To: Distribution

From: J. Falksen / Dave Ward

Date: July 9, 1976

Subject: LRK, a Translator Construction System

This MTB describes the LRK system. LRK translates a BNF-like language description into a parser for the language. The output from LRK 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. LRK has options which allow the control tables to be generated as a Multics object segment, an ALM source segment or a GMAP source segment.

The parser created by LRK (the tables along with the parser procedure) is a "bottom-up" SLR(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. LRK 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.

LRK 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. LRK assures the correspondence between what a language is published to be and the parser that "says" what the language "is".

Because of LRK'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. LRK 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 LRK. The compilation and generation listings are included at the end of this MTB. This version was run against the installed one for a few cases. The execution time of the LRK version was from 98% to 144% of that of the installed calc. The bound object size of the LRK version was 64% of that of the installed one. It took 7 1/2 hours to complete.

The following non-trivial example of the use of LRK is available for inspection on System M:

```
   >udd>m>odf>schema>mids_tis_parse_.list
   >udd>m>odf>schema>mids_tis_parse_g.list
```

This parses the subset of I-D-S/II Schema Definition Language supported by Multics Integrated Data Store.

Glossary

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.

terminal - a symbol of a language.

variable - a non-terminal of a language.

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 a set of symbols; e.g., <integer>.

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 LRK system. You do not have to master all of this information to attempt a use of LRK. Various parts are of interest only after you have tried LRK and are selecting among different approaches in using LRK to aid in the implementation of a translator.

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

1. Prepare a sample grammar, the input to lrk. (See Source format, page 4, and Grammar format, page 5, and, e.g., ted text editor).

2. Execute lrk. (See lrk, page 8).

3. Repair the grammar if it is not acceptable (scratch head). (e.g., ted text editor).

4. Test the parser by executing lrkp, after the grammar is accepted by lrk. lrk_parse, page 9).

5. If the facilities of lrk_parse are sufficient, you then supply your semantics for that environment. If desired, write a scanner following the lrk_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 3).

Consideration will be needed to accommodate error reporting and recovery. (See Error Recovery, page 6) 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 LRK must be understood. (See Unreserved Keywords, page 6)

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

Processor functions

An LRK 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 thru.

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 "-" then it contains control lines. If not, then the grammar begins with the first character.

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

```
-hash N          1 space separates the keyword from the N.
-alm
-gmap
-tl
-thl
-count
-mark X          1 space separates the keyword from the X.
-sem X           1 space separates the keyword from the X.
```

-table X	1 space separates the keyword from the X.
-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. A minumum of 1 space separates the keyword from the first terminal. Thereafter, each terminal is separated by white space. This control lasts up until the next line which begins with a "-".
-recover t t ...	This specifies terminals for skip-recovery. See Error Recovery. The format is like -order.
-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 is really a PL/I procedure. LRK will create a compileable segment from it by these steps.
1) Put /* and */ around the control portion, if present.
2) Put /* and */ around each LRK rule.
3) Replace each %%%% in the semantics with a 4-digit number of the rule which this represents.

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 the rule described has been recognized. The rules have this basic form:

<var> ::= <prod> ! <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>	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". 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 "::=".

Observe some LRK detail:

1. Rule ordering is unimportant, except that the rule that defines the "start symbol" must be physically first.
2. Ordering of productions (rule parts) is unimportant.
3. Each rule must be terminated by an exclaimation mark, "!". It is after this mark that semantic code is placed.
4. LRK reserves the use of the symbols, "<", "::=", "¦", "´" and "!". Spaces are not required except between adjacent terminal symbols, i.e., "<O>::=+¦-!" is acceptable.
5. To specify symbols involving these reserved characters and "space" characters the following escape character convention is implemented. The right apostrophe, "´", signals an escaped character. It may be followed by three octal digits, whose 9-bit value specifies the Multics ASCII character desired, or if not followed (immediately) by three octal digits, whatever character does follow is the character being escaped, i.e., "´ " and "´040" both indicate one blank character. This escape convention causes the restriction of the use of the right 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, FF, or VT.

The parsing of the LRK 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

LRK 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 dependant on your language. It is not usually an easy thing to take care of.

One simple case is in an interactive interpretor. 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 LRK compiler; local recovery and skip recovery.

Local recovery

Local recovery uses the current (unacceptable) input symbol and the next input symbol to simulate parses from this point up until the next state which reads a symbol. It then decides which action to take, if any.
Given:

```
          B is the current (bad) symbol
          N is the next symbol
          C is the current state
          R is the "next" read state
```

These are the conditions which can exist:

```
  C( N ) R( B N)    kind of error
     0      1 0     symbol leading to R is missing
     0      0 1     B is a wrong symbol (alternate is chosen)
     1      1 0     B and N are reversed in input
     1      0 x     B is an extra symbol in input
     0      0 0     recovery fails
```

The recovery trys to find a useable combination. If one exists, it is remembered but the search does not stop. If a second one is found then the search will stop and the error message can include the fact that the recovery done was not unique. The first one found is the one used. It then adjusts the look-ahead stack by either dropping a symbol, interchanging two symbols or generating a symbol.

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 lrk 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 all states which can read a skip terminal.

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 to try to find a state which could read the found terminal. If one is found, it adjusts the lexical stack top and then procedes.

Before proceding it puts the encoding for <nil> in the look-ahead stack. If the state does not contain a use of the <nil> symbol, then it is discarded and the next symbol is used.

