %true%
.sr skip_reads 1
.sr skip_reads_param %false%
.sr skip_depth 1
.sr skip_depth_param %false%
.sr skip_cleanup ""
```

These determine whether or not the skip recovery mechanism is included in the parser and, if so, how many succesive input symbols, following a skip symbol, must be recognized to terminate a skip and how deep in the parse stack skip recovery will go to find a state from which the parse can be resumed. "skip_recover" may be set %false% if not needed. "skip_reads" and "skip_depth" are meaningful only when "skip_recover" is %true%. When used, "skip_reads" must be a number in the range 1 to 9 inclusive or the name of a variable or parameter containing such a value; e.g. it could be set to "max_skip_reads". If "skip_reads_param" is %true%, "skip_reads" is generated as a fixed bin parameter to the parser. In this case it must be listed in the "parameters" variable described above and its default is changed to skip_reads. If "skip_reads_param" is %false%, no declaration is generated for "skip_reads". When used, "skip_depth" must must a number or the name of a variable or parameter containing a numeric value. If "skip_depth_param" is %true%, "skip_depth" is generated as a fixed bin parameter to the parser. In this case it must be listed in the "parameters" variable described above and its default is changed to skip_depth. If "skip_depth_param" is %false%, no declaration is generated for "skip_depth". In the earlier discussion of skip recovery, the values given by "skip_depth" and "skip_depth" were referred to as N and M respectively. "skip_cleanup", when not "", is one or or statements (with terminating semicolons) to be executed immediately before returning from the skip recovery procedure. Normally these statements are used to back up the lexical stack and the semantic actions' output to a consistent state.

```
.sr max_recover 0
.sr max_recover_param %true%
.sr local_reads 2
.sr local_reads_param %false%
.sr local_recover_sw ""
.sr local_recover_sw_param %true%
.sr make_symbol ""
.sr make_symbol_incl "make_symbol"
```

These control generation of local recovery code. "max_recover" is the upper limit on the number of local recoveries which can occur in a row. If it is zero, no local recovery coding will be generated. It may be a parameter to the parser, a variable declared in a block containing the parser, or a number. If "max_recover_param" is true and "max_recover" is not a number, "max_recover" is generated as a fixed bin parameter to the parser. In all other cases, no declaration of "max_recover" is generated. "local_reads" and "local_reads_param" are not meaningful when "max_recover" is 0. Loosely speaking, "local_reads" specifies the number of input symbols following the bad symbol that must be accepted for a particular local recovery to be considered successful. See the tables given under local recovery (page 17) for a precise definition. When used, "local_reads" must be a number in the range 1 to 9 inclusive or the name of a variable or parameter containing such a value; e.g. it could be set to "min_good_symbols". If "local_reads_param" is %true%, "local_reads" is generated as a fixed bin parameter to the parser. If "local_reads_param" is %false%, no declaration is generated for "local_reads". "local_recover_sw", when not "", causes a switch to enable the local recovery at run-time to be generated. "local_recover_sw" gives the name of this switch. If "local_recover_sw_param" is %true%, "local_recover_sw" is generated as a bit (1) parameter to the parser. In this case, "local_recover_sw" must be listed in the parameters variable describe above. If "local_recover_sw_param" is %false%, "local_recover_sw is generated as a global variable with its declaration preceding the parser's procedure statement. "make_symbol" is the name of a procedure to be called to complete the fabrication of a symbol by local recovery. When "make_symbol" is called, local recovery will have already placed the encoded value of the symbol being created in the symbol field of the lexical/lookahead stack entry and set the symlen field to zero. The symptr field will not have been altered. "make_symbol is called with two fixed bin paramaters if "deferred_actions" is false or three fixed bin parameters if it is true. For the sake of discussion, call these parameters i, j, and k. Then, i is the lexical/lookahead stack index for the symbol being created. If the symbol is being inserted, j and k will be equal to i. If the created symbol is replacing the bad symbol, j and k will both be the lexical/lookahead stack index of an entry containing the unaltered bad symbol. If the created symbol is replacing the previous symbol, j will be the lexical/lookahead stack index of an entry containing the unaltered previous symbol and k will be the lexical/lookahead stack index of the bad symbol. "make_symbol_incl" is the name of an include file (without the incl.pl1 suffix) which contains the "make_symbol" procedure. If "make_symbol_incl" is not "", an %include statement will be generated inside the local recovery procedure to include it. If "make_symbol" is "", "make_symbol_incl" is ignored.

After this macro source is prepared it is processed by executing

```
runoff X -in 0 -sm; dl X.runout
```

This will cause X.incl.pl1 and optionally xx.incl.pl1 (stack declaration) to be created.

Sample usage of LALR

This example demonstrates the implementation of an online interpreter of logical expressions.

With the text editor (e.g., ted) create a segment log.lalr as in Figure 2. Then execute

```
lalr log -source -symbols -term
```

to check it out. This is a useable grammar. Note on the 2nd line that a "|" is wanted in the language and so must be entered as "'|". On the 6th line, however, the "|" is the LALR "or" operator.

```
<log>      ::= <or> !
<or>       ::= <or> '| <and> !;
<or>       ::= <and> !
<and>      ::= <and> & <not> !;
<and>      ::= <not> !
<not>      ::= ^ <bit>  | <bit> !
<bit>      ::= X !;
<bit>      ::= ( <or> ) !
```

Figure 2. Basic log.lalr (grammar only)

At this point you could try out the language to see if it indeed describes what you think it should. If you execute

```
lalr_parse log -trace
```

it will type LALRP (6.0) and then wait for you to type a statement. If you reply something like:

```
^(X|X|(X&X&X))&X
```

you will see a trace of the parsing action. It will stop when it reaches the end of the line. You then reply

```
EOI
```

to signal end-of-input and the trace will complete.

The trace will be made up of things like

```
   56 APLY (-3 1 -4) sd = 1 (19)
 * 37 READ operator symbol "|"
```

The first number on the line is the state number; if preceded by a "*" it means it was stacked (on the parse stack). The numbers in the parentheses following APLY are the rule, alternative, and production numbers of the production being applied. If the DPDA was generated with use of the -production control (see the lalr command described beginning on page 20), only the production number will appear. If the rule number (and production number) is negative, no semantics exist for it. "sd = 1" means 1 element is deleted from the parse stack and 1 element is also to be deleted from the lexical stack. (The lexical and parse stacks always contain the same number of elements.) The list of numbers inside the second "()"'s tell the states which are deleted from the parse stack.

The 'operator symbol "|"' following the READ indicates that the symbol read was a vertical bar. (All terminal symbols, other than complicated terminals, that begin with a special character are called operator symbols. The READ can also be followed by the phrase 'reserved word "XXX"' or 'keyword "XXX"' or by the name of a complicated terminal followed by its representation in the input.

You decide you need your own parser; the skeleton of one can be generated with the macro. You decide that you need an entry in the lexical stack to hold the bit value of the result. You then create a macro input segment as in Figure 3, and then execute

```
rf log_parse_ -sm; dl log_parse_.runout
```

to get log_parse_.incl.pl1, your parse procedure.

```
.if lalr_skel
.sr ls_dcl1 "val bit(1)"
.if lalr_skel
```

Figure 3. Macro input, log_parse_.runoff

You then build the rest of your semantics procedure around the grammar that you know is acceptable to LALR. This gives a source which looks like Figure 4.

Now you run LALR again with

```
lalr log -source
```

This gives a listing file because of the -source option in the command call, and a semantics include file because of the -sem option in the source.

In the semantics include file, you will notice that the %%%%'s have been replaced with zero suppressed numbers, and since this is an incl.pl1 file all rules have been converted to PL/I comments. Figure 5 is this generated include file.

```
-sem log.incl.pl1
-parse

<log>      ::= <or> !
rule (%%%%):
     call ioa_ ("result is ^1b", lex_stack.val (ls_top));
     return;
<or>       ::= <or> '| <and> !;
rule (%%%%):
     lex_stack.val (ls_top-2) = lex_stack.val (ls_top-2)
                              | lex_stack.val (ls_top);
     return;

<or>       ::= <and> !
<and>      ::= <and> & <not> !;
rule (%%%%):
     lex_stack.val (ls_top-2) = lex_stack.val (ls_top-2)
                              & lex_stack.val (ls_top);
     return;

<and>      ::= <not> !
<not>      ::= ^ <bit>   | <bit> !
rule (%%%%):
    if atl_no = 1 then
     lex_stack.val (ls_top-1) = ^lex_stack.val (ls_top);
     return;

<bit>      ::= X !;
<bit>      ::= ( <or> ) !
rule (%%%%):
     lex_stack.val (ls_top-2) = lex_stack.val (ls_top-1);
     return;
```

Figure 4. Completed log.lrk

```
semantics: proc (rule_no, alt_no);

dcl (rule_no, alt_no) fixed bin parameter;

     goto rule (rule_no);

/* -sem log.incl.pl1
-parse */
/* <log>  ::= <or> ! */
rule (1):
     call ioa_ ("result is ^1b", lex_stack.val (ls_top));
     return;
/* <or>   ::= <or> '| <and> ! */;
rule (2):
     lex_stack.val (ls_top-2) = lex_stack.val (ls_top-2)
                              | lex_stack.val (ls_top);
     return;

/* <or>   ::= <and> ! */
/* <and>  ::= <and> & <not> ! */;
rule (4):
     lex_stack.val (ls_top-2) = lex_stack.val (ls_top-2)
                              & lex_stack.val (ls_top);
     return;

/* <and>  ::= <not> ! */
/* <not>  ::= ^ <bit>  | <bit> ! */
rule (6):
     if alt_no = 1 then
      lex_stack.val (ls_top-1) = ^lex_stack.val (ls_top);
      return;

/* <bit>  ::= X ! */;
/* <bit>  ::= ( <or> ) ! */
rule (8):
      lex_stack.val (ls_top-2) = lex_stack.val (ls_top-1);
      return;

end log;
```

Figure 5. log.incl.pl1

The listing file, Figure 6, does not show all of the source; only the rules. The line numbers are, however, correct. You will notice that some of the rules are double spaced and some are single spaced. There is a convention which allows you to control this. The character following the semantic separator, "!", is included in the listing. If this character is a NL, as in line 4 or 21, then an empty line will follow it. If this character is a ";" or a space, as in line 8 or 28, then there is no empty line following.

Notice that the alternative on line 22 uses the "|" form. This means that the alternative number must be used to determine what portion of the semantics to do.

The alternative on lines 15 and 21 use the multiple definition form. Since each of the definitions is a separate rule, the alternative number need not be checked (it is always 1).

```
            GENERATION LISTING OF SEGMENT log
            Processed by: Prange.SLANG.a using LALR 6.0
               of Friday, April 16, 1982
            Processed on: 04/16/82  1720.8 est Fri
                 Options: -source

   4        <log>     ::= <or> !

   8        <or>      ::= <or> '| <and> !;
  14        <or>      ::= <and> !

  15        <and>     ::= <and> & <not> !;
  21        <and>     ::= <not> !

  22        <not>     ::= ^ <bit>  | <bit> !

  28        <bit>     ::= X !;
  29        <bit>     ::= ( <or> )  !
```
Figure 6. logg.list

Non-LALR (k) Grammars Let us consider the arithmetic expression grammar shown in Figure 7. The sentence i+i*i has two distinct leftmost derivations:

```
<e> => <e> + <e>               <e> => <e> * <e>
    => i + <e>                     => <e> + <e> * <e>
    => i + <e> * <e>               => i + <e> * <e>
    => i + i * <e>                 => i + i * <e>
    => i + i * i                   => i + i * i
```

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence.

```
<e> ::= <e> + <e>
      | <e> * <e>
      | ( <e> )
      | - <e>
      | i !
```

Figure 7. Ambiguous e.lalr (grammar only)

LALR is unable to generate parsers for ambiguous grammars. When the grammar of Figure 7 is presented to LALR, it will be rejected. Three diagnostics will be written to the user_output I/O switch for this grammar. Each will be of the form:

```
WARNING: One or more configurations converged on the same next
set. This implies infinite look ahead.
Inadequate set is:
<e> (-1, 4, 4) at line 4
<e> (-1, 1, 1) at line 1
<e> (-1, 2, 2) at line 2
```

This diagnostic identifies three productions in an inadequate configuration set that LALR is unable to resolve through the use of look aheads. The symbol <e> in each of the last three lines is the variable naming the production. The number in parentheses are the rule number, alternative, and production number of the production. The minus sign preceding the rule number indicates the production does not have a significant semantic action. If the inadequate set's closure set had not been empty, a dashed line would have appeared between the productions in its basis set and those in its closure set.

More extensive information will appear in the listing. Here the diagnostics will take the form:

```
WARNING: One or more configurations converged on the same next
set. This implies infinite look ahead.