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> ;
```

then 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:

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

then 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 thru 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: lrk

The lrk command invokes the LRK compiler to translate a segment containing the text of the *LRK* 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: lrk segment_name -list_arg- -ctl_arg-

1) segment_name — is the pathname of the LRK source segment containing the grammar to be processed. The entry portion of this pathname can contain an optional .lrk suffix.

2) list_arg — may be one or more of the following optional arguments. If the source segment is named X.lrk, then the list segment will be named Xg.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.

- -source -sc — produces a line-numbered listing of the rules of the grammar. No semantics are listed, only the rules.
- -symbols -sb — produces a listing of the terminals and variables used in the grammar.
- -list -ls — produces a "machine" listing of the DPDA resulting from the LRK execution.
- -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.
- -term — produces a listing of the terminals in encoding order, showing the encoding.
- -ss — produces source and symbols.
- -ssl — produces source, symbols, and list.

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

- -sem X — produces a semantics file named X. X must have a .pl1 suffix.
- -mark X — mark terminal X (see Unreserved keywords)
- -hash N — set the hash value of the variable and terminal tables to N.
- -table X — produces a table named X (with all suffixes removed) and an include file named X (with the supplied suffix). At present the only suffix supported is .incl.pl1. Unless this argument is supplied, the arguments below (-tl, etc.) are meaningless. The default is to produce the table as a Multics object segment.
- -tl — include the terminals list in the table.
- -thl — include the terminals list and terminals hash list in the table.
- -alm — produce the table as an alm segment X.alm. X is the name supplied in the -table parameter less all suffixes.
- -gmap — produce the table as a gmap segment X.gmap.

Options -alm and -gmap may occur together.

Names: lrk_parse, lrkp

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

Usage: lrk_parse grammar_name -source- -ctl_args-

1) grammar_name — is the pathname of the grammar. It must be without the .lrk suffix. The directory referenced must be the one containing the tables generated from lrk.

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, then it must provide for reading user_input if no source is specified, or it must report an error.

3) ctl_arg — may be one or more of the following optional arguments. (E represents an entryname; it is found according to the search rules.)

-sem E — is the entryname of a semantics procedure which corresponds to the grammar. The default semantics do nothing.

-scan E — is the entryname of a scanner procedure which corresponds to the grammar. The default scanner is explained below.

-trace — causes a trace of the parsing and error recovery action to be printed.

-print — causes each line from source to be printed (with linenumber) before starting to scan it. This is true of the default scanner. If a user scanner is supplied, then it determines whether or not printing is available.

Scanner/Semantics

lrk_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.

Usage:

```
dcl  E entry(fixed bin(24),fixed bin(24),ptr,fixed bin(24));
call E (rulen,altn,addr(lex_stack),ls_top);
```

rulen is the number of the rule completed
altn is which rule alternative was used
ls_top is the location in the lexical stack corresponding to the rightmost rule alternative symbol.

The values in lex_stack should not be modified.

2) 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$init entry(ptr,fixed bin(24),bit(1));
call E$init(input,leng,prsw);
```

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.

3) 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$E entry(ptr,fixed bin(24));
call E$E (stkp,putl);
```

stkp is a pointer to the lexical stack. The stack declaration is in lrk_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 found symbol (may be zero).
 stk.line  (putl)   linenumber where symbol begins.
 stk.symbol(putl)   encoding for the found symbol.
```

These fields may be set:

```
 stk.ptr1(putl)     pointer to user data
 stk.ptr2(putl)     pointer to user data
```

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>", and "<symbol>" 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 the terminal "<integer>" exists in the grammar, this encoding is returned.

Otherwise, if 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, then 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 if the lists contained a match at this point in the input, then the scanner moves ahead one character and tries again. If the character skipped is <= \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>", or "<string>" is processed then 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 lrk system has available a macro which can generate a skeleton parser. Once this parser is obtained, then 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 lrk_skel
        [ .sr XXX YYY ]
          ...
          .if lrk_skel
```

The first call to lrk_skel sets the default values in some variables. Then you

```
initialize
do while (^EOI);
     if READ_state then do;
          enter state number into parse stack
          if look-ahead stack empty
          then call scanner; /* puts to look-ahead stack */
          look in read-table for 1st look-ahead symbol
          if not found then ERROR
          set next state from read-table
          if look-ahead transition
          then delete 1 state from parse stack
          else move symbol from look-ahead stack
                            to lex stack
     end;
     else if LOOK_state then do; /* look ahead n */
          do until n symbols in look-ahead stack;
               call scanner; /* put to look-ahead stack */
          end;
          look in look-table for n'th look-ahead symbol
          if not found then ERROR
          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
          look in apply-table for top stacked state
          set next state from apply-table
     end;
end;
```

Figure 1. Generalized parse procedure.

adjust any of these values you wish. The second call to lrk_skel generates the parser, directed by values in the variables. The result is a segment named X.incl.pl1.

If the segment is named X.runoff then 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.

```
          .sr parameters        ""
```

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

```
          .sr db_sw             "db_sw"
```

This controls the inclusion of the trace coding and names the switch to control it. The declaration precedes the proc statement. If the value is "" then no trace coding is included.

```
          .sr lex_stack_incl    ""
          .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". It also is the level 1 name of the structure generated. If the value is "" then no include file is generated.
ls_attr is the attributes wanted on the structure in the include file.

```
          .sr lex_stack         "lex_stack"
          .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.
ls_dim is the size of the lexical stack.
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 lrk_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"
```

Remember that in quoted strings runoff requires:

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

```
          .sr la_dim            4
```

This is the size (dimension) of the look-ahead stack (FIFO). The lexical stack is declared as

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

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

```
          .sr ps_dim            100
```

This is the size of the parse stack.

```
          .sr reserved_kw       %false%
```

This controls the symbol lookup as to whether you have reserved or unreserved keywords. Can be set to %true%. 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.

```
          .sr scanner          "scanner"
          .sr sc_args          ""
          .sr sc_incl          ""
```

These specify things about the scanner procedure.
  scanner is the name of the scanner to be called.
  sc_args is the arguments to be passed to it.
  sc_incl is the name of an include file which contains the scanner. If this is specified, then an %include statement will be generated inside the parser. Then the lexical stack will be available without any include file or parameter passing necessary.

```
          .sr semantics        "semantics"
          .sr sem_args         "rulen,altn"
          .sr sem_incl         ""
```

These specify things about the semantics procedure.
  semantics is the name of the semantics procedure to be called when an apply is done.
  sem_args is the arguments to be passed to it. The default is to pass the rule number and alternative number of the apply being done.
  sem_incl is the name of an include file which contains the semantics procedure. If this is specified, then an %include statement will be generated inside the parser.

```
          .sr skip_recover     %true%
```

This determines whether or not the skip recovery mechanism is included in the parser.
  skip_recover may be set %false% if not needed.

```
          .sr max_recover      0
```

This is the upper limit on the number of local recoveries which can occur in a row. If zero, then no local recovery coding will be generated.

After this macro source is prepared it is processed by executing

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

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

Sample usage of LRK

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

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

```
          lrk log -source -symbols -terms
```

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 LRK "or" operator.

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

```
          lrk_parse log -trace
```

it will type LRKP(2.0) and then wait for you to type a statement. If you reply something like:

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

Figure 2. Basic log.lrk (grammar only)

```
^(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) pd=1 ld=0( 19)
*  37  READ  ¦
```

The first number on the line is the state number; if preceded by a "*" it means it was stacked (parse stack). The number pair following APLY is the rule/alternative being applied. If the rule is negative, then no semantics exist for it. "pd=1" means 1 element is deleted from the parse stack. "ld=0" mens 0 elements are deleted from the lexical stack. The list of numbers inside tha second "()"'s tell the states which are deleted from the parse stack.

The "¦" following the READ is the symbol read. If it is followed by a quoted string, this is the string in the source which is scanned as the named symbol.

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 lex 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.

You then build the rest of your semantics procedure around the grammar that you

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

Figure 3. Macro input, log_parse_.runoff

know is acceptable to LRK. This gives a source which looks like Figure 4. Now you run LRK again with

    lrk 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 4-digit numbers, and since this is an incl.pl1 file all rules have been converted to PL/I comments. This is done in such a way that the semantics

---

```
-sem log.incl.pl1
-parse
semantics: proc (rulen,alt);

dcl  rule fixed bin,     /* rule being applied */
     alt  fixed bin;     /* alternate being applied */

     goto rule(rulen);

<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 (alt = 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;

end;
```

Figure 4. Completed log.lrk

file line numbers and source file line numbers are identical. Figure 5, is this generated include file.

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

```
/* -sem log.incl.pl1
-parse */
semantics: proc (rulen,alt);

dcl  rule fixed bin,     /* rule being applied */
     alt  fixed bin;     /* alternate being applied */

     goto rule(rulen);

/* <log>  ::= <or> ! */
rule(0001):
     call ioa_("result is ^1b",lex_stack.val(ls_top));
     return;
/* <or>   ::= <or> '| <and> ! */;
rule(0002):
     lex_stack.val(ls_top-2) = lex_stack.val(ls_top-2)
                             | lex_stack.val(ls_top);
     return;

/* <or>   ::= <and> ! */
/* <and>  ::= <and> & <not> ! */;
rule(0004):
     lex_stack.val(ls_top-2) = lex_stack.val(ls_top-2)
                             & lex_stack.val(ls_top);
     return;

/* <and>  ::= <not> ! */
/* <not>  ::= ^ <bit>  | <bit> ! */
rule(0006):
    if (alt = 1) then
     lex_stack.val(ls_top-1) = ^ lex_stack.val(ls_top);
     return;

/* <bit>  ::= X ! */;
/* <bit>  ::= ( <or> ) ! */
rule(0008):
     lex_stack.val(ls_top-2) = lex_stack.val(ls_top-1);
     return;

end;
```

Figure 5. log.incl.pl1

```
            GENERATION LISTING OF SEGMENT log
            Processed by: LRK 2.1 of 18 June 1976
            Processed on: 06/18/76  1720.8 mst Fri
                 Options: -source

   10       <log>      ::= <or> !

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

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

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

   34       <bit>      ::= X !;
   35       <bit>      ::= ( <or> ) !
```

Figure 6. logg.list

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 10 or 27, then an empty line will follow it. If this character is a ";", as in line 14 or 34, then there is no empty line following.

Notice that the alternative on line 28 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 21 and 27 use the multiple definition form. Since each of the definitions is a separate rule, then the alternative number need not be checked (it is always 1).

Bibliography

This is a listing of many items having to to with language processing. LRK 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-Holland 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.

```
        GENERATION LISTING OF SEGMENT calc_
        Processed by: LRK 2.0e of 11 June 1976
        Processed on: 06/24/76  1125.3 mst Thu
             Options: -ssl -term -ct

  32    <calc>    ::= <line...> q <nl> | q <nl> !

  36    <line...> ::= <line> !;
  37    <line...> ::= <line...> <line> !

  38    <line>    ::= list <nl> !;
  45    <line>    ::= <symbol> = <exp> <nl> !;
  50    <line>    ::= <exp> <nl> !

  56    <nl>      ::= '012 !

  57    <exp>     ::= <exp> + <term> !;
  62    <exp>     ::= <exp> - <term> !;
  67    <exp>     ::= <term> !

  68    <term>    ::= <term> * <pwr> !;
  73    <term>    ::= <term> / <pwr> !;
  78    <term>    ::= <pwr> !

  79    <pwr>     ::= <pwr> ** <factor> !;
  84    <pwr>     ::= <factor> !

  85    <factor>  ::= <ref> !;
  86    <factor>  ::= + <ref> !;
  91    <factor>  ::= - <ref> !;
  96    <factor>  ::= ( <exp> ) !

 101    <ref>     ::= <real> !;
 102    <ref>     ::= <symbol> !;
 107    <ref>     ::= sin ( <exp> ) !;
 112    <ref>     ::= cos ( <exp> ) !;
 117    <ref>     ::= tan ( <exp> ) !;
 122    <ref>     ::= atan ( <exp> ) !;
 127    <ref>     ::= abs ( <exp> ) !;
 132    <ref>     ::= ln ( <exp> ) !;
 137    <ref>     ::= log ( <exp> ) !

***********************
*    28 Rules         *
*    30 Productions   *
*    13 Variables     *
*    30 Terminals     *
*    77 States        *
*   296 DPDA words    *
***********************
```