10. contention_set  look ahead level 1
   (    49)       39... "EOI" (->6)->-15
   (    50)       39... "+" (->7)->-15
   (    51)       40... "+" (->7)->-14
   (    52)       39... "*" (->8)->-15
   (    53)       41... "*" (->8)->-14
   (    54)       39... ")" (->13)->-15

10. inadequate_set <e>
   (    39) {<e>} ::= - <e>
   (    40) <e> ::= <e> {+} <e> -> 7
   (    41) <e> ::= <e> {*} <e> -> 8
```

First the contention set LALR is trying to eliminate is presented. In this example it is the tenth configuration set and represents a look ahead 1 state of the parse. The parenthesized numbers, 49 through 54, show that the contention set occupies elements 49 through 54 in LALR's CNFG table. The next column of numbers identify (by CNFG element number) the initial configuration in each look ahead string. The symbols in the middle column are the terminals being looked ahead at. The next column of numbers indicate which set to examine if still in contention after this level of look ahead. The last column of numbers indicate which set to examine if this level of look ahead resolves the contention. In this example there are two configurations "looking ahead at" the terminal "+", both examining set number 7 next and two configurations looking ahead at the terminal "*", both examining set number 8 next.

After the contention set, the inadequate set that LALR was trying to convert into look ahead sets and adequate sets is presented. In this example it was the tenth configuration set and the "read" symbol was the variable <e>. (A configuration is a production with one of its symbols designated as the "marked" symbol.) The parenthesized numbers, 39, 40, and 41, show that the inadequate set occupies elements 39, 40, and 41 in LALR's CNFG table. These numbers are followed by the configurations with their marked symbol indicated by enclosing it in braces. If the marked symbol is not the production's left hand side, the next set to examine when the symbol is read is shown at the extreme right.

After the diagnostic information is presented, LALR performs an error recovery to allow processing to continue. For infinite look ahead, the error recovery is simply the deletion of each configuration which has the same next set as some preceding configuration in the contention set has for the same terminal symbol. In this example, configuration 51 and 53 are deleted.

If the grammar shown in Figure 7 is replaced with the unambiguous grammar shown in Figure 8, if will be accepted by LALR.

```
<e> ::= <e> + <t> | <t> !
<t> ::= <t> * <f> | <f> !
<f> ::= - <f> | <p> !
<p> ::= ( <e> ) | i !
```

Figure 8. Unambiguous e.lalr (grammar only)

Now consider the look ahead 6 grammar shown in Figure 9. If this grammar is processed with a maximum look ahead of 4 specified it will be rejected. The following diagnostic will be written to the user_output I/O switch:

```
WARNING: Exceeded LALR (4).
Inadequate set is:
<a> (-2, 1, 3) at line 3
<A> (-8, 1, 9) at line 9
```

This diagnostic information is interpreted the same as described above for the infinite look ahead situation.

In the listing, the diagnostic will be:

```
WARNING:  Exceeded LALR (4) while processing this set of  configu-
rations.

    35. contention_set  look ahead level 5
       (     55)       12..."f" (-24)->-31
       (     56)       11..."f" (-26)->-30

     5. inadequate_set a
       (     11) {<a>} ::= a
       (     12) {<A>} ::= a
```

This diagnostic information is also interpreted as described above for infinite look ahead. In this example LALR is trying to generate look ahead sets to ultimely decide when the terminal "a" should be reduced to the non-terminal <a> and when it should be reduced to the non-terminal <A>.

The error recovery for exceeding the maximum look ahead is to ignore all except the first configuration in the contention set for each terminal symbol appearing in that set. In this example, configuration 56 is ignored causing the terminal "a" to be reduced to the non-terminal <A>.

```
<s> ::= <a> <b> <c> <d> <e> <f> p
      | <A> <B> <C> <D> <E> <F> z!
<a> ::= a!
<b> ::= b!
<c> ::= c!
<d> ::= d!
<e> ::= e!
<f> ::= f!
<A> ::= a!
<B> ::= b!
<C> ::= c!
<D> ::= d!
<E> ::= e!
<F> ::= f!
```

Figure 9. Look ahead 6 s.lalr (grammar only)

Finally consider the context sensitive grammar shown in Figure 10. When this grammar is processed the following diagnostic information will be written to the user_output I/O switch:

```
NOTE: The LALR (4) contention set is identical to the LALR (2)
contention set. This implies indefinite recursion.
Inadequate set is:
EOI (0, 0, 0) at line 0
- - - - - - - - - -
<S> (-1, 1, 1) at line 1
<S> (-1, 2, 2) at line 1
<a> (-2, 1, 3) at line 3
<a> (-2, 2, 4) at line 3
<b> (-3, 1, 5) at line 5
<b> (-3, 2, 6) at line 5
```

This diagnostic information is interpreted the same as described above for the infinite look ahead case.

In the listing, the following diagnostic information will appear:

```
NOTE:  The LALR (4)  contention set is identical  to the LALR  (2)
contention set.  This implies indefinite recursion.

    41. contention_set  look ahead level 4
      (    74)       5..."C"  (->19)->-27
      (    75)       7...","  (->16)->-28
      (    76)       5...","  (->20)->-27

     1. inadequate_set EOI
      (     1) EOI ::= {<S>} EOI -> 4
             - - - - - - - - - - -
      (     2) <S> ::= {<a>} <l> <c> -> 3
      (     3) <S> ::= {<b>} <m> <d> -> 2
      (     4) <a> ::= {A} -> 5
      (     5) {<a>} ::=
      (     6) <b> ::= {B} -> 6
      (     7) {<b>} ::=
```

This diagnostic information is also interpreted as described above for infinite look ahead.

Note: The first configuration set always has a single internally generated production as its basis set. The right hand side of this production is the user's start symbol followed by the terminal "EOI". This production is really anonymous, the use of the terminal "EOI" to name it is an artifact of the production display routines.

The error recovery for indefinite recursion is simply to generate a DPDA exhibiting the same indefinite recursion. In this example, this is done by directing the look ahead 3 transitions that would have gone to the look ahead 4 state back to the look ahead 2 state. This makes the look ahead 2 and 3 states behave as look ahead 2*N and 2*N+1 states, respectively, where N is the iteration count.

```
<S> ::= <a> <l> <c> | <b> <m> <d> !
<a> ::= A | !
<b> ::= B | !
<c> ::= C !
<d> ::= D !
<l> ::= I | I , <l> | J , <l> !
<m> ::= K | I , <m> | K , <m> !
```

Figure 10. S.lalr (grammar only)

Bibliography

This is a listing of many items having to to with language processing. LALR is based on much of this material. Of particular significance is that of Knuth [33], followed by DeRemer [13][14].

1. Aho, A. V. Denning, P. J. and Ullman, J. D. "Weak and mixed strategy precedence parsing." J. ACM 19, 2 (1972), 225-243
2. --- Johnson, S. C. and Ullman, J. D. "Deterministic Parsing of Ambiguous Grammars." Comm. ACM 18, 8 (1975), 441-452
3. --- Johnson, S. C. and Ullman, J. D. "Deterministic parsing of ambiguous grammars." Conference Record of ACM Symposium of Principles of Programming Languages (Oct. 1973), 1-21.
4. --- and Johnson, S. C. "LR Parsing." Computing Surveys 6, 2 (June 1974), 99-124.
5. --- and Peterson, T. G. "A minimum distance error-correcting parser for context-free languages." SIAM J. Computing 1, 4 (1972) 305-312
6. --- and Ullman, J. D. "A technique for speeding up LR(k) parsers." SIAM J. Computing 2, 2 (1973), 106-127
7. --- and Ullman, J. D. "Optimization of LR(k) parsers." J. Computer and System Sciences 6, 6 (1972), 573-602.
8. --- and Ullman, J. D. The theory of Parsing, Translation and Compiling. Prentice-Hall, Englewood Cliffs, N. J., 1972
9. Altman, V. E. A Language Implementation System. MS Thesis, Mass. Inst. Technology, 1973.
10. Anderson, T. Syntactic analysis of LR(k) languages. PhD Thesis, Univ. Newcastle-upon-Tyne, Northumberland, England (1972).
11. --- Eve, J. and Horning, J. J. "Efficient LR(1) parsers." Acta Informatica 2 (1973), 12-39
12. Conway, M. E. "Design of a separable transition-diagram compiler." Comm. ACM 6, 7 (July 1963), 396-408
13. DeRemer, F. L. "Practical translators for LR(k) languages." PhD Thesis, Oct. 1969, Project MAC Report MAC TR-65, MIT, Cambridge, Mass, 1969.
14. --- "Simple LR(k) grammars." Comm. ACM 14, 7 (1971), 453-460,
15. Demers, A. "Elimination of single productions and merging nonterminal symbols of LR(1) grammars." Technical Report TR-127, Computer Science Lab., Dept. of Electrical Engineering, Princeton Univ., Princeton, N. J., July 1973.
16. Demers, A. J. "Skeletal LR parsing." IEEE Conf. Record of 15th Annual Symposium of Switching and Automata Theory, 1974.
17. --- "An efficient context-free parsing algorithm." Comm. ACM 13, 2 (1970), 94-102.
18. Earley, J. Ambiguity and precedence in syntax description. Tech Rep. 13, Dept. Computer Science, Univ. of California, Berkeley.
19. El Djabri, N. Extending the LR parsing technique to some non-LR grammars. TR 121, Computer Science Lab., Dept. Electr. Eng., Princeton Univ., Princeton, N. J., 1973
20. Feldman, J. A. and Gries, D. "Translator writing systems." Comm. ACM 11, 2 (1968), 77-113.

21. Fischer, M. J. "Some properties of precedence languages." Proc. ACM Symposium on Theory of Computing, May 1969, pp. 181-190.
22. Floyd, R. W. "Syntactic analysis and operator precedence." J. ACM 10, 3 (1963), 316-333.
23. Friedman, E. P. "The inclusion problem for simple machines." Proc. Eighth Annual Princeton Conference on Information Sciences and Systems, 1974, pp. 173-177.
24. Ginsburg,S. and Spanier, E. H. "Control sets on grammars." Mathematical Systems Theory 2, 2 (1968), 159-178.
25. Graham, S. L. and Rhodes, S. P. "Practical syntactic error recovery in compilers." Conference Record of ACM Symposium on Principles of Programming Languages (Oct. 1973), 52-58.
26. Gries, D. Compiler Construction for Digital Computers. Wiley, New York, 1971.
27. Hopcroft, J. E. and Ullman, J. D. Formal Languages and their Relation to Automata. Addison-Wesley, Reading, Mass., 1969.
28. Ichbiah, J. D. and Morse, S. P. "A technique for generating almost optimal Floyd-Evans productions for precedence grammars." Comm. ACM 13, 8 (1970), 501-508.
29. James, L. R. "A syntax directed error recovery method." Technical Report CSRG-13, Computer Systems Research Group, Univ. Toronto, Toronto, Canada, 1972.
30. Jolliat, M. L. "On the reduced matrix representation of LR(k) parser tables." PhD Thesis, Univ. Toronto, Toronto, Canada (1973).
31. --- "Practical minimization of LR(k) parser tables." Proc. IFIP Congress 1974, pp. 376-380.
32. Kernighan, B. W. and Chery, L. L. "A system for typesetting mathematics." Comm. ACM 18, 3 (March 1975), 151-156.
33. Knuth, D. E. "On the translation of languages from left to right." Information and Control 8, 6 (1965), 607-639. (Note: this paper contains the original definition of LR grammars and languages).
34. --- "Top down syntax analysis." Acta Informatica 1, 2 (1971), 97-110.
35. Korenjak, A. J. "A practical method of constructing LR(k) processors." Comm. ACM 12, 11 (1969), 613-623.
36. --- and Hopcroft, J. E. "Simple deterministic languages." IEEE Conf. Record of 7th Annual Symposium on Switching and Automata Theory, 1966 pp. 36-46.
37. Lalonde, W. R. Lee, E. S. and Horning, J. J. "An LALR(k) parser generator." Proc. IFIP Congress 71. TA-3, North-Hollad Publishing Co., Amsterdam, the Netherlands (1971), pp. 153-157.
38. Leinius, P. "Error detection and recovery for syntax directed compiler systems." PhD Thesis, Univ. Wisconsin, Madison, Wisc. (1970).
39. Lewis, P. M. and Rosenkrantz, D. J. "An Algol compiler designed using automata theory." Proc. Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, N. Y., 1971, pp. 75-88.
40. --- Rosenkrantz, D. J. and Stearns, R. E. "Attributed translations." Proc. Fifth Annual ACM Symposium on theory of Computing (1973) Pages 160-171.
41. --- and Stearns, R. E. "Syntax directed transduction." J. ACM 15, 3 (1968), 464-488.
42. Manna, Z., Ness, S. and Vuillemin, J. "Inductive methods for proving properties of programs." Proc. ACM Conf. on Proving Assertions About Programs, 1972, pp. 27-50.

43. McGruther, T. "An approach to automating syntax error detection, recovery, and correction for LR(k) grammars." Master's Thesis, Naval Postgraduate School, Montery, Calif., 1972.
44. McKeeman, W. M. Horning, J. J. and Wortman, D. B. A Compiler Generator. Prentice-Hall, Englewood Cliffs, N. J., 1970.
45. Mickunas, M. D. and Schneider, V. B. "A parser generating system for constructing compressed compilers." Comm. ACM 16, 11 (November 1973), 667-675.
46. Pager, D. "A fast left-to-right parser for context-free grammars." Technical Report PE-24 , Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1972.
47. Pager, D. "A solution to an open problem by Knuth." Information and Control 17 (1970), 462-473.
48. --- "On eliminating unit productions from LR(k) parsers." Technical Report. (See 26). 1974.
49. --- "On the incremental approach to left-to-right parsing." Technical Report PE 238, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1972a.
50. Peterson, T. G. "Syntax error detection, correction and recovery in parsers." PhD Thesis, Stevens Institute of Technology, Hoboken, N. J., 1972.
51. Rosenkrantz, D. J. and Stearns, R. E. "Properties of deterministic top-down grammars." Inf. Control 14, 5 (1969), 226-256.
52. Stearns, R. E. "Deterministic top-down parsing." Proc. Fifth Annual Princeton Conf. on Information Science and Systems, 1971, pp. 182-188
53. Walters, D. A. "Deterministic context-sensitive languages." Inf. Contr. 8 (1970), 14-61.
54. Wood, D. "The theory of left factored languages." Computer J. 12, 4 (1969) 349-356 and 13, 1 (1970), 55-62.
55. Dijkstra, E. W. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N. J., 1976

```
GENERATION LISTING OF SEGMENT calc2_

Generated by: Prange.SLANG.a using LALR 7.0 of
              Saturday, September 25, 1982
Generated at: TCO 68/80 Multics Billerica, Ma.
Generated on: 09/25/82  1707.6 edt Sat
     Options: -ssl -terms -ct -ot -dx -ll 65 -pl 46
File options: -order -tl -table -sem -production

28     1     1 <calc> ::= <line...> | !

29     2     3 <line...> ::= <line> |
30           4            <line...> <line>!

31     3     5 <line> ::= list <nl> |
32           6            <symbol> = <expression> <nl> |
33           7            <expression> <nl> |
34           8            <nl>!

46     4     9 <expression> ::= <term> |
47          10            <expression> + <term> |
48          11            <expression> - <term> !

55     5    12 <term> ::= <factor> |
56          13            <term> * <factor> |
57          14            <term> / <factor> !

64     6    15 <factor> ::= <primary> |
65          16            <factor> ** <primary>!

69     7    17 <primary> ::= <reference> |
70          18            + <primary> |
71          19            - <primary> |
72          20            (<expression>)  !

82     8    21 <reference> ::= <real> |
83          22            <symbol> |
84          23            e |
85          24            pi |
86          25            sin  (<expression>) |
87          26            cos  (<expression>) |
88          27            tan  (<expression>) |
89          28            atan  (<expression>) |
90          29            abs  (<expression>) |
91          30            ln  (<expression>) |
92          31            log  (<expression>)  !
```

```
          SOURCE FILES USED IN THIS GENERATION.

LINE      NUMBER  DATE MODIFIED      NAME
                    PATHNAME
             0    09/02/82  1522.6  calc2_.lalr
>user_dir_dir>SLANG>Prange>stb>calc.s::calc2_.lalr
```

```
*******************************
*      1 Look ahead           *
*      8 Rules                *
*     31 Productions          *
*      8 Variables            *
*     21 Terminals            *
*      0 Synonyms             *
*     72 States               *
*    214 DPDA words           *
*      0 SKIP words           *
*                             *
*     Optimization removed    *
*     88 Read Transitions     *
*     29 Look Transitions     *
*      9 Read/Look States     *
*     23 Lookback Transitions *
*      3 Apply States         *
*      0 MLook Transitions    *
*      0 MLook States         *
*    102 DPDA words           *
*******************************
```

TERMINALS USED

SYMBOL	CODE	REFERENCES
(	6	ref 72 86 87 88 89 90 91 92
)	7	ref 72 86 87 88 89 90 91 92
*	4	ref 56
**	9	ref 65
+	2	ref 47 70
-	3	ref 48 71
/	5	ref 57
<nl>	8	ref 31 32 33 34
<real>	10	ref 82
<symbol>	11	ref 32 83
=	1	ref 32
abs	12	ref 90
atan	13	ref 89
cos	14	ref 87
e	15	ref 84
list	16	ref 31
ln	17	ref 91
log	18	ref 92
pi	19	ref 85
sin	20	ref 86
tan	21	ref 88

VARIABLES USED

SYMBOL	CODE	REFERENCES
<calc>	-1	def 28 28 ref "Start Symbol"
<expression>	-4	def 46 47 48 ref 32 33 47 48 72 86 87 88 89 90 91 92
<factor>	-6	def 64 65 ref 55 56 57 65
<line...>	-2	def 29 30 ref 28 30
<line>	-3	def 31 32 33 34 ref 29 30
<primary>	-7	def 69 70 71 72 ref 64 65 70 71
<reference>	-8	def 82 83 84 85 86 87 88 89 90 91 92 ref 69
<term>	-5	def 55 56 57 ref 46 47 48 56 57

TERMINAL ENCODING

1	=	7	)	13	atan	19	pi
2	+	8	<nl>	14	cos	20	sin
3	-	9	**	15	e	21	tan
4	*	10	<real>	16	list		
5	/	11	<symbol>	17	ln		
6	(	12	abs	18	log		

```
DPDA LISTING OF SEGMENT
     >udd>SLANG>Prange>stb>calc2_.grammar
Generated by: Prange.SLANG.a using LALR 7.0 of
     Saturday, September 25, 1982
Generated at: TCO 68/80 Multics Billerica, Ma.
Generated on: 09/25/82  1707.6 edt Sat
Generated from:
     >udd>SLANG>Prange>stb>calc.s::calc2_.lalr
Maximum look ahead: 1
```

DPDA LISTING

```
     1  00017    00002   RD/LK CON
       -00002    00051   CONTINUED AT
        00000->-00183   LOOK "EOI"
Refs: 4D 10D 13D 16D 19D 24D 72D 76D 79D 82D 186D 191D 194D 197D
209D 212D

     4  00011    00005   APPLY
        00000    00000   sd/RFU
       -00017    00019   prod/def
        00036-> 00010
        00057-> 00013
        00059-> 00024
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R 94A 103A 113A 162A 165A 168A 171A 174A 177A 180A

    10  00013    00002   APPLY SHARE
        00001    00001   sd/RFU
        00018    00004   prod/ust
Refs: 4A 10A 13A 16A

    13  00013    00002   APPLY SHARE
        00001    00001   sd/RFU
        00019    00004   prod/ust
Refs: 4A 10A 13A 16A

    16  00013    00002   APPLY SHARE
        00002    00002   sd/RFU
        00020    00004   prod/ust
Refs: 141R

    19  00011    00004   APPLY
        00000    00000   sd/RFU
       -00015    00027   prod/def
        00060-> 00116
        00061-> 00119
Refs: 4A 10A 13A 16A

    24  00013    00002   APPLY SHARE
        00002    00002   sd/RFU
        00016    00019   prod/ust
Refs: 4A 10A 13A 16A

    27  00015    00002   RD/LK DEF
       -00001->-00186   LOOK DEFAULT
```

```
      * 00009-> 00059  READ "**"
Refs: 19A 24A

    30  00015   00003  RD/LK DEF
       -00001->-00197  LOOK DEFAULT
      * 00004-> 00060  READ "*"
      * 00005-> 00061  READ "/"
Refs: 186A 191A 194A

    34  00000   00001  READ/LOOK
      * 00000-> 00000  READ "EOI"
Refs: 183A

    36  00000   00014  READ/LOOK
      * 00002-> 00036  READ "+"
      * 00003-> 00057  READ "-"
      * 00006-> 00058  READ "("
      * 00010-> 00004  READ <real>
      * 00011-> 00113  READ <symbol>
      * 00012-> 00088  READ "abs"
      * 00013-> 00090  READ "atan"
      * 00014-> 00092  READ "cos"
      * 00015-> 00094  READ "e"
      * 00017-> 00099  READ "ln"
      * 00018-> 00101  READ "log"
      * 00019-> 00103  READ "pi"
      * 00020-> 00106  READ "sin"
      * 00021-> 00108  READ "tan"
Refs: 1R 4B 10B 13B 16B 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R
65R 66R 67R 68R 69R 70R 71R

    51  00017   00005  RD/LK CON
       -00002   00036  CONTINUED AT
      * 00000-> 00000  READ "EOI"
      * 00008-> 00072  READ <nl>
      * 00011-> 00085  READ <symbol>
      * 00016-> 00097  READ "list"
Refs: 4D 10D 13D 16D 19D 24D 72A 72B 76A 76B 79A 79B 82A 82B 110A
186D 191D 194D 197D 209D 212D

    57  00002   00036  RD/LK SHARE
Refs: 1R 4B 10B 13B 16B 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R
65R 66R 67R 68R 69R 70R 71R
```

DPDA LISTING

58 00002 00036 RD/LK SHARE
Refs: 1R 4D 10D 13D 16D 19D 24D 36R 51R 57R 58R 59R 60R 61R 62R
63R 64R 65R 66R 67R 68R 69R 70R 71R 186D 191D 194D 197B 209B 212B

59 00002 00036 RD/LK SHARE
Refs: 4B 10B 13B 16B 27R 116R 119R

60 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19B 24B 30R 122R 126R

61 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19B 24B 30R 122R 126R

62 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 130R 134R 137R 141R 144R 147R 150R
153R 156R 159R 186B 191B 194B

63 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 130R 134R 137R 141R 144R 147R 150R
153R 156R 159R 186B 191B 194B

64 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 85R 186D 191D 194D 197B 209B 212B

65 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 88R 186D 191D 194D 197B 209B 212B

66 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 90R 186D 191D 194D 197B 209B 212B

67 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 92R 186D 191D 194D 197B 209B 212B

68 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 99R 186D 191D 194D 197B 209B 212B

69 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 101R 186D 191D 194D 197B 209B 212B

70 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 106R 186D 191D 194D 197B 209B 212B

71 00002 00036 RD/LK SHARE
Refs: 4D 10D 13D 16D 19D 24D 108R 186D 191D 194D 197B 209B 212B

```
   72  00011    00003  APPLY
       00000    00000  sd/RFU
      -00008    00051  prod/def
       00051-> 00110
Refs: 1R 51R

   76  00013    00002  APPLY SHARE
       00001    00001  sd/RFU
       00007    00072  prod/ust
Refs: 134R

   79  00013    00002  APPLY SHARE
       00001    00001  sd/RFU
       00005    00072  prod/ust
Refs: 97R

   82  00013    00002  APPLY SHARE
       00003    00003  sd/RFU
       00006    00072  prod/ust
Refs: 130R

   85  00015    00002  RD/LK DEF
      -00001->-00113  LOOK DEFAULT
     * 00001-> 00064  READ "="
Refs: 1R 51R

   88  00000    00001  READ/LOOK
     * 00006-> 00065  READ "("
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

   90  00000    00001  READ/LOOK
     * 00006-> 00066  READ "("
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

   92  00000    00001  READ/LOOK
     * 00006-> 00067  READ "("
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

   94  00012    00002  APPLY 1
       00000    00000  sd/RFU
       00023    00004  prod/tran
```

DPDA LISTING

```
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

   97  00000    00001  READ/LOOK
     * 00008-> 00079  READ <nl>
Refs: 1R 51R

   99  00000    00001  READ/LOOK
     * 00006-> 00068  READ "("
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

  101  00000    00001  READ/LOOK
     * 00006-> 00069  READ "("
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

  103  00012    00002  APPLY 1
       00000    00000  sd/RFU
       00024    00004  prod/tran
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

  106  00000    00001  READ/LOOK
     * 00006-> 00070  READ "("
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

  108  00000    00001  READ/LOOK
     * 00006-> 00071  READ "("
Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R
69R 70R 71R

  110  00012    00002  APPLY 1
       00001    00001  sd/RFU
      -00004    00051  prod/tran
Refs: 72A 76A 79A 82A

  113  00012    00002  APPLY 1
       00000    00000  sd/RFU
       00022    00004  prod/tran
Refs: 36R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R 67R 68R 69R 70R
71R 85L
```

```
  116  00015    00002  RD/LK DEF
      -00001->-00191  LOOK DEFAULT
     * 00009-> 00059  READ "**"
Refs: 19A 24A

  119  00015    00002  RD/LK DEF
      -00001->-00194  LOOK DEFAULT
     * 00009-> 00059  READ "**"
Refs: 19A 24A

  122  00015    00003  RD/LK DEF
      -00001->-00209  LOOK DEFAULT
     * 00004-> 00060  READ "*"
     * 00005-> 00061  READ "/"
Refs: 186A 191A 194A

  126  00015    00003  RD/LK DEF
      -00001->-00212  LOOK DEFAULT
     * 00004-> 00060  READ "*"
     * 00005-> 00061  READ "/"
Refs: 186A 191A 194A

  130  00000    00003  READ/LOOK
     * 00002-> 00062  READ "+"
     * 00003-> 00063  READ "-"
     * 00008-> 00082  READ <nl>
Refs: 197A 209A 212A

  134  00017    00002  RD/LK CON
      -00002    00130  CONTINUED AT
     * 00008-> 00076  READ <nl>
Refs: 197A 209A 212A

  137  00000    00003  READ/LOOK
     * 00002-> 00062  READ "+"
     * 00003-> 00063  READ "-"
     * 00007-> 00180  READ ")"
Refs: 197A 209A 212A

  141  00017    00002  RD/LK CON
      -00002    00137  CONTINUED AT
     * 00007-> 00016  READ ")"
Refs: 197A 209A 212A
```

DPDA LISTING

```
  144  00017    00002  RD/LK CON
      -00002    00137  CONTINUED AT
     * 00007-> 00162  READ ")"
Refs: 197A 209A 212A

  147  00017    00002  RD/LK CON
      -00002    00137  CONTINUED AT
     * 00007-> 00165  READ ")"
Refs: 197A 209A 212A

  150  00017    00002  RD/LK CON
      -00002    00137  CONTINUED AT
     * 00007-> 00168  READ ")"
Refs: 197A 209A 212A

  153  00017    00002  RD/LK CON
      -00002    00137  CONTINUED AT
     * 00007-> 00171  READ ")"
Refs: 197A 209A 212A

  156  00017    00002  RD/LK CON
      -00002    00137  CONTINUED AT
     * 00007-> 00174  READ ")"
Refs: 197A 209A 212A

  159  00017    00002  RD/LK CON
      -00002    00137  CONTINUED AT
     * 00007-> 00177  READ ")"
Refs: 197A 209A 212A

  162  00012    00002  APPLY 1
       00003    00003  sd/RFU
       00029    00004  prod/tran
Refs: 144R

  165  00012    00002  APPLY 1
       00003    00003  sd/RFU
       00028    00004  prod/tran
Refs: 147R

  168  00012    00002  APPLY 1
       00003    00003  sd/RFU
       00026    00004  prod/tran
Refs: 150R
```