TERMINALS USED

SYMBOL	CODE	REFERENCES
'012	5	ref 56
(	7	ref 96 107 112 117 122 127 132 137
)	13	ref 96 107 112 117 122 127 132 137
*	11	ref 68
**	12	ref 79
+	8	ref 57 86
-	9	ref 62 91
/	10	ref 73
<real>	2	ref 101
<symbol>	1	ref 45 102
=	6	ref 45
abs	14	ref 127
atan	15	ref 122
cos	16	ref 112
list	3	ref 38
ln	17	ref 132
log	18	ref 137
q	4	ref 32 32
sin	19	ref 107
tan	20	ref 117

VARIABLES USED

<calc>	-1	def 32 32 ref
<exp>	-5	def 57 62 67 ref 45 50 57 62 96 107 112 117 122 127 132 137
<factor>	-8	def 85 86 91 96 ref 79 84
<line...>	-2	def 36 37 ref 32 37
<line>	-3	def 38 45 50 ref 36 37
<nl>	-4	def 56 ref 32 32 38 45 50
<pwr>	-7	def 79 84 ref 68 73 78 79
<ref>	-9	def 101 102 107 112 117 122 127 132 137 ref 85 86 91
<term>	-6	def 68 73 78 ref 57 62 67 68 73

TERMINAL ENCODING

1	<symbol>
2	<real>
3	list
4	q
5	'012
6	=
7	(
8	+
9	-
10	/
11	*
12	**
13	)