```
  171  00012    00002   APPLY 1
       00003    00003   sd/RFU
       00030    00004   prod/tran
Refs: 153R

  174  00012    00002   APPLY 1
       00003    00003   sd/RFU
       00031    00004   prod/tran
Refs: 156R

  177  00012    00002   APPLY 1
       00003    00003   sd/RFU
       00025    00004   prod/tran
Refs: 159R

  180  00012    00002   APPLY 1
       00003    00003   sd/RFU
       00027    00004   prod/tran
Refs: 137R

  183  00012    00002   APPLY 1
      -00001   -00001   sd/RFU
     *-00002    00034   prod/tran
Refs: 1L

  186  00011    00004   APPLY
       00000    00000   sd/RFU
      -00012    00030   prod/def
       00062->  00122
       00063->  00126
Refs: 27L

  191  00013    00002   APPLY SHARE
       00002    00002   sd/RFU
       00013    00186   prod/ust
Refs: 116L

  194  00013    00002   APPLY SHARE
       00002    00002   sd/RFU
       00014    00186   prod/ust
Refs: 119L

  197  00011    00011   APPLY
       00000    00000   sd/RFU
```

DPDA LISTING

```
         -00009    00134  prod/def
          00058->  00141
          00064->  00130
          00065->  00144
          00066->  00147
          00067->  00150
          00068->  00153
          00069->  00156
          00070->  00159
          00071->  00137
Refs: 30L

    209   00013    00002  APPLY SHARE
          00002    00002  sd/RFU
          00010    00197  prod/ust
Refs: 122L

    212   00013    00002  APPLY SHARE
          00002    00002  sd/RFU
          00011    00197  prod/ust
Refs: 126L
```

DPDA LISTING

TERMINAL REFERENCES

```
( (6) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R 88R 90R 92R 99R 101R 106R 108R
) (7) Refs: 137R 141R 144R 147R 150R 153R 156R 159R
* (4) Refs: 30R 122R 126R
** (9) Refs: 27R 116R 119R
+ (2) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R 130R 134R 137R 141R 144R 147R 150R
        153R 156R 159R
- (3) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R 130R 134R 137R 141R 144R 147R 150R
        153R 156R 159R
/ (5) Refs: 30R 122R 126R
<nl> (8) Refs: 1R 51R 97R 130R 134R
<real> (10) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R
        66R 67R 68R 69R 70R 71R
<symbol> (11) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R
        65R 66R 67R 68R 69R 70R 71R
= (1) Refs: 85R
EOI (i.e., end of information) (0) Refs: 1L 34R 51R
abs (12) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R
atan (13) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R
        66R 67R 68R 69R 70R 71R
cos (14) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R
e (15) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R
list (16) Refs: 1R 51R
ln (17) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R
log (18) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R
pi (19) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R
sin (20) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R
tan (21) Refs: 1R 36R 51R 57R 58R 59R 60R 61R 62R 63R 64R 65R 66R
        67R 68R 69R 70R 71R
```

DPDA LISTING

VARIABLE REFERENCES

```
<calc> (-1) Refs: 183S
<expression> (-4) Refs: 197T 209U 212U
<factor> (-6) Refs: 19T 24U
<line...> (-2) Refs: 110S
<line> (-3) Refs: 72T 76U 79U 82U
<primary> (-7) Refs: 4T 10U 13U 16U
<reference> (-8) Refs: 94S 103S 113S 162S 165S 168S 171S 174S
        177S 180S
<term> (-5) Refs: 186T 191U 194U
```

COMPILATION LISTING OF SEGMENT calc2

Compiled by: Multics PL/I Compiler, Release 27b, of September 15, 1981
Compiled at: TCO 68/80 Multics Billerica, Ma.
Compiled on: 09/18/82 1457.2 edt Sat

Options: optimize table map

```
calc2:  proc;


/* Version  of calc using  LALR.  */

dcl arg_list_ptr  ptr;

dcl buffer  char (buffer_length) based  (buffer_ptr);
dcl buffer_length  fixed bin (21);
dcl buffer_ptr  ptr;

dcl cleanup  condition;
dcl code  fixed bin (35);
dcl com_err_  entry options (variable);

dcl cu_$arg_count  entry (fixed bin,  fixed bin (35));
dcl cu_$arg_list_ptr  entry (ptr);
dcl current_arg  fixed bin;

dcl input  char (input_length) based  (input_ptr);
dcl input_length  fixed bin (21);
dcl input_ptr  ptr;

dcl ioa_  entry options (variable);
dcl line_number  fixed bin;
dcl msg  char (100) varying;

dcl my_name  char (5) internal  static options (constant)  init ("calc2");
dcl newline  char (1) internal  static options (constant)  init ("
");

dcl next_char_pos  fixed bin;
dcl null  builtin;
dcl number_of_args  fixed bin;

dcl quit_arg  char (2) internal  static options (constant)  init ("q
");
dcl 1  sym based like  sym_;

dcl 1  sym_ (200),
2 name  char (8),
2 val  float bin (27);

dcl sym_num  fixed bin;
```

```
call cu_$arg_count  (number_of_args, code);
if code  ^= 0
then do;
bail_out:
call com_err_  (code, my_name);
return;
end;
call cu_$arg_list_ptr  (arg_list_ptr);
current_arg =  0;
sym_num =  0;
line_number =  0;
buffer_ptr =  null ();
if number_of_args  = 0
then do;
on cleanup  go to exit;
buffer_length =  200;
allocate buffer  set (buffer_ptr);
input_ptr =  buffer_ptr;
end;
retry:
next_char_pos =  1;
input_length =  0;
call calc2_p;
if code  ^= 0 then
if number_of_args  = 0 then
go to  retry;
exit:
if buffer_ptr  ^= null ()  then
free buffer;
return;
error:
call ioa_  ("^a", msg);
if number_of_args  = 0 then
go to  retry;
else go  to exit;

trn:  entry;
db_sw =  "1"b;
```

```
return;

trf:  entry;
db_sw =  "0"b;

return;

dcl  db_sw bit (1)  internal static init  ("0"b);

/*  Recovery failed.  */
dcl  syntax_error fixed bin  (35) internal static  options (constant) init  (1);
/*  Parse stack underflow  or local recovery  encountered

* impossible  conditions.  Both caused  by bad DPDA.   */
dcl  logic_error fixed bin  (35) internal static  options (constant) init  (2);
/*  Parse, lexical, or  lookahead stack overflow.   */

dcl  stack_overflow fixed bin  (35) internal static  options (constant) init  (3);
/*  Unrecognized table type  in the DPDA.   */
dcl  unrecognized_state fixed bin  (35) internal static  options (constant) init  (4);

calc2_p:   proc;

/*  Parser for tables  created by LALR.   */



/*  BEGIN INCLUDE FILE  .....  calc2_t_.incl.pl1 .....

*
*
*SCANNER AND  PARSER TABLES FROM  SEGMENT

* >user_dir_dir>SLANG>Prange>stb>calc2_.grammar
*
* Generated  by:  Prange.SLANG.a using  LALR 7.0 of  Friday, September 17,  1982

* Generated  at:  TCO 68/80  Multics Billerica, Ma.
* Generated  on:  09/18/82 1408.0  edt Sat
* Generated  from:  >user_dir_dir>SLANG>Prange>stb>calc.s::calc2_.lalr */


dcl  1 calc2_t_$terminals_list external  static,
2  terminals_list_size fixed bin,

2  terminals_list (21),
3  position fixed bin  (18) unsigned unaligned,
3  length fixed bin  (18) unsigned unaligned;


dcl  1 calc2_t_$terminal_characters external  static,
```

```
2  terminal_characters_length fixed bin,
2  terminal_characters char (55);
dcl  1 calc2_t_$dpda external  static,
2  dpda_size fixed bin,
2  dpda (214),
3  (v1, v2) fixed  bin (17) unaligned;
dcl  1 calc2_t_$skip external  static,
2  skip_size fixed bin,
2  skip (2),
3  (v1, v2) fixed  bin (17) unaligned;
dcl  1 calc2_t_$standard_prelude external  static,
2  standard_prelude_length fixed bin,
2  standard_prelude char (0);
dcl  1 calc2_t_$production_names external  static,
2  production_names_size fixed bin,
2  production_names (31) fixed  bin (17) unaligned;
dcl  1 calc2_t_$variables_list external  static,
2  variables_list_size fixed bin,
2  variables_list (8),
3  (position, length) fixed  bin (18) unsigned  unaligned;
dcl  1 calc2_t_$variable_characters external  static,
2  variable_characters_length fixed bin,
2  variable_characters char (67);
/*  END INCLUDE FILE  .....  calc2_t_.incl.pl1 .....   */
dcl  1 lstk (-1:50),
/*  -1:-1 is the  lookahead stack (FIFO)  */
/*  1:50 is the  lexical stack (LIFO)  */
2  symptr ptr, /*  pointer to symbol  (must be valid)  */
2  symlen fixed bin,  /* length of  symbol (may be  0) */
2  line_id aligned, /*  identification of line  where symbol begins  */
3  file fixed bin  (17) unaligned, /*  the include file  number */
```

```
3  line fixed bin  (17) unaligned, /*  the line number  within the include  file */
2  symbol fixed bin,  /* encoding of  the symbol */
2  value float bin  (27),
2  def ptr;
dcl  1 lookahead (-1:50)  defined lstk like  lstk;
dcl  abs builtin;
dcl  current_state fixed bin;  /* number of  current state */
dcl  current_table fixed bin;  /* number of  current table */
dcl  1 db_data unaligned,
2  flag char (1),  /* * means  stacked */
2  state picture "zzz9",
2  top picture "zzz9",
2  filler char (2),
2  type char (6),
2  data char (100);
dcl  db_item char (117)  defined (db_data);
dcl  db_separator char (1);
dcl  divide builtin;
dcl  hbound builtin;
dcl  i fixed bin;  /* temp */
dcl  ioa_$nnl entry options  (variable);
dcl  lb fixed bin;
dcl  ls_top fixed bin  defined parse_stack_top; /*  location of the  top of the  lexical stack */
dcl  lookahead_count fixed bin;  /* number of  terminals in lookahead  stack */
dcl  lookahead_get fixed bin  internal static options  (constant) init (-1);
dcl  lookahead_put fixed bin  internal static options  (constant) init (-1);
dcl  next_state fixed bin;  /* number of  next state */
dcl  parse_stack (50) fixed  bin aligned; /*  parse stack */
dcl  parse_stack_top fixed bin;  /* location of  the top of  the parse stack  */
dcl  production_number fixed bin;  /* APPLY production  number */
dcl  recov_msg char (250)  varying;
dcl  t fixed bin;
dcl  test_state fixed bin;  /* top state  from parse stack  during look back  lookups */
dcl  test_symbol fixed bin  defined lstk.symbol (-1);  /* encoding of  current symbol */
dcl  ub fixed bin;
dcl  unspec builtin;
```

```
current_state  = 1;
parse_stack_top  = 0;
lookahead_count  = 0;
unspec  (lstk) = ""b;
code  = 0; /*  Preset the status  code.  */

/*  The parsing loop.   */
NEXT:
if  current_state = 0
then  do;
parse_done:
return;
end;
current_table  = current_state;
db_item  = "";
db_data.state  = current_state;
db_data.top  = parse_stack_top;
goto  CASE (dpda.v1 (current_table));

CASE  (10):  /* Obsolete  -- Lookahead 1  (sometimes called read  without
stacking)  with shared transition  table.  */

CASE  (2):  /* Read  and stack and/or  lookahead 1 (sometimes  called
read  without stacking) with  shared transition table.
(Read  transitions to state  S are coded  as +S while
lookahead  transitions to state  S are coded  -S.)  */
current_table  = dpda.v2 (current_table);

CASE  (0):  /* Read  and stack and/or  lookahead 1 with  neither a
default  transition nor a  marked symbol transition.   */
CASE  (9):  /* Obsolete  -- Lookahead 1  (sometimes called
read  without stacking).  */
CASE  (15):  /* Read  and stack and/or  lookahead 1 with
a  default transition.  */
CASE  (17):  /* Read  and stack and/or  lookahead 1 with  the table
continued  at another state.   */

if  lookahead_count <= 0  /* Make sure  a symbol is  available.  */
```

```
then  do;

call  scanner;
lookahead_count  = lookahead_count+1;
end;

search_table:
/*  Look current symbol  up in the  read list.  */
lb  = current_table+1;

ub  = current_table+dpda.v2 (current_table);
do  while (lb <=  ub);
i  = divide (ub+lb,  2, 17, 0);

if  dpda.v1 (i) =  test_symbol
then  do;
next_state  = dpda.v2 (i);

goto  got_symbol;
end;
else  if dpda.v1 (i)  < test_symbol then

lb  = i+1;
else  ub = i-1;
end;

if  dpda.v1 (current_table+1) <  0 then
if  dpda.v1 (current_table+1) =  -1
then  do;

current_state  = -dpda.v2 (current_table+1);
if  db_sw
then  do;

db_data.type  = "LKO1D";
db_data.data  = get_terminal (lookahead_get);
call  ioa_$nnl ("^a^/", db_item);

end;
goto  NEXT;
end;

else  do;
current_table  = dpda.v2 (current_table+1);
goto  search_table;

end;

if  db_sw then

call  ioa_$nnl (" ^4i  ", current_state);
call  set_line_id (lookahead_get);
```

```
recov_msg  = recov_msg ||  "at ";
recov_msg  = recov_msg ||  get_terminal (lookahead_get);
recov_msg  = recov_msg ||  ".";
call  print_recov_msg;
code  = syntax_error;
go  to parse_done;

got_symbol:
if  db_sw then
db_data.data  = get_terminal (lookahead_get);
if  next_state < 0
then  do; /* This  is a lookahead  transition.  */
db_data.type  = "LK01";
current_state  = -next_state;
end;
else  do; /* This  is a read  transition.  */
db_data.type  = "READ";
db_data.flag  = "*";
if  parse_stack_top >= hbound  (parse_stack, 1) then
call  parse_stack_overflow;
parse_stack_top  = parse_stack_top+1;
parse_stack  (parse_stack_top) = current_state;  /* Stack the  current state.  */
unspec  (lstk (parse_stack_top)) =  unspec (lookahead (lookahead_get));
lookahead_count  = 0;
current_state  = next_state;
end;
if  db_sw then
call  ioa_$nnl ("^a^/", db_item);
goto  NEXT;

CASE  (3):  /* Multiple  lookahead (k >  1) with shared  look table.  */
CASE  (1):  /* Multiple  lookahead (k >  1) without default  transition.  */
CASE  (14):  /* Multiple  lookahead (k >  1) with default  transition.  */
CASE  (16):  /* Multiple  lookahead (k >  1) with the  table
 continued  at another state.   */

CASE  (7):  /* Obsolete  state type --  Skip table.  */
CASE  (8):  /* Obsolete  state type --  Skip recovery adjust  table.  */
```

```
CASE  (4):  /* Apply  by rule and  alternative with lookback  table.  */

CASE  (5):  /* Apply  by rule and  alternative without lookback.   */
CASE  (6):  /* Apply  by rule and  alternative with shared  lookback table.  */
call  set_line_id (lookahead_get);

recov_msg  = recov_msg ||  "Unrecognized DPDA state  encountered -- Parse  fails.";
call  print_recov_msg;
code  = unrecognized_state;

go  to parse_done;

CASE  (13):  /* Apply  by production with  shared lookback table.   */

current_table  = dpda.v2 (current_state+2);
CASE  (11):  /* Apply  by production with  lookback table.  */
CASE  (12):  /* Apply  by production without  lookback.  */

production_number  = dpda.v1 (current_state+2);
if  production_number > 0  then
call  calc2_ (production_number);

if  db_sw
then  begin;
dcl  production_name char (variables_list.length  (-production_names (abs (production_number))))

defined  (variable_characters)
position  (variables_list.position (-production_names (abs  (production_number))));
db_data.type  = "APLY";

db_data.data  = "(";
if  dpda.v1 (current_state+1) <  0 then
db_data.flag  = "*";

call  ioa_$nnl ("^a^i", db_item,  production_number);
if  production_names_size > 0  then
call  ioa_$nnl (" ^a",  production_name);

call  ioa_$nnl (")^-sd =  ^i ", dpda.v1  (current_state+1));
if  dpda.v1 (current_state+1) >  0
then  do;

db_separator  = "(";
do  t = parse_stack_top  to parse_stack_top-dpda.v1 (current_state+1)+1  by -1;
call  ioa_$nnl ("^1a^d", db_separator,  parse_stack (t));

db_separator  = "";
end;
call  ioa_$nnl (")");

end;
call  ioa_$nnl ("^/");
```

```
end;
/*  Check for an  apply of an  empty production.
* In  this case the  apply state number  must be
* pushed  onto the parse  stack.  (Reference
* LaLonde,  W.  R.:  An  efficient LALR Parser  Generator.
* Tech.   Report CSRG-2, 1971,  pp.  34-35.)  */
if  dpda.v1 (current_state+1) <  0
then  do;
if  parse_stack_top >= hbound  (parse_stack, 1) then
call  parse_stack_overflow;
parse_stack  (parse_stack_top+1) = current_state;
end;
/*  Delete lexical &  parse stack entries.   */
parse_stack_top  = parse_stack_top-dpda.v1 (current_state+1);
if  parse_stack_top <= 0
then  do;
call  set_line_id (lookahead_get);
recov_msg  = recov_msg ||  "lexical/parse stack empty  -- Parse fails.";
call  print_recov_msg;
code  = logic_error;
go  to parse_done;
end;
lb  = current_table+3;
ub  = current_table+dpda.v2 (current_table);
test_state  = parse_stack (parse_stack_top);
do  while (lb <=  ub);
i  = divide (ub+lb,  2, 17, 0);
if  dpda.v1 (i) =  test_state
then  do;
current_state  = dpda.v2 (i);
goto  NEXT;
end;
else  if dpda.v1 (i)  < test_state then
lb  = i+1;
else  ub = i-1;
end;
current_state  = dpda.v2 (current_table+2);
goto  NEXT;
```

```
get_terminal:   proc (lstk_index) returns  (char (100) varying);

dcl  lstk_index fixed bin  parameter;
dcl  alphanumeric (0:511) bit  (1) unaligned internal  static options (constant)  init (
(32)  (1) "0"b, /*  control characters */
(4)  (1) "0"b, /*  SP !  "  # */
"1"b,  /* $ */
(11)  (1) "0"b, /*  % & '  ( ) *  + , -  .  / */
(10)  (1) "1"b, /*  digits */
(7)  (1) "0"b, /*  :; < =  > ?  @  */
(26)  (1) "1"b, /*  upper case letters  */
(4)  (1) "0"b, /*  [ \ ]  ^ */
"1"b,  /* underscore */
"0"b,  /*   */
(26)  (1) "1"b, /*  lower case letters  */
(5)  (1) "0"b, /*  { | }  ¬ DEL */
(384)  (1) "0"b); /*  rest of 9-bit  ASCII code set  */

if  lstk.symbol (lstk_index) =  0 then
return  ("end-of-information");
else  begin;
dcl  temp char (100)  varying;
dcl  (length, min, rank,  substr) builtin;
dcl  symbol char (min  (50, lstk.symlen (lstk_index)))  based (lstk.symptr (lstk_index));
dcl  terminal char (terminals_list.length  (lstk.symbol (lstk_index)))
defined  (terminal_characters)
position  (terminals_list.position (lstk.symbol (lstk_index)));
if  length (terminal) >  2
&  substr (terminal, 1,  1) = "<"
&  substr (terminal, length  (terminal), 1) =  ">"
then  do;
temp  = substr (terminal,  2, length (terminal)-2);
if  length (symbol) >  0
then  do;
temp  = temp ||  " ";
if  substr (symbol, 1,  1) = """"
|  substr (symbol, 1,  1) = "'"  then
temp  = temp ||  symbol;
else  do;
```

```
temp  = temp ||  """";
temp  = temp ||  symbol;
temp  = temp ||  """";
end;
end;
end;
else  if alphanumeric (rank  (substr (terminal, 1,  1)))
then  do;
temp  = "reserved word  """;
if  length (symbol) >  0 then
temp  = temp ||  symbol;
else  temp = temp  || terminal;
temp  = temp ||  """";
end;
else  do;
temp  = "operator symbol  """;
temp  = temp ||  terminal;
temp  = temp ||  """";
end;
return  (temp);
end;
end  get_terminal;
```

```
/*  BEGIN INCLUDE FILE  .....  calc_s.incl.pl1 .....   06/24/76 J Falksen  */

scanner:   proc;

dcl  addr builtin;
dcl  alpha char (53)  internal static options  (constant)
init  ("abcdefghijklmnopqrstuvwxyz_ABCDEFGHIJKLMNOPQRSTUVWXYZ");

dcl  alphanumeric char (63)  internal static options  (constant)
init  ("abcdefghijklmnopqrstuvwxyz_0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ");
dcl  char8 char (8);

dcl  conversion condition;
dcl  convert builtin;
dcl  cu_$cp entry (ptr,  fixed bin (21),  fixed bin (35));

dcl  divide builtin;
dcl  exp_op_code fixed bin  internal static options  (constant) init (9);
dcl  flb float bin  (27);

dcl  hbound builtin;
dcl  index builtin;
dcl  lbound builtin;

dcl  mult_op_code fixed bin  internal static options  (constant) init (4);
dcl  next_char char (1)  defined (input) position  (next_char_pos);
dcl  one_char_ops char (8)  internal static options  (constant) init ("=+-*/()

");
dcl  RW (12:21) char  (8) internal static  options (constant)
init  ("abs", "atan", "cos",  "e", "list", "ln",  "log", "pi", "sin",  "tan");

dcl  real_code fixed bin  internal static options  (constant) init (10);
dcl  symbol_code fixed bin  internal static options  (constant) init (11);
dcl  substr builtin;

dcl  third_next_char char (1)  defined (input) position  (next_char_pos+2);
dcl  verify builtin;


MORE:
do  while (next_char_pos >  input_length);

call  get_line;
if  input = "."
then  do;

call  ioa_ ("^a", my_name);
input_length  = 0;
```

```
end;

if  input_length > 2  then
if  substr (input, 1,  2) = ".."
then  do;

call  cu_$cp (addr (third_next_char),  input_length-2, code);
input_length  = 0;
end;

if  input = quit_arg
then  do;
lstk.symptr  (lookahead_put) = input_ptr;

lstk.symlen  (lookahead_put) = 0;
lstk.file  (lookahead_put) = 0;
lstk.line  (lookahead_put) = line_number;

lstk.symbol  (lookahead_put) = 0;
return;
end;

end;
lstk.symptr  (lookahead_put) = addr  (next_char);
lstk.symlen  (lookahead_put) = 0;

lstk.file  (lookahead_put) = 0;
lstk.line  (lookahead_put) = line_number;
if  index (alpha, next_char)  ^= 0

then  do;
i  = verify (substr  (input, next_char_pos, input_length-next_char_pos+1),
alphanumeric)-1;

if  i < 0  then
i  = input_length-next_char_pos+1;
char8  = substr (input,  next_char_pos, i);

next_char_pos  = next_char_pos+i;
lb  = lbound (RW,  1);
ub  = hbound (RW,  1);

do  while (lb <=  ub);
i  = divide (ub+lb,  2, 17, 0);
if  RW (i) =  char8

then  do;
lstk.symbol  (lookahead_put) = i;
return;

end;
if  RW (i) <  char8 then
```

```
lb  = i+1;

else  ub = i-1;
end;
do  i = 1  to sym_num;

if  sym_.name (i) =  char8
then  goto found_sym;
end;

i,  sym_num = sym_num+1;
sym_.name  (sym_num) = char8;
sym_.val  (sym_num) = 0.0;

found_sym:
lstk.def  (lookahead_put) = addr  (sym_ (i));
lstk.symbol  (lookahead_put) = symbol_code;

return;
end;
else  do;

i  = verify (substr  (input, next_char_pos, input_length-next_char_pos+1),
"0123456789.")-1;
if  i < 0  then

i  = input_length-next_char_pos+1;
if  i > 0
then  do;

if  substr (input, next_char_pos+i,  1) = "e"
then  do;
i  = i+1;

if  substr (input, next_char_pos+i,  1) = "+"
|  substr (input, next_char_pos+i,  1) = "-"
then  i = i+1;

i  = i +  verify (substr (input,  next_char_pos+i, next_char_pos+i+1),
"0123456789")-1;
end;

on  conversion begin;
msg  = "missing operator";
goto  error;

end;
flb  = convert (flb,  substr (input, next_char_pos,  i));
lstk.value  (lookahead_put) = flb;

lstk.symbol  (lookahead_put) = real_code;
lstk.symlen  (lookahead_put) = i;
```

```
next_char_pos  = next_char_pos+i;

return;
end;
else  do;

i  = index (one_char_ops,  next_char);
if  i ^= 0
then  do;

lstk.symbol  (lookahead_put) = i;
next_char_pos  = next_char_pos+1;
if  i = mult_op_code  then

if  next_char = "*"
then  do;
lstk.symbol  (lookahead_put) = exp_op_code;

next_char_pos  = next_char_pos+1;
end;
return;

end;
end;
end;

if  substr (input, next_char_pos,  1) = "  "
then  do;
next_char_pos  = next_char_pos+1;

goto  MORE;
end;
msg  = "illegal char  ";

msg  = msg ||  substr (input, next_char_pos,  1);
goto  error;


get_line:   proc;
dcl  code fixed bin  (35);
dcl  cu_$arg_ptr_rel entry (fixed  bin, ptr, fixed  bin (21), fixed  bin (35), ptr);

dcl  (error_table_$end_of_info, error_table_$long_record) fixed  bin (35) external  static;
dcl  iox_$get_line entry (ptr,  ptr, fixed bin  (21), fixed bin  (21), fixed bin  (35));
dcl  iox_$user_input ptr ext  static;

dcl  k fixed bin  (21);
dcl  length builtin;
line_number  = line_number+1;

next_char_pos  = 1;
if  number_of_args ^= 0  then
```

```
3 if  current_arg < number_of_args

3 then  do;
3 current_arg  = current_arg+1;
3 call  cu_$arg_ptr_rel (current_arg, input_ptr,  input_length,

3 code,  arg_list_ptr);
3 if  code ^= 0  then
3 go  to bail_out;

3 end;
3 else  if current_arg =  number_of_args
3 then  do;

3 current_arg  = current_arg+1;
3 input_ptr  = addr (newline);
3 input_length  = length (newline);

3 end;
3 else  do;
3 input_ptr  = addr (quit_arg);

3 input_length  = length (quit_arg);
3 end;
3 else  do;

3 input_length  = 0;
3 read_line:
3 call  iox_$get_line (iox_$user_input,

3 addr  (next_char), buffer_length-input_length, k,  code);
3 input_length  = input_length+k;
3 if  code = error_table_$long_record

3 then  do;
3 buffer_length  = buffer_length+200;
3 allocate  buffer set (buffer_ptr);

3 substr  (buffer, 1, input_length)  = input;
3 free  input;
3 input_ptr  = buffer_ptr;

3 next_char_pos  = input_length+1;
3 goto  read_line;
3 end;

3 if  code = error_table_$end_of_info
3 then  do;
3 input_ptr  = addr (quit_arg);