14 abs
15 atan
16 cos
17 ln
18 log
19 sin
20 tan

DPDA LISTING

```
[     1] 000000    000014
         000001-> 000016   READ "<symbol>"
         000002-> 000025   READ "<real>"
         000003-> 000031   READ "list"
         000004-> 000033   READ "q"
         000007-> 000034   READ "("
         000008-> 000047   READ "+"
         000009-> 000057   READ "-"
         000014-> 000058   READ "abs"
         000015-> 000060   READ "atan"
         000016-> 000062   READ "cos"
         000017-> 000064   READ "ln"
         000018-> 000066   READ "log"
         000019-> 000068   READ "sin"
         000020-> 000070   READ "tan"

[    16] 000000    000008
         000005->-000263   LOOK "
"
         000006-> 000122   READ "="
         000008->-000263   LOOK "+"
         000009->-000263   LOOK "-"
         000010->-000263   LOOK "/"
         000011->-000263   LOOK "*"
         000012->-000263   LOOK "**"
         000013->-000263   LOOK ")"

     25  000004    000005   APPLY
         000000    000000   pd ld
        -000020    000001   rule/alt
         000000-> 000117
         000047-> 000146
         000057-> 000149

[    31] 000000    000001
         000005-> 000123   READ "
"
[    33] 000002    000031   SHARE

[    34] 000000    000012
         000001-> 000139   READ "<symbol>"
         000002-> 000025   READ "<real>"
         000007-> 000034   READ "("
         000008-> 000047   READ "+"
         000009-> 000057   READ "-"
         000014-> 000058   READ "abs"
         000015-> 000060   READ "atan"
         000016-> 000062   READ "cos"
         000017-> 000064   READ "ln"
         000018-> 000066   READ "log"
         000019-> 000068   READ "sin"
         000020-> 000070   READ "tan"

[    47] 000000    000009
         000001-> 000139   READ "<symbol>"
         000002-> 000025   READ "<real>"
         000014-> 000058   READ "abs"
         000015-> 000060   READ "atan"
         000016-> 000062   READ "cos"
         000017-> 000064   READ "ln"
         000018-> 000066   READ "log"
         000019-> 000068   READ "sin"
         000020-> 000070   READ "tan"

[    57] 000002    000047   SHARE

[    58] 000000    000001
         000007-> 000152   READ "("

[    60] 000000    000001
         000007-> 000153   READ "("

[    62] 000000    000001
         000007-> 000154   READ "("

[    64] 000000    000001
         000007-> 000155   READ "("

[    66] 000000    000001
         000007-> 000156   READ "("

[    68] 000000    000001
         000007-> 000157   READ "("

[    70] 000000    000001
         000007-> 000158   READ "("

[    72] 000000    000001
         000000-> 000000   READ "EOI"
[    74] 000000    000014
         000001-> 000016   READ "<symbol>"
         000002-> 000025   READ "<real>"
         000003-> 000031   READ "list"
         000004-> 000159   READ "q"
         000007-> 000034   READ "("
         000008-> 000047   READ "+"
         000009-> 000057   READ "-"
         000014-> 000058   READ "abs"
         000015-> 000060   READ "atan"
         000016-> 000062   READ "cos"
         000017-> 000064   READ "ln"
         000018-> 000066   READ "log"
```

```
        000019-> 000068   READ "sin"
        000020-> 000070   READ "tan"
    89  000005   000074   APPLY 1
        000000   000000   pd ld
       -000002   000001   rule/alt

[   92] 000000   000003
        000005-> 000123   READ "
"
        000008-> 000163   READ "+"
        000009-> 000164   READ "-"
[   96] 000000   000006
        000005->-000266   LOOK "
"
        000008->-000266   LOOK "+"
        000009->-000266   LOOK "-"
        000010-> 000168   READ "/"
        000011-> 000169   READ "*"
        000013->-000266   LOOK ")"
[  103] 000000   000007
        000005->-000279   LOOK "
"
        000008->-000279   LOOK "+"
        000009->-000279   LOOK "-"
        000010->-000279   LOOK "/"
        000011->-000279   LOOK "*"
        000012-> 000170   READ "**"
        000013->-000279   LOOK ")"
   111  000004   000005   APPLY
        000000   000000   pd ld
       -000015   000001   rule/alt
        000000-> 000103
        000168-> 000220
        000169-> 000228

   117  000004   000004   APPLY
        000000   000000   pd ld
       -000016   000001   rule/alt
        000000-> 000111
        000170-> 000236

[  122] 000002   000034   SHARE

   123  000004   000007   APPLY
        000000   000000   pd ld
       -000007   000001   rule/alt
        000000-> 000131
        000033-> 000136
        000092-> 000165
```

```
        000159-> 000203
        000171-> 000239
   131  000004   000004   APPLY
        000001   000001   pd ld
        000004   000001   rule/alt
        000000-> 000089
        000074-> 000160

   136  000005   000072   APPLY 1
        000001   000001   pd ld
        000001   000002   rule/alt
   139  000006   000025   APPLY SHR
        000000   000000   pd ld
        000021   000001   rule/alt

[  142] 000000   000003
        000008-> 000163   READ "+"
        000009-> 000164   READ "-"
        000013-> 000172   READ ")"
   146  000006   000117   APPLY SHR
        000001   000001   pd ld
        000017   000001   rule/alt

   149  000006   000117   APPLY SHR
        000001   000001   pd ld
        000018   000001   rule/alt

[  152] 000002   000034   SHARE
[  153] 000002   000034   SHARE

[  154] 000002   000034   SHARE

[  155] 000002   000034   SHARE

[  156] 000002   000034   SHARE

[  157] 000002   000034   SHARE

[  158] 000002   000034   SHARE

[  159] 000002   000031   SHARE

   160  000005   000074   APPLY 1
        000001   000001   pd ld
       -000003   000001   rule/alt

[  163] 000002   000034   SHARE

[  164] 000002   000034   SHARE
```

```
    165   000006    000131   APPLY SHR
          000001    000001   pd ld
          000006    000001   rule/alt

[   168]  000002    000034   SHARE

[   169]  000002    000034   SHARE

[   170]  000002    000034   SHARE

[   171]  000002    000092   SHARE

    172   000006    000117   APPLY SHR
          000002    000002   pd ld
          000019    000001   rule/alt

[   175]  000000    000003
          000008->  000163   READ "+"
          000009->  000164   READ "-"
          000013->  000242   READ ")"

[   179]  000000    000003
          000008->  000163   READ "+"
          000009->  000164   READ "-"
          000013->  000245   READ ")"

[   183]  000000    000003
          000008->  000163   READ "+"
          000009->  000164   READ "-"
          000013->  000248   READ ")"

[   187]  000000    000003
          000008->  000163   READ "+"
          000009->  000164   READ "-"
          000013->  000251   READ ")"

[   191]  000000    000003
          000008->  000163   READ "+"
          000009->  000164   READ "-"
          000013->  000254   READ ")"

[   195]  000000    000003
          000008->  000163   READ "+"
          000009->  000164   READ "-"
          000013->  000257   READ ")"

[   199]  000000    000003
          000008->  000163   READ "+"
          000009->  000164   READ "-"
          000013->  000260   READ ")"

    203   000005    000072   APPLY 1
          000002    000002   pd ld
          000001    000001   rule/alt

[   206]  000000    000006
          000005->-000285    LOOK "
"
          000008->-000285    LOOK "+"
          000009->-000285    LOOK "-"
          000010-> 000168    READ "/"
          000011-> 000169    READ "*"
          000013->-000285    LOOK ")"

[   213]  000000    000006
          000005->-000288    LOOK "
"
          000008->-000288    LOOK "+"
          000009->-000288    LOOK "-"
          000010-> 000168    READ "/"
          000011-> 000169    READ "*"
          000013->-000288    LOOK ")"

[   220]  000000    000007
          000005->-000291    LOOK "
"
          000008->-000291    LOOK "+"
          000009->-000291    LOOK "-"
          000010->-000291    LOOK "/"
          000011->-000291    LOOK "*"
          000012-> 000170    READ "**"
          000013->-000291    LOOK ")"

[   228]  000000    000007
          000005->-000294    LOOK "
"
          000008->-000294    LOOK "+"
          000009->-000294    LOOK "-"
          000010->-000294    LOOK "/"
          000011->-000294    LOOK "*"
          000012-> 000170    READ "**"
          000013->-000294    LOOK ")"

    236   000006    000111   APPLY SHR
          000002    000002   pd ld
          000014    000001   rule/alt

    239   000006    000131   APPLY SHR
          000003    000003   pd ld
          000005    000001   rule/alt

    242   000006    000025   APPLY SHR
          000003    000003   pd ld
          000026    000001   rule/alt
```

245	000006	000025	APPLY SHR
	000003	000003	pd ld
	000025	000001	rule/alt
248	000006	000025	APPLY SHR
	000003	000003	pd ld
	000023	000001	rule/alt
251	000006	000025	APPLY SHR
	000003	000003	pd ld
	000027	000001	rule/alt
254	000006	000025	APPLY SHR
	000003	000003	pd ld
	000028	000001	rule/alt
257	000006	000025	APPLY SHR
	000003	000003	pd ld
	000022	000001	rule/alt
260	000006	000025	APPLY SHR
	000003	000003	pd ld
	000024	000001	rule/alt
263	000006	000025	APPLY SHR
	000001	000000	pd ld
	000021	000001	rule/alt
266	000004	000012	APPLY
	000001	000000	pd ld
	-000010	000001	rule/alt
	000000->	000092	
	000034->	000142	
	000122->	000171	
	000152->	000175	
	000153->	000179	
	000154->	000183	
	000155->	000187	
	000156->	000191	
	000157->	000195	
	000158->	000199	
279	000004	000005	APPLY
	000001	000000	pd ld
	-000013	000001	rule/alt
	000000->	000096	
	000163->	000206	
	000164->	000213	
285	000006	000266	APPLY SHR
	000003	000002	pd ld
	000008	000001	rule/alt
288	000006	000266	APPLY SHR
	000003	000002	pd ld
	000009	000001	rule/alt
291	000006	000279	APPLY SHR
	000003	000002	pd ld
	000012	000001	rule/alt
294	000006	000279	APPLY SHR
	000003	000002	pd ld
	000011	000001	rule/alt

COMPILATION LISTING OF SEGMENT lcalc
Compiled by: Multics PL/I Compiler, Release 20e, of May 22, 1976
Compiled on: 06/24/76 1242.8 mst Thu
    Options: map table

```
lcalc:    proc;

/*        version of calc using LRK */

dcl       1 sym_(200),
          2 name    char(8),
          2 val     float bin(27);
dcl       1 sym based like sym_;
dcl       parenct   fixed bin(24);
dcl       ifile     char(200);
dcl       ifc(200) char(1)unal defined (ifile);
dcl       ifln      fixed bin(24);
dcl       ifi       fixed bin(24);
dcl       ifl       fixed bin(24);
dcl       ife       fixed bin(24);
dcl       sym_num   fixed bin(24);

dcl       TLanl     fixed bin(24) int static init(9);
dcl       TLan(9)   fixed bin(24) int static init(3,4,14,15,16,17,18,19,20);
dcl       TLstl     fixed bin(24) int static init(9);
dcl       TLst(9)   fixed bin(24) int static init(13,12,11,10,9,8,7,5,6);

          sym_num = 2;
          sym_.name(1) = "pi";
          sym_.val(1) = 3.14159265;
          sym_.name(2) = "e";
          sym_.val(2) = 2.7182818;

          ifln = 0;
retry:
          parenct = 0;
          ifi = 1;
          ife = 200;
          ifl = 0;
          call calc_p;
                    return;

error:
          call ioa_("^a",msg);
          goto retry;

dcl       msg       char(100)var;
dcl       ioa_      entry options(variable);

dcl       1 calc_t_$TL        ext static,
            2 TLsize          fixed bin,
            2 TL(20),
              3 (pt,ln)       fixed bin(17)unal;
```

```

dcl       1 calc_t_$TC          ext static,
            2 TCsize            fixed bin,
            2 TC    char(50);

dcl       1 calc_t_$DPDA        ext static,
            2 DPDAsize          fixed bin,
            2 DPDA(296),
              3 (v1,v2)         fixed bin(17)unal;
dcl       DPDAp     ptr;


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

calc_p: proc ();

/* Parser for tables created by LRK. */



          current_state = 1;
          ls_top, ps_top = 0;
          la_put, la_get = 1;
          la_ct = 0;

/* The parsing loop. */
NEXT:
          if (current_state = 0)
          then do;
done_parse:
               return;
          end;
          current_table = current_state;
          goto CASE (DPDA.v1 (current_table));

CASE (3): /* Shared look */                               /* . . . */
          current_table = DPDA.v2 (current_table);
CASE (1): /* Look. */                                     /* . . . */
          la_use = mod (la_get+la_need-1, -lbound (lstk, 1))+1;
          if (la_need = -lbound (lstk, 1))
          then signal condition (lastk_ovflo);
dcl  lastk_ovflo condition;
          la_need = la_need + 1;
          goto read_look;

CASE (10): /* Shared read */                              /* . . . */
          current_table = DPDA.v2 (current_table);

CASE (9): /* Read. */                                     /* . . . */
          la_need = 1;
          la_use = la_get;
          goto read_look;