3 input_length  = length (quit_arg);
3 end;
```

```
3 196 end;

3 197 next_char_pos  = 1;
3 198 return;
3 199 end  get_line;

3 200 end  scanner;
3 201
3 202 /*  END INCLUDE FILE  .....  calc_s.incl.pl1 .....   */

1 315
```

```
calc2_:   proc (prod_no);

/*  SEMANTICS SEGMENT calc2_.incl.pl1

* Generated  by:  Prange.SLANG.a using  LALR 7.0 of  Friday, September 17,  1982
* Generated  at:  TCO 68/80  Multics Billerica, Ma.
* Generated  on:  09/18/82 1408.0  edt Sat

* Generated  from:  >user_dir_dir>SLANG>Prange>stb>calc.s::calc2_.lalr
**/

dcl  prod_no fixed bin  parameter;

go  to prod (prod_no);


/*  -order =
* +

* -
* *
* /

* (
* )
* <nl>

* **
* <real>
* <symbol>

* abs
* atan
* cos

* e
* list
* ln

* log
* pi
* sin

* tan
*-tl
*-table calc2_t_.incl.pl1

*-sem calc2_.incl.pl1
*-production
```

```
*-parse */

dcl  (abs, atan, cos,  log, log10, sin,  tan) builtin;
/*  <calc> ::= <line...>  | !  */
/*  <line...> ::= <line>  |

* <line...>  <line>!  */
/*  <line> ::= list  <nl> |
* <symbol>  = <expression> <nl>  |

* <expression>  <nl> |
* <nl>!   */
prod  (5):

do  i = sym_num  to 1 by  -1;
call  ioa_ ("^8a =  ^f", sym_.name (i),  sym_.val (i));
end;

return;
prod  (6):
lstk.def  (ls_top-3) -> sym.val  = lstk.value (ls_top-1);

return;
prod  (7):
call  ioa_ ("= ^f",  lstk.value (ls_top-1));

return;
/*  <expression> ::= <term>  |
* <expression>  + <term> |

* <expression>  - <term> !   */
prod  (10):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) + lstk.value  (ls_top);

return;
prod  (11):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) - lstk.value  (ls_top);

return;
/*  <term> ::= <factor>  |
* <term>  * <factor> |

* <term>  / <factor> !   */
prod  (13):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) * lstk.value  (ls_top);

return;
prod  (14):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) / lstk.value  (ls_top);

return;
/*  <factor> ::= <primary>  |
```

```
<factor>  ** <primary>!  */

prod  (16):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) ** lstk.value  (ls_top);
return;

/*  <primary> ::= <reference>  |
+  <primary> |
-  <primary> |

(<expression>)  !  */
prod  (18):
lstk.value  (ls_top-1) = lstk.value  (ls_top);

return;
prod  (19):
lstk.value  (ls_top-1) = -lstk.value  (ls_top);

return;
prod  (20):
lstk.value  (ls_top-2) = lstk.value  (ls_top-1);

return;
/*  <reference> ::= <real>  |
<symbol>  |

e  |
pi  |
sin  (<expression>) |

cos  (<expression>) |
tan  (<expression>) |
atan  (<expression>) |

abs  (<expression>) |
ln  (<expression>) |
log  (<expression>) !  */

prod  (22):
lstk.value  (ls_top) = lstk.def  (ls_top) -> sym.val;
return;

prod  (23):
lstk.value  (ls_top) = 2.71828182845904523536;
return;

prod  (24):
lstk.value  (ls_top) = 3.14159265358979323846;
return;

prod  (25):
lstk.value  (ls_top-3) = sin  (lstk.value (ls_top-1));
```

```
return;

prod  (26):
lstk.value  (ls_top-3) = cos  (lstk.value (ls_top-1));
return;

prod  (27):
lstk.value  (ls_top-3) = tan  (lstk.value (ls_top-1));
return;

prod  (28):
lstk.value  (ls_top-3) = atan  (lstk.value (ls_top-1));
return;

prod  (29):
lstk.value  (ls_top-3) = abs  (lstk.value (ls_top-1));
return;

prod  (30):
lstk.value  (ls_top-3) = log  (lstk.value (ls_top-1));
return;

prod  (31):
lstk.value  (ls_top-3) = log10  (lstk.value (ls_top-1));
return;


end  calc2_;
```

```
parse_stack_overflow:   proc;
dcl  ltrim builtin;
dcl  omega picture "zzzzz9";

omega  = hbound (lstk,  1);
call  set_line_id (lookahead_get);
recov_msg  = recov_msg ||  "exceeded ";
recov_msg  = recov_msg ||  ltrim (omega);
recov_msg  = recov_msg ||
"  entries of the  parser's lexical/parse stack.   Parser cannot continue.";
call  print_recov_msg;
code  = stack_overflow;
goto  parse_done;
end  parse_stack_overflow;

set_line_id:   proc (lookahead_use);

dcl  lookahead_use fixed bin  parameter;
dcl  omega picture "------";

dcl  ltrim builtin;

recov_msg  = "ERROR on  line ";
if  lstk.file (lookahead_get) ^=  0
then  do;
omega  = lstk.file (lookahead_use);
recov_msg  = recov_msg ||  ltrim (omega);
recov_msg  = recov_msg ||  "-";
end;
omega  = lstk.line (lookahead_use);
recov_msg  = recov_msg ||  ltrim (omega);
recov_msg  = recov_msg ||  ":  ";
return;
end  set_line_id;
```

```
print_recov_msg:   proc;
dcl  addr builtin;
dcl  code fixed bin  (35);
dcl  iox_$put_chars entry (ptr,  ptr, fixed bin  (21), fixed bin  (35));
dcl  iox_$user_output external static  ptr;
dcl  length builtin;
dcl  newline char (1)  internal static options  (constant) init ("
");
dcl  substr builtin;
recov_msg  = recov_msg ||  newline;
call  iox_$put_chars (iox_$user_output, addr  (substr (recov_msg, 1,  1)),
length  (recov_msg), code);
return;
end  print_recov_msg;
end  calc2_p;
end calc2;
```

SOURCE FILES USED IN THIS COMPILATION.

```
 LINE NUMBER DATE  MODIFIED NAME PATHNAME
         0 09/18/82 1456.4  calc2.pl1 >udd>SLANG>Prange>stb>calc2.pl1 79  1 09/18/82 1421.7  calc2_p.incl.pl1 >udd>SLANG>Prange>s
>>calc2_p.incl.pl1
                 1-16
                      2
                        09/18/82
                                 1410.7
                                       calc2_t_.incl.pl1
 >udd>SLANG>Prange>stb>calc2_t_.incl.pl1    1-315    3    09/18/82    1457.3    calc_s.incl.pl1    >udd>SLANG>Prange>stb>calc_s.in
:.pl1
        1-317
                4
                    09/18/82
                                1410.3
                                        calc2_.incl.pl1

 >udd>SLANG>Prange>stb>calc2_.incl.pl1
```

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$                                                              $
$  Requested 10/04/82  1405.1 edt Mon                          $
$  Output    10/04/82  1431.0 edt Mon                          $
$                                                              $
$  Output mode ctl_char                                        $
$  x9700 queue 3            vm370.x9700                        $
$  Forms compose,^h,1s                                         $
$  25 original pages at $0.065 per page                 1.63   $
$  3 duplicate copies at $0.03 per page                 2.25   $
$  895 lines                                                   $
$                                                              $
$  Charge to Margulies.Multics.a                        3.88   $
$    Rate structure default.                                   $
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

```
      *          *     *******               *********  *      **  ********
     **         **    *********              *********  **     **  *********
    ***        ***    **     **              **         ***    **  **     **
   ****       ****    **                     **         ****   **  **     **
  ** **      ** **    ********     *******   ******     ** **  **  **     **
 **  **     **  **    *********    *******   ******     **  ** **  **     **
*********  *********  **     **              **         **   ****  **     **
*********  *********  **     **              **         **    ***  **     **
     **         **    *********              *********  **     **  *********
     **         **     *******               *********  **      *  ********
```

RSCS	RSCS	USERID	ORIGIN
MULTICS	IPC	DISTCODE	SYSTEM
MARGULIE		FILENAME	FILETYPE
10/04/82	14:33:07	FILE CREATION DATE	
2311	A	SPOOLID	CLASS
10/04/82	14:57:33	FILE PRINT DATE	
NX	003	FORM	DEVICE

PRINT RATE: FIRST COPY IS 6.5 CENTS/PAGE, SUBSEQUENT COPIES ARE 3.0 CENTS/PAGE

PRINT COST: $4.34 FOR 4 COPIES OF 28 PAGE REPORT

```
*********  *      **  ********                     *          *     *******
*********  **     **  *********                   **         **    *********
**         ***    **  **     **                  ***        ***    **     **
**         ****   **  **     **                 ****       ****    **
******     ** **  **  **     **    *******     ** **      ** **    ********
******     **  ** **  **     **    *******    **  **     **  **    *********
**         **   ****  **     **              *********  *********  **     **
**         **    ***  **     **              *********  *********  **     **
*********  **     **  *********                   **         **    *********
*********  **      *  ********                    **         **     *******
```

>no_backup_dir_dir>Multics>Margulies>mcr-mtb>lalr>lalr.landscape.mtb.compout

```
 ******     ****     ******   **              **  **    **
**     *     **     **     *  **              **  ***  ***
**           **     **        **             **   ** ** **                                           ****        **
**           **      ******   **            **    ** ** **   ******   * ***      ******   **    **     **                ******
**           **           **  **           **     **    **        **  *** **    **        **    **     **       ***     **    **
**           **           **  **          **      **    **   *******  **        **  ****  **    **     **        **     *******
**     *     **     *     **  **         **       **    **  **    **  **        **    **  **    **     **        **     **
 ******     ****     ******   ********  **        **    **   *******  **         ******    *******     ****     ****     ******
                                        **
```

10/04/82 1431.0 edt Mon x9700 vm370.x9700 MIT, Cambridge, Mass.

```
                                                                                      **
                                                                                      **
 ****                ****                          ****                               **
   **      ******      **     * ***                  **      ******   * *****   *******   ******    *******   ******   *******
   **           **     **     *** **                 **           **  **    **  **    **  **        **              **  **    **
   **      *******     **     **                     **      *******  **    **  **    **   ******   **         *******  **    **
   **     **    **     **     **         **          **     **    **  **    **  **    **        **  **        **    **  *******
   ****    *******     ****   **         **          ****    *******  **    **   *******   ******    *******   *******  **
                                                                                                                        **
```

>no_backup_dir_dir>Multics>Margulies>mcr-mtb>lalr>lalr.landscape.mtb.compout

COMPILATION LISTING OF SEGMENT calc2

Compiled by: Multics PL/I Compiler, Release 27b, of September 15, 1981
Compiled at: TCO 68/80 Multics Billerica, Ma.
Compiled on: 09/18/82 1457.2 edt Sat

Options: optimize table map

```
calc2:  proc;

/* Version  of calc using  LALR.  */

dcl arg_list_ptr  ptr;
dcl buffer  char (buffer_length) based  (buffer_ptr);
dcl buffer_length  fixed bin (21);
dcl buffer_ptr  ptr;
dcl cleanup  condition;
dcl code  fixed bin (35);
dcl com_err_  entry options (variable);
dcl cu_$arg_count  entry (fixed bin,  fixed bin (35));
dcl cu_$arg_list_ptr  entry (ptr);
dcl current_arg  fixed bin;
dcl input  char (input_length) based  (input_ptr);
dcl input_length  fixed bin (21);
dcl input_ptr  ptr;
dcl ioa_  entry options (variable);
dcl line_number  fixed bin;
dcl msg  char (100) varying;
dcl my_name  char (5) internal  static options (constant)  init ("calc2");
dcl newline  char (1) internal  static options (constant)  init ("
");
dcl next_char_pos  fixed bin;
dcl null  builtin;
dcl number_of_args  fixed bin;
dcl quit_arg  char (2) internal  static options (constant)  init ("q
");
dcl 1  sym based like  sym_;
dcl 1  sym_ (200),
2 name  char (8),
2 val  float bin (27);
dcl sym_num  fixed bin;
```

```
call cu_$arg_count  (number_of_args, code);
if code  ^= 0
then do;

bail_out:
call com_err_  (code, my_name);
return;

end;
call cu_$arg_list_ptr  (arg_list_ptr);
current_arg =  0;

sym_num =  0;
line_number =  0;
buffer_ptr =  null ();

if number_of_args  = 0
then do;
on cleanup  go to exit;

buffer_length =  200;
allocate buffer  set (buffer_ptr);
input_ptr =  buffer_ptr;

end;
retry:
next_char_pos =  1;

input_length =  0;
call calc2_p;
if code  ^= 0 then

if number_of_args  = 0 then
go to  retry;
exit:

if buffer_ptr  ^= null ()  then
free buffer;
return;

error:
call ioa_  ("^a", msg);
if number_of_args  = 0 then

go to  retry;
else go  to exit;

trn:  entry;
db_sw =  "1"b;
```

```
return;

trf:  entry;
db_sw =  "0"b;

return;

dcl  db_sw bit (1)  internal static init  ("0"b);

/*  Recovery failed.  */
dcl  syntax_error fixed bin  (35) internal static  options (constant) init  (1);
/*  Parse stack underflow  or local recovery  encountered

* impossible  conditions.  Both caused  by bad DPDA.   */
dcl  logic_error fixed bin  (35) internal static  options (constant) init  (2);
/*  Parse, lexical, or  lookahead stack overflow.   */

dcl  stack_overflow fixed bin  (35) internal static  options (constant) init  (3);
/*  Unrecognized table type  in the DPDA.   */
dcl  unrecognized_state fixed bin  (35) internal static  options (constant) init  (4);

calc2_p:   proc;

/*  Parser for tables  created by LALR.   */



/*  BEGIN INCLUDE FILE  .....  calc2_t_.incl.pl1 .....