```

```
CASE (2): /* Stack and Shared read */                          /* . . . */
          current_table = DPDA.v2 (current_table);

CASE (0): /* Stack and Read. */                                /* . . . */
          la_need = 1;
          la_use = la_get;
          if (ps_top = hbound (parse_stack, 1))
          then signal condition (pstk_ovflo);
dcl  pstk_ovflo condition;
          ps_top = ps_top+1;                                   /* Top of  parsing stack. */
          parse_stack (ps_top) = current_state;    /* Stack the current state. */
          cur_lex_top (ps_top) = ls_top;           /* save current lex top (for recovery) */
read_look:
          do while (la_ct < la_need);   /* make sure enough symbols are available */
               call scanner ();
               la_put = mod (la_put, -lbound (lstk, 1))+1;
               la_ct = la_ct + 1;
          end;
          test_symbol = lstk.symbol (-la_use);
          lb = current_table+1;
          ub = current_table+DPDA.v2 (current_table);
          do while (lb <= ub);
               m = divide (ub+lb, 2, 24, 0);
               if (DPDA.v1 (m) = test_symbol)
               then do;
                    next_state = DPDA.v2 (m);
                    goto got_symbol;
               end;
               if (DPDA.v1 (m) < test_symbol)
               then lb = m+1;
               else ub = m-1;
          end;


          if (test_symbol ^= 5)
          then parenct = 0;
          msg = errmsg(sign(parenct));
          goto error;
dcl errmsg(-1:1) char(16)int static init(
"too many )",
"missing operator",
"too few )");

got_symbol:
          current_state = next_state;
          if (current_state < 0) then do;          /* Transition is a look-ahead state. */
               current_state = -current_state;
          end;
          else do;
               if (ls_top = hbound (lstk, 1))
               then signal condition (lstk_ovflo);
dcl  lstk_ovflo condition;
               ls_top = ls_top + 1;
```

```
               lstk (ls_top) = lstk (-la_get);
               la_get = mod (la_get, -lbound (lstk, 1)) + 1;
               la_ct = la_ct - 1;
          end;
          goto NEXT;

CASE (4): /* Apply state. */                                 /* . . . */
CASE (5): /* Apply single */                                 /* . . . */
CASE (6): /* Apply Shared */                                 /* . . . */
          la_need = 1;
          rulen = DPDA.v1 (current_table+2);
          altn = DPDA.v2 (current_table+2);
          if (rulen > 0) then do;
               call semantics (rulen, altn);
          end;
          ps_top = ps_top - DPDA.v1 (current_table+1);       /* Delete parse stack states. */
          ls_top = ls_top - DPDA.v2 (current_table+1);       /* delete lex stack states */
          if (DPDA.v1 (current_state) = 5)
          then do;
               current_state = DPDA.v2 (current_table);
               goto NEXT;
          end;
          if (DPDA.v1 (current_state) = 6)
          then do;
               current_table = DPDA.v2 (current_table);
          end;
          do i = current_table+4 to current_table+DPDA.v2 (current_table);
               if (DPDA.v1 (i) = parse_stack (ps_top))
               then do;
                    current_state = DPDA.v2 (i);
                    goto NEXT;
               end;
          end;
          current_state = DPDA.v2 (current_table+3);
          goto NEXT;

dcl 1 lstk (-4:50)
                                        /* -4:-1 is the look-ahead stack (FIFO) */
                                        /* 1:50 is the lexical stack (LIFO) */
     , 2 symptr ptr                     /* pointer to symbol (must be valid) */
     , 2 symlen fixed bin (24)          /* length of symbol (may be 0) */
     , 2 line fixed bin (24)            /* line where symbol begins */
     , 2 symbol fixed bin (24)          /* encoding of symbol */
     , 2 value float bin (27)
     , 2 def ptr

     ;
dcl  ls_top fixed bin (24);             /* location of top of lexical stack */
dcl  cur_lex_top (100) fixed bin (24);  /* current lex top stack (with parse_stack) */
dcl  parse_stack (100) fixed bin (24);  /* parse stack */
dcl  altn fixed bin (24);               /* APPLY alternative number */
dcl  current_state fixed bin (24);      /* number of current state */
dcl  test_symbol fixed bin (24);        /* encoding of current symbol */
```

```
dcl  current_table fixed bin (24);       /* number of current table */
dcl  i fixed bin (24);                   /* temp */
dcl  la_ct fixed bin (24);               /* number of terminals in look-ahead stack */
dcl  la_get fixed bin (24);              /* location in look_ahead stack to get next symbol */
dcl  la_need fixed bin (24);             /* number of look-ahead symbols needed */
dcl  la_put fixed bin (24);              /* location in look_ahead stack to put next symbol */
dcl  la_use fixed bin (22);              /* location in look-ahead stack to test with */
dcl  next_state fixed bin (24);          /* number of next state */
dcl  nil_sym fixed bin (24);
dcl  ps_top fixed bin (24);              /* location of top of parse stack */
dcl  recov_msg char (150)var;
dcl  rulen fixed bin (24);               /* APPLY rule number */
dcl  t fixed bin (24);
dcl  ioa_ entry options (variable);


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

scanner: proc;



MORE:
          lstk.symptr (-la_put) = addr (ifc (ifi));
          lstk.symlen (-la_put) = 0;
          lstk.line (-la_put) = ifln;
          if (ifi > ifl)
          then do;
               if (ifi > ife)
               then do;
                    lstk.symbol (-la_put) = 0;
                    return;
               end;
               call get_line;
               goto MORE;
          end;
          i = verify (substr (ifile, ifi, ifl-ifi+1), alpha);
dcl  alpha char (53)int static
          init ("ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz");
          if (i > 1)
          then do;
               i = i - 1;
dcl  char8 char (8);
               char8 = substr (ifile, ifi, i);
               ifi = ifi + i;
               do jj = 1 to TLanl;
                    j = TLan (jj);
                    if (substr (TC, TL.pt (j), TL.ln (j)) = char8)
                    then do;
                         lstk.symbol (-la_put) = j;
                         return;
                    end;
               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 (-la_put) = addr (sym_ (i));
               lstk.symbol (-la_put) = 1;
               return;
          end;
          else do;
               j = verify (substr (ifile, ifi, ifl-ifi+1), "0123456789.");
               if (j > 1)
               then do;
                    if (substr (ifile, ifi+j-1, 1) = "e")
                    then do;
                         j = j + 1;
                         if (substr (ifile, ifi+j-1, 1) = "+")
                         | (substr (ifile, ifi+j-1, 1) = "-")
                         then j = j + 1;
                         j = j - 1
                              + verify (substr (ifile, ifi+j-1, ifl), "0123456789");
                    end;
dcl  flb float bin (27);
                    j = j - 1;
                    on conversion begin;
                         msg = "missing operator";
                         goto error;
                    end;
                    flb = convert (flb,substr (ifile, ifi, j));
                    lstk.value (-la_put) = flb;
                    lstk.symbol (-la_put) = 2;
                    lstk.symlen (-la_put) = j;
                    ifi = ifi + j;
                    return;
               end;
               else do;
                    do jj = 1 to TLstl;
                         j = TLst (jj);
                         if (substr (TC, TL.pt (j), TL.ln (j))
                              = substr (ifile, ifi, TL.ln (j)))
                         then do;
                              lstk.symbol (-la_put) = j;
                              ifi = ifi + TL.ln (j);
                              if (j = 7)          /* left paren */
                              then parenct = parenct + 1;
                              else if(j = 13)     /* right paren */
                              then parenct = parenct - 1;
                              return;
                         end;
                    end;
```

```
                    end;
               end;
               if (substr (ifile, ifi, 1) = " ")
               then do;
                    ifi = ifi + 1;
                    goto MORE;
               end;
               msg = "illegal char ";
               msg = msg || substr(ifile,ifi,1);
               goto error;

/*             /* . . . GET_LINE . . . */

get_line: proc;
                    ifln = ifln + 1;
                         ifi = 1;
                         ifl = 1;
                         do while(ifl < 2);
                              call iox_$get_line (iox_$user_input, addr (ifile), 200, ifl, 0);
                         end;
dcl iox_$user_input ptr ext static;
                         if (substr (ifile, ifi, ifl) = "EOI
")
                         then ifl, ife = 0;
               end;


          end;

/* END INCLUDE FILE ..... calc_s.incl.pl1 ..... */


/* -order                  <symbol>
               <real>
               list
               q
                012
               =
               (
               +
               -
               /
               *
               **
               )
               abs
               atan
               cos
               ln
               log
               sin
               tan
-tl
```

```
4   -table calc_t_.incl.pl1
4   -sem calc_.incl.pl1
4   -parse */
4   semantics:          proc(rulen,altn);
4
4   dcl         rulen       fixed bin(24),
4               altn        fixed bin(24);
4
4               goto rule(rulen);
4
4   /* <calc> ::= <line...> q <nl> ¦ q <nl> ! */
4   rule(0001):
4               goto done_parse;
4
4   /* <line...>          ::= <line> ! */;
4   /* <line...>          ::= <line...> <line> ! */
4   /* <line> ::= list <nl> ! */;
4   rule(0004):
4               do i = sym_num to 1 by -1;
4                    call ioa_("^8a = ^f",sym_.name(i),sym_.val(i));
4               end;
4               return;
4
4   /* <line> ::= <symbol> = <exp> <nl> ! */;
4   rule(0005):
4               lstk.def(ls_top-3)->sym.val = lstk.value(ls_top-1);
4               return;
4
4   /* <line> ::= <exp> <nl> ! */
4   rule(0006):
4               call ioa_("=    ^f",lstk.value(ls_top-1));
4               return;
4   dcl         char15      char(17);
4
4   /* <nl>   ::= '012 ! */
4   /* <exp>  ::= <exp> + <term> ! */;
4   rule(0008):
4               lstk.value(ls_top-2) = lstk.value(ls_top-2) + lstk.value(ls_top);
4               return;
4
4   /* <exp>  ::= <exp> - <term> ! */;
4   rule(0009):
4               lstk.value(ls_top-2) = lstk.value(ls_top-2) - lstk.value(ls_top);
4               return;
4
4   /* <exp>  ::= <term> ! */
4   /* <term> ::= <term> * <pwr> ! */;
4   rule(0011):
4               lstk.value(ls_top-2) = lstk.value(ls_top-2) * lstk.value(ls_top);
4               return;
4
4   /* <term> ::= <term> / <pwr> ! */;
4   rule(0012):
```

```
                lstk.value(ls_top-2) = lstk.value(ls_top-2) / lstk.value(ls_top);
                return;

/* <term> ::= <pwr> ! */
/* <pwr>  ::= <pwr> ** <factor> ! */;
rule(0014):
                lstk.value(ls_top-2) = lstk.value(ls_top-2) ** lstk.value(ls_top);
                return;

/* <pwr>  ::= <factor> ! */
/* <factor>          ::= <ref> ! */;
/* <factor>          ::= + <ref> ! */;
rule(0017):
                lstk.value(ls_top-1) = lstk.value(ls_top);
                return;

/* <factor>          ::= - <ref> ! */;
rule(0018):
                lstk.value(ls_top-1) = - lstk.value(ls_top);
                return;

/* <factor>          ::= ( <exp> ) ! */
rule(0019):
                lstk.value(ls_top-2) = lstk.value(ls_top-1);
                return;

/* <ref>  ::= <real> ! */;
/* <ref>  ::= <symbol> ! */;
rule(0021):
                lstk.value(ls_top) = lstk.def(ls_top)->sym.val;
                return;

/* <ref>  ::= sin ( <exp> ) ! */;
rule(0022):
                lstk.value(ls_top-3) = sin(lstk.value(ls_top-1));
                return;

/* <ref>  ::= cos ( <exp> ) ! */;
rule(0023):
                lstk.value(ls_top-3) = cos(lstk.value(ls_top-1));
                return;

/* <ref>  ::= tan ( <exp> ) ! */;
rule(0024):
                lstk.value(ls_top-3) = tan(lstk.value(ls_top-1));
                return;

/* <ref>  ::= atan ( <exp> ) ! */;
rule(0025):
                lstk.value(ls_top-3) = atan(lstk.value(ls_top-1));
                return;

/* <ref>  ::= abs ( <exp> ) ! */;
```

```
4     rule(0026):
4               lstk.value(ls_top-3) = abs(lstk.value(ls_top-1));
4               return;
4
4     /* <ref>  ::= ln ( <exp> ) ! */;
4     rule(0027):
4               lstk.value(ls_top-3) = log(lstk.value(ls_top-1));
4               return;
4
4     /* <ref>  ::= log ( <exp> ) ! */
4     rule(0028):
4               lstk.value(ls_top-3) = log10(lstk.value(ls_top-1));
4               return;
4
4     end;
2
2          end;
2
2     /* END INCLUDE FILE ..... calc_p.incl.pl1 ..... */

      end;
```

INCLUDE FILES USED IN THIS COMPILATION.

LINE	NUMBER	NAME	PATHNAME
44	1	calc_t_.incl.pl1	>udd>m>jaf>cur>calc_t_.incl.pl1
46	2	calc_p.incl.pl1	>udd>m>jaf>cur>calc_p.incl.pl1
2-163	3	calc_s.incl.pl1	>udd>m>jaf>cur>calc_s.incl.pl1
2-164	4	calc_.incl.pl1	>udd>m>jaf>cur>calc_.incl.pl1