*
*
*SCANNER AND  PARSER TABLES FROM  SEGMENT

* >user_dir_dir>SLANG>Prange>stb>calc2_.grammar
*
* Generated  by:  Prange.SLANG.a using  LALR 7.0 of  Friday, September 17,  1982

* Generated  at:  TCO 68/80  Multics Billerica, Ma.
* Generated  on:  09/18/82 1408.0  edt Sat
* Generated  from:  >user_dir_dir>SLANG>Prange>stb>calc.s::calc2_.lalr */


dcl  1 calc2_t_$terminals_list external  static,
2  terminals_list_size fixed bin,

2  terminals_list (21),
3  position fixed bin  (18) unsigned unaligned,
3  length fixed bin  (18) unsigned unaligned;


dcl  1 calc2_t_$terminal_characters external  static,
```

```
2 19 2  terminal_characters_length fixed bin,

2 20 2  terminal_characters char (55);
2 21
2 22 dcl  1 calc2_t_$dpda external  static,

2 23 2  dpda_size fixed bin,
2 24 2  dpda (214),
2 25 3  (v1, v2) fixed  bin (17) unaligned;

2 26
2 27 dcl  1 calc2_t_$skip external  static,
2 28 2  skip_size fixed bin,

2 29 2  skip (2),
2 30 3  (v1, v2) fixed  bin (17) unaligned;
2 31

2 32 dcl  1 calc2_t_$standard_prelude external  static,
2 33 2  standard_prelude_length fixed bin,
2 34 2  standard_prelude char (0);

2 35
2 36 dcl  1 calc2_t_$production_names external  static,
2 37 2  production_names_size fixed bin,

2 38 2  production_names (31) fixed  bin (17) unaligned;
2 39
2 40 dcl  1 calc2_t_$variables_list external  static,

2 41 2  variables_list_size fixed bin,
2 42 2  variables_list (8),
2 43 3  (position, length) fixed  bin (18) unsigned  unaligned;

2 44
2 45 dcl  1 calc2_t_$variable_characters external  static,
2 46 2  variable_characters_length fixed bin,

2 47 2  variable_characters char (67);
2 48
2 49 /*  END INCLUDE FILE  .....  calc2_t_.incl.pl1 .....   */

1 16
1 17 dcl  1 lstk (-1:50),
1 18 /*  -1:-1 is the  lookahead stack (FIFO)  */

1 19 /*  1:50 is the  lexical stack (LIFO)  */
1 20 2  symptr ptr, /*  pointer to symbol  (must be valid)  */
1 21 2  symlen fixed bin,  /* length of  symbol (may be  0) */

1 22 2  line_id aligned, /*  identification of line  where symbol begins  */
1 23 3  file fixed bin  (17) unaligned, /*  the include file  number */
```

```
3  line fixed bin  (17) unaligned, /*  the line number  within the include  file */

2  symbol fixed bin,  /* encoding of  the symbol */
2  value float bin  (27),
2  def ptr;

dcl  1 lookahead (-1:50)  defined lstk like  lstk;
dcl  abs builtin;
dcl  current_state fixed bin;  /* number of  current state */

dcl  current_table fixed bin;  /* number of  current table */
dcl  1 db_data unaligned,
2  flag char (1),  /* * means  stacked */

2  state picture "zzz9",
2  top picture "zzz9",
2  filler char (2),

2  type char (6),
2  data char (100);
dcl  db_item char (117)  defined (db_data);

dcl  db_separator char (1);
dcl  divide builtin;
dcl  hbound builtin;

dcl  i fixed bin;  /* temp */
dcl  ioa_$nnl entry options  (variable);
dcl  lb fixed bin;

dcl  ls_top fixed bin  defined parse_stack_top; /*  location of the  top of the  lexical stack */
dcl  lookahead_count fixed bin;  /* number of  terminals in lookahead  stack */
dcl  lookahead_get fixed bin  internal static options  (constant) init (-1);

dcl  lookahead_put fixed bin  internal static options  (constant) init (-1);
dcl  next_state fixed bin;  /* number of  next state */
dcl  parse_stack (50) fixed  bin aligned; /*  parse stack */

dcl  parse_stack_top fixed bin;  /* location of  the top of  the parse stack  */
dcl  production_number fixed bin;  /* APPLY production  number */
dcl  recov_msg char (250)  varying;

dcl  t fixed bin;
dcl  test_state fixed bin;  /* top state  from parse stack  during look back  lookups */
dcl  test_symbol fixed bin  defined lstk.symbol (-1);  /* encoding of  current symbol */

dcl  ub fixed bin;
dcl  unspec builtin;
```

```
current_state  = 1;
parse_stack_top  = 0;
lookahead_count  = 0;
unspec  (lstk) = ""b;
code  = 0; /*  Preset the status  code.  */

/*  The parsing loop.   */
NEXT:
if  current_state = 0
then  do;
parse_done:
return;
end;
current_table  = current_state;
db_item  = "";
db_data.state  = current_state;
db_data.top  = parse_stack_top;
goto  CASE (dpda.v1 (current_table));

CASE  (10):  /* Obsolete  -- Lookahead 1  (sometimes called read  without
stacking)  with shared transition  table.  */

CASE  (2):  /* Read  and stack and/or  lookahead 1 (sometimes  called
read  without stacking) with  shared transition table.
(Read  transitions to state  S are coded  as +S while
lookahead  transitions to state  S are coded  -S.)  */
current_table  = dpda.v2 (current_table);

CASE  (0):  /* Read  and stack and/or  lookahead 1 with  neither a
default  transition nor a  marked symbol transition.   */
CASE  (9):  /* Obsolete  -- Lookahead 1  (sometimes called
read  without stacking).  */
CASE  (15):  /* Read  and stack and/or  lookahead 1 with
a  default transition.  */
CASE  (17):  /* Read  and stack and/or  lookahead 1 with  the table
continued  at another state.   */

if  lookahead_count <= 0  /* Make sure  a symbol is  available.  */
```

```
then  do;
call  scanner;
lookahead_count  = lookahead_count+1;
end;
search_table:
/*  Look current symbol  up in the  read list.  */
lb  = current_table+1;
ub  = current_table+dpda.v2 (current_table);
do  while (lb <=  ub);
i  = divide (ub+lb,  2, 17, 0);
if  dpda.v1 (i) =  test_symbol
then  do;
next_state  = dpda.v2 (i);
goto  got_symbol;
end;
else  if dpda.v1 (i)  < test_symbol then
lb  = i+1;
else  ub = i-1;
end;
if  dpda.v1 (current_table+1) <  0 then
if  dpda.v1 (current_table+1) =  -1
then  do;
current_state  = -dpda.v2 (current_table+1);
if  db_sw
then  do;
db_data.type  = "LK01D";
db_data.data  = get_terminal (lookahead_get);
call  ioa_$nnl ("^a^/", db_item);
end;
goto  NEXT;
end;
else  do;
current_table  = dpda.v2 (current_table+1);
goto  search_table;
end;

if  db_sw then
call  ioa_$nnl (" ^4i  ", current_state);
call  set_line_id (lookahead_get);
```

```
recov_msg  = recov_msg ||  "at ";
recov_msg  = recov_msg ||  get_terminal (lookahead_get);
recov_msg  = recov_msg ||  ".";
call  print_recov_msg;
code  = syntax_error;
go  to parse_done;

got_symbol:
if  db_sw then
db_data.data  = get_terminal (lookahead_get);
if  next_state < 0
then  do; /* This  is a lookahead  transition.  */
db_data.type  = "LK01";
current_state  = -next_state;
end;
else  do; /* This  is a read  transition.  */
db_data.type  = "READ";
db_data.flag  = "*";
if  parse_stack_top >= hbound  (parse_stack, 1) then
call  parse_stack_overflow;
parse_stack_top  = parse_stack_top+1;
parse_stack  (parse_stack_top) = current_state;  /* Stack the  current state.  */
unspec  (lstk (parse_stack_top)) =  unspec (lookahead (lookahead_get));
lookahead_count  = 0;
current_state  = next_state;
end;
if  db_sw then
call  ioa_$nnl ("^a^/", db_item);
goto  NEXT;

CASE  (3):  /* Multiple  lookahead (k >  1) with shared  look table.  */
CASE  (1):  /* Multiple  lookahead (k >  1) without default  transition.  */
CASE  (14):  /* Multiple  lookahead (k >  1) with default  transition.  */
CASE  (16):  /* Multiple  lookahead (k >  1) with the  table
 continued  at another state.   */

CASE  (7):  /* Obsolete  state type --  Skip table.  */
CASE  (8):  /* Obsolete  state type --  Skip recovery adjust  table.  */

```

```
CASE  (4):  /* Apply  by rule and  alternative with lookback  table.  */

CASE  (5):  /* Apply  by rule and  alternative without lookback.   */
CASE  (6):  /* Apply  by rule and  alternative with shared  lookback table.  */
call  set_line_id (lookahead_get);

recov_msg  = recov_msg ||  "Unrecognized DPDA state  encountered -- Parse  fails.";
call  print_recov_msg;
code  = unrecognized_state;

go  to parse_done;

CASE  (13):  /* Apply  by production with  shared lookback table.   */

current_table  = dpda.v2 (current_state+2);
CASE  (11):  /* Apply  by production with  lookback table.  */
CASE  (12):  /* Apply  by production without  lookback.  */

production_number  = dpda.v1 (current_state+2);
if  production_number > 0  then
call  calc2_ (production_number);

if  db_sw
then  begin;
dcl  production_name char (variables_list.length  (-production_names (abs (production_number))))

defined  (variable_characters)
position  (variables_list.position (-production_names (abs  (production_number))));
db_data.type  = "APLY";

db_data.data  = "(";
if  dpda.v1 (current_state+1) <  0 then
db_data.flag  = "*";

call  ioa_$nnl ("^a^i", db_item,  production_number);
if  production_names_size > 0  then
call  ioa_$nnl (" ^a",  production_name);

call  ioa_$nnl (")^-sd =  ^i ", dpda.v1  (current_state+1));
if  dpda.v1 (current_state+1) >  0
then  do;

db_separator  = "(";
do  t = parse_stack_top  to parse_stack_top-dpda.v1 (current_state+1)+1  by -1;
call  ioa_$nnl ("^1a^d", db_separator,  parse_stack (t));

db_separator  = "";
end;
call  ioa_$nnl (")");

end;
call  ioa_$nnl ("^/");
```

```
end;
/*  Check for an  apply of an  empty production.
In  this case the  apply state number  must be
pushed  onto the parse  stack.  (Reference
LaLonde,  W.  R.:  An  efficient LALR Parser  Generator.
Tech.   Report CSRG-2, 1971,  pp.  34-35.)  */
if  dpda.v1 (current_state+1) <  0
then  do;
if  parse_stack_top >= hbound  (parse_stack, 1) then
call  parse_stack_overflow;
parse_stack  (parse_stack_top+1) = current_state;
end;
/*  Delete lexical &  parse stack entries.   */
parse_stack_top  = parse_stack_top-dpda.v1 (current_state+1);
if  parse_stack_top <= 0
then  do;
call  set_line_id (lookahead_get);
recov_msg  = recov_msg ||  "lexical/parse stack empty  -- Parse fails.";
call  print_recov_msg;
code  = logic_error;
go  to parse_done;
end;
lb  = current_table+3;
ub  = current_table+dpda.v2 (current_table);
test_state  = parse_stack (parse_stack_top);
do  while (lb <=  ub);
i  = divide (ub+lb,  2, 17, 0);
if  dpda.v1 (i) =  test_state
then  do;
current_state  = dpda.v2 (i);
goto  NEXT;
end;
else  if dpda.v1 (i)  < test_state then
lb  = i+1;
else  ub = i-1;
end;
current_state  = dpda.v2 (current_table+2);
goto  NEXT;
```

```
get_terminal:   proc (lstk_index) returns  (char (100) varying);
dcl  lstk_index fixed bin  parameter;
dcl  alphanumeric (0:511) bit  (1) unaligned internal  static options (constant)  init (
(32)  (1) "0"b, /*  control characters */
(4)  (1) "0"b, /*  SP !  "  # */
"1"b,  /* $ */
(11)  (1) "0"b, /*  % & '  ( ) *  + , -  .  / */
(10)  (1) "1"b, /*  digits */
(7)  (1) "0"b, /*  :; < =  > ?  @  */
(26)  (1) "1"b, /*  upper case letters  */
(4)  (1) "0"b, /*  [ \ ]  ^ */
"1"b,  /* underscore */
"0"b,  /*   */
(26)  (1) "1"b, /*  lower case letters  */
(5)  (1) "0"b, /*  { | }  ¬ DEL */
(384)  (1) "0"b); /*  rest of 9-bit  ASCII code set  */
if  lstk.symbol (lstk_index) =  0 then
return  ("end-of-information");
else  begin;
dcl  temp char (100)  varying;
dcl  (length, min, rank,  substr) builtin;
dcl  symbol char (min  (50, lstk.symlen (lstk_index)))  based (lstk.symptr (lstk_index));
dcl  terminal char (terminals_list.length  (lstk.symbol (lstk_index)))
defined  (terminal_characters)
position  (terminals_list.position (lstk.symbol (lstk_index)));
if  length (terminal) >  2
&  substr (terminal, 1,  1) = "<"
&  substr (terminal, length  (terminal), 1) =  ">"
then  do;
temp  = substr (terminal,  2, length (terminal)-2);
if  length (symbol) >  0
then  do;
temp  = temp ||  " ";
if  substr (symbol, 1,  1) = """"
|  substr (symbol, 1,  1) = "'"  then
temp  = temp ||  symbol;
else  do;
```

```
1 293 temp  = temp ||  """";

1 294 temp  = temp ||  symbol;
1 295 temp  = temp ||  """";
1 296 end;

1 297 end;
1 298 end;
1 299 else  if alphanumeric (rank  (substr (terminal, 1,  1)))

1 300 then  do;
1 301 temp  = "reserved word  """;
1 302 if  length (symbol) >  0 then

1 303 temp  = temp ||  symbol;
1 304 else  temp = temp  || terminal;
1 305 temp  = temp ||  """";

1 306 end;
1 307 else  do;
1 308 temp  = "operator symbol  """;

1 309 temp  = temp ||  terminal;
1 310 temp  = temp ||  """";
1 311 end;

1 312 return  (temp);
1 313 end;
1 314 end  get_terminal;
```

```
/*  BEGIN INCLUDE FILE  .....  calc_s.incl.pl1 .....   06/24/76 J Falksen  */

scanner:   proc;

dcl  addr builtin;
dcl  alpha char (53)  internal static options  (constant)
init  ("abcdefghijklmnopqrstuvwxyz_ABCDEFGHIJKLMNOPQRSTUVWXYZ");

dcl  alphanumeric char (63)  internal static options  (constant)
init  ("abcdefghijklmnopqrstuvwxyz_0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ");
dcl  char8 char (8);

dcl  conversion condition;
dcl  convert builtin;
dcl  cu_$cp entry (ptr,  fixed bin (21),  fixed bin (35));

dcl  divide builtin;
dcl  exp_op_code fixed bin  internal static options  (constant) init (9);
dcl  flb float bin  (27);

dcl  hbound builtin;
dcl  index builtin;
dcl  lbound builtin;

dcl  mult_op_code fixed bin  internal static options  (constant) init (4);
dcl  next_char char (1)  defined (input) position  (next_char_pos);
dcl  one_char_ops char (8)  internal static options  (constant) init ("=+-*/()

");
dcl  RW (12:21) char  (8) internal static  options (constant)
init  ("abs", "atan", "cos",  "e", "list", "ln",  "log", "pi", "sin",  "tan");

dcl  real_code fixed bin  internal static options  (constant) init (10);
dcl  symbol_code fixed bin  internal static options  (constant) init (11);
dcl  substr builtin;

dcl  third_next_char char (1)  defined (input) position  (next_char_pos+2);
dcl  verify builtin;


MORE:
do  while (next_char_pos >  input_length);

call  get_line;
if  input = "."
then  do;

call  ioa_ ("^a", my_name);
input_length  = 0;
```

```
end;

if  input_length > 2  then
if  substr (input, 1,  2) = ".."
then  do;

call  cu_$cp (addr (third_next_char),  input_length-2, code);
input_length  = 0;
end;

if  input = quit_arg
then  do;
lstk.symptr  (lookahead_put) = input_ptr;

lstk.symlen  (lookahead_put) = 0;
lstk.file  (lookahead_put) = 0;
lstk.line  (lookahead_put) = line_number;

lstk.symbol  (lookahead_put) = 0;
return;
end;

end;
lstk.symptr  (lookahead_put) = addr  (next_char);
lstk.symlen  (lookahead_put) = 0;

lstk.file  (lookahead_put) = 0;
lstk.line  (lookahead_put) = line_number;
if  index (alpha, next_char)  ^= 0

then  do;
i  = verify (substr  (input, next_char_pos, input_length-next_char_pos+1),
alphanumeric)-1;

if  i < 0  then
i  = input_length-next_char_pos+1;
char8  = substr (input,  next_char_pos, i);

next_char_pos  = next_char_pos+i;
lb  = lbound (RW,  1);
ub  = hbound (RW,  1);

do  while (lb <=  ub);
i  = divide (ub+lb,  2, 17, 0);
if  RW (i) =  char8

then  do;
lstk.symbol  (lookahead_put) = i;
return;

end;
if  RW (i) <  char8 then
```

```
lb = i+1;

else ub = i-1;
end;
do i = 1 to sym_num;

if sym_.name (i) = char8
then goto found_sym;
end;

i, sym_num = sym_num+1;
sym_.name (sym_num) = char8;
sym_.val (sym_num) = 0.0;

found_sym:
lstk.def (lookahead_put) = addr (sym_ (i));
lstk.symbol (lookahead_put) = symbol_code;

return;
end;
else do;

i = verify (substr (input, next_char_pos, input_length-next_char_pos+1),
"0123456789.")-1;
if i < 0 then

i = input_length-next_char_pos+1;
if i > 0
then do;

if substr (input, next_char_pos+i, 1) = "e"
then do;
i = i+1;

if substr (input, next_char_pos+i, 1) = "+"
| substr (input, next_char_pos+i, 1) = "-"
then i = i+1;

i = i + verify (substr (input, next_char_pos+i, next_char_pos+i+1),
"0123456789")-1;
end;

on conversion begin;
msg = "missing operator";
goto error;

end;
flb = convert (flb, substr (input, next_char_pos, i));
lstk.value (lookahead_put) = flb;

lstk.symbol (lookahead_put) = real_code;
lstk.symlen (lookahead_put) = i;
```

```
next_char_pos  = next_char_pos+i;
return;
end;
else  do;
i  = index (one_char_ops,  next_char);
if  i ^= 0
then  do;
lstk.symbol  (lookahead_put) = i;
next_char_pos  = next_char_pos+1;
if  i = mult_op_code  then
if  next_char = "*"
then  do;
lstk.symbol  (lookahead_put) = exp_op_code;
next_char_pos  = next_char_pos+1;
end;
return;
end;
end;
end;
if  substr (input, next_char_pos,  1) = "  "
then  do;
next_char_pos  = next_char_pos+1;
goto  MORE;
end;
msg  = "illegal char  ";
msg  = msg ||  substr (input, next_char_pos,  1);
goto  error;

get_line:   proc;
dcl  code fixed bin  (35);
dcl  cu_$arg_ptr_rel entry (fixed  bin, ptr, fixed  bin (21), fixed  bin (35), ptr);
dcl  (error_table_$end_of_info, error_table_$long_record) fixed  bin (35) external  static;
dcl  iox_$get_line entry (ptr,  ptr, fixed bin  (21), fixed bin  (21), fixed bin  (35));
dcl  iox_$user_input ptr ext  static;
dcl  k fixed bin  (21);
dcl  length builtin;
line_number  = line_number+1;
next_char_pos  = 1;
if  number_of_args ^= 0  then
```

```
3 if  current_arg < number_of_args
3 then  do;
3 current_arg  = current_arg+1;
3 call  cu_$arg_ptr_rel (current_arg, input_ptr,  input_length,
3 code,  arg_list_ptr);
3 if  code ^= 0  then
3 go  to bail_out;
3 end;
3 else  if current_arg =  number_of_args
3 then  do;
3 current_arg  = current_arg+1;
3 input_ptr  = addr (newline);
3 input_length  = length (newline);
3 end;
3 else  do;
3 input_ptr  = addr (quit_arg);
3 input_length  = length (quit_arg);
3 end;
3 else  do;
3 input_length  = 0;
3 read_line:
3 call  iox_$get_line (iox_$user_input,
3 addr  (next_char), buffer_length-input_length, k,  code);
3 input_length  = input_length+k;
3 if  code = error_table_$long_record
3 then  do;
3 buffer_length  = buffer_length+200;
3 allocate  buffer set (buffer_ptr);
3 substr  (buffer, 1, input_length)  = input;
3 free  input;
3 input_ptr  = buffer_ptr;
3 next_char_pos  = input_length+1;
3 goto  read_line;
3 end;
3 if  code = error_table_$end_of_info
3 then  do;
3 input_ptr  = addr (quit_arg);
3 input_length  = length (quit_arg);
3 end;
```

```
end;

next_char_pos  = 1;
return;
end  get_line;

end  scanner;

/*  END INCLUDE FILE  .....  calc_s.incl.pl1 .....   */
```

```
calc2_:   proc (prod_no);

/*  SEMANTICS SEGMENT calc2_.incl.pl1
* Generated  by:  Prange.SLANG.a using  LALR 7.0 of  Friday, September 17,  1982
* Generated  at:  TCO 68/80  Multics Billerica, Ma.
* Generated  on:  09/18/82 1408.0  edt Sat
* Generated  from:  >user_dir_dir>SLANG>Prange>stb>calc.s::calc2_.lalr
**/

dcl  prod_no fixed bin  parameter;

go  to prod (prod_no);


/*  -order =
* +
* -
* *
* /
* (
* )
* <nl>
* **
* <real>
* <symbol>
* abs
* atan
* cos
* e
* list
* ln
* log
* pi
* sin
* tan
*-tl
*-table calc2_t_.incl.pl1
*-sem calc2_.incl.pl1
*-production
```

```
-parse */

dcl  (abs, atan, cos,  log, log10, sin,  tan) builtin;
/*  <calc> ::= <line...>  |  !  */
/*  <line...> ::= <line>  |

<line...>  <line>!  */
/*  <line> ::= list  <nl> |
<symbol>  = <expression> <nl>  |

<expression>  <nl> |
<nl>!   */
prod  (5):

do  i = sym_num  to 1 by  -1;
call  ioa_ ("^8a =  ^f", sym_.name (i),  sym_.val (i));
end;

return;
prod  (6):
lstk.def  (ls_top-3) -> sym.val  = lstk.value (ls_top-1);

return;
prod  (7):
call  ioa_ ("= ^f",  lstk.value (ls_top-1));

return;
/*  <expression> ::= <term>  |
<expression>  + <term> |

<expression>  - <term> !   */
prod  (10):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) + lstk.value  (ls_top);

return;
prod  (11):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) - lstk.value  (ls_top);

return;
/*  <term> ::= <factor>  |
<term>  * <factor> |

<term>  / <factor> !   */
prod  (13):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) * lstk.value  (ls_top);

return;
prod  (14):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) / lstk.value  (ls_top);

return;
/*  <factor> ::= <primary>  |
```

```
<factor>  ** <primary>!  */

prod  (16):
lstk.value  (ls_top-2) = lstk.value  (ls_top-2) ** lstk.value  (ls_top);
return;

/*  <primary> ::= <reference>  |
+  <primary> |
-  <primary> |

(<expression>)  !  */
prod  (18):
lstk.value  (ls_top-1) = lstk.value  (ls_top);

return;
prod  (19):
lstk.value  (ls_top-1) = -lstk.value  (ls_top);

return;
prod  (20):
lstk.value  (ls_top-2) = lstk.value  (ls_top-1);

return;
/*  <reference> ::= <real>  |
<symbol>  |

e  |
pi  |
sin  (<expression>) |

cos  (<expression>) |
tan  (<expression>) |
atan  (<expression>) |

abs  (<expression>) |
ln  (<expression>) |
log  (<expression>) !  */

prod  (22):
lstk.value  (ls_top) = lstk.def  (ls_top) -> sym.val;
return;

prod  (23):
lstk.value  (ls_top) = 2.71828182845904523536;
return;

prod  (24):
lstk.value  (ls_top) = 3.14159265358979323846;
return;

prod  (25):
lstk.value  (ls_top-3) = sin  (lstk.value (ls_top-1));
```

```
return;

prod (26):
lstk.value (ls_top-3) = cos (lstk.value (ls_top-1));
return;

prod (27):
lstk.value (ls_top-3) = tan (lstk.value (ls_top-1));
return;

prod (28):
lstk.value (ls_top-3) = atan (lstk.value (ls_top-1));
return;

prod (29):
lstk.value (ls_top-3) = abs (lstk.value (ls_top-1));
return;

prod (30):
lstk.value (ls_top-3) = log (lstk.value (ls_top-1));
return;

prod (31):
lstk.value (ls_top-3) = log10 (lstk.value (ls_top-1));
return;

end calc2_;
```

```
parse_stack_overflow:   proc;
dcl  ltrim builtin;
dcl  omega picture "zzzzz9";
omega  = hbound (lstk,  1);
call  set_line_id (lookahead_get);
recov_msg  = recov_msg ||  "exceeded ";
recov_msg  = recov_msg ||  ltrim (omega);
recov_msg  = recov_msg ||
"  entries of the  parser's lexical/parse stack.   Parser cannot continue.";
call  print_recov_msg;
code  = stack_overflow;
goto  parse_done;
end  parse_stack_overflow;

set_line_id:   proc (lookahead_use);
dcl  lookahead_use fixed bin  parameter;
dcl  omega picture "------";
dcl  ltrim builtin;
recov_msg  = "ERROR on  line ";
if  lstk.file (lookahead_get) ^=  0
then  do;
omega  = lstk.file (lookahead_use);
recov_msg  = recov_msg ||  ltrim (omega);
recov_msg  = recov_msg ||  "-";
end;
omega  = lstk.line (lookahead_use);
recov_msg  = recov_msg ||  ltrim (omega);
recov_msg  = recov_msg ||  ":  ";
return;
end  set_line_id;
```

```
print_recov_msg:   proc;
dcl  addr builtin;
dcl  code fixed bin  (35);

dcl  iox_$put_chars entry (ptr,  ptr, fixed bin  (21), fixed bin  (35));
dcl  iox_$user_output external static  ptr;
dcl  length builtin;

dcl  newline char (1)  internal static options  (constant) init ("
");
dcl  substr builtin;


recov_msg  = recov_msg ||  newline;
call  iox_$put_chars (iox_$user_output, addr  (substr (recov_msg, 1,  1)),

length  (recov_msg), code);
return;
end  print_recov_msg;

end  calc2_p;

end calc2;
```