PARALLEL LR PARSING

Dan Ligett
Glen McCluskey
W. M. McKeeman

July, 1982

Technical Report TR-82-03

Key Words and Phrases: Parallel processors, Parsing
CR categories: C.1.2 (Multiple data stream architectures),
D.3.4 (Processors)

ABSTRACT

Existing LR algorithms are sequential, passing over source text left-to-right, discovering the phrase structure in a depth-first, left-to-right walking of the parse tree. Here we extend the LR technique to allow arbitrarily many processors to share the simultaneous building of the parse tree in parallel. The algorithm is a modification of the error recovery algorithm of DeRemer and Pennello where the LR table constructor starts on an indefinite intermediate configuration instead of the specific leftmost configuration of Knuth. The technique is applicable to variants of the LR algorithm.

Performance of the algorithm was measured experimentally. The criterion was speed, the parameters are number of processors and size of the territory per processor. The result is a linear relation between speed and number of processors for territories of reasonable size. More precisely, run time $T$ is given by:

$$T = K(1/N)$$

with $N$ being the number of processors, and $K$ varying inversely with the number of tokens in each processor's territory. $K$ is typically small (1.25 or less, resulting in 80% or greater efficiency) when each processor has at least 15 tokens to work on.
LR Parsers.

LR parsers for programming languages usually take the form of a table generating algorithm and a parsing algorithm. The generating algorithm is typically some efficient variant of the LR(1) algorithm [Knuth 65]; the parser uses a stack and the parsing tables to build a parse tree or some semantically sufficient equivalent. The invariant of the process is that the concatenation of the parse stack and unprocessed text is always a sentential form. Such algorithms are single-mindedly sequential.

This report introduces a form of LR(1) algorithm, PARALR, that can function in parallel. For reasons of clarity, the algorithm is expressed in terms of the original LR(1) algorithm of Knuth rather than one of the variants mentioned above. It is presumed that the reader is familiar with this material. It is conjectured that this technique can be used with any of the LR variants. The proposed application of this new algorithm is on a rather long input text with a large number of independent processors eager to parse it.

The parallel parser algorithm is based on the concept of an "error recovery parser" [Pennello and DeRemer 1978]. An error recovery parser is one that can be started at an arbitrary point in the input string without the left context normally provided by the parse stack. It can proceed some distance to the right before becoming blocked. At that point an ad hoc repair is effected that is consistent with the previously parsed input and the information just gathered by the error recovery parser. The presumption is that the repair is better when based on the information gathered by the error recovery parser.

Parallel parsing a syntactically correct text is achieved by breaking the input text into contiguous "territories", starting a parser on each territory in parallel, each but the leftmost acting as an error recovery parser until one of three things occurs:

1. a parser must poach on the territory of a parser on its right to read or lookahead
2. a parser must poach on the territory of a parser to its left to make a reduction
3. a parser lacks enough information to choose a reduction (similar to a failure to be LR(1))

In case (1) the parser is blocked. There are two choices. It may wait for the parser to its right to release some territory or it may release its own territory and accept reassignment to other tasks. The leftmost parser must choose to wait. In cases (2) and (3) the parser must release the part of its territory it has just processed (perhaps freeing a blocked parser to its left) and start afresh on the as yet untouched territory still to its right. We call this "stuttering". Stuttering is reduced when strong left bracketing symbols such as "(" or "begin" are encountered.
Notation.

Symbols (tokens) A B C... -- upper case letters
Strings (perhaps nil) a b c... -- lower case letters
Text marker .
Rewriting rule A = b

Table Building.

The parsing table is built in two stages. First LR(1) tables are built. The construction is as usual except both terminals and nonterminals are recorded for lookahead. Then the error recovery tables are joined to them. The difference is that the normal LR(1) table constructor starts with a configuration set containing only the leftmost rule marked in the leftmost position [Knuth 65] while the error recovery constructor starts with a configuration set which is the union of all the LR(1) configuration sets less configurations that signify a reduce decision. This starting set therefore contains all marked rules of the form:

\[ A = a.BbC \]

where a and b may be empty. The text marker "." may be anywhere in the right hand side except the rightmost position; the symbol C is the lookahead-1 symbol. For sake of brevity, marked rules that are identical except for lookahead are notationally combined by listing the set of lookahead symbols as a string following a single instance of the marked rule. When error recovery table generation converges to configuration sets found in the normal tables, they need not be duplicated. The combined configuration sets are mapped onto a finite state machine as usual. The states corresponding to the two initial states above are called the left state and the mid state; the rest of the states are numbered arbitrarily, each number corresponding to a computed configuration set.

Parsing.

The same parser algorithm is used in both cases. Suppose there are N+1 parsers. The leftmost parser is started in the left state; the remaining N parallel parsers are started in the mid state. Suppose the text is

\[ t = t_0t_1t_2...t_N \]
Assuming that $t$ is syntactically correct, then initially the predicate

$$\text{SententialForm}(t)$$

is true. This is the invariant of the process. As the parsing proceeds, the territories $t_i$ undergo substitution in place. Except for $t_0$, each is eventually replaced by the empty string as the $i$th processor abandons its task and all its partial results to the processor to its left. $t_0$ becomes $G$, the leftmost symbol of the grammar. The invariant is maintained since upon termination

$$t_0t_1\ldots t_n = t_0 = G$$

This must, of course, be shown to be true.

Example.

Given the LR(1) grammar for expressions with implicit multiplication,

$$G = E^+\#$$
$$E = E+T$$
$$E = T$$
$$T = T.Z$$
$$T = Z$$

The parallel LR(1) tables are constructed from the sets:

<table>
<thead>
<tr>
<th>state 0 (left)</th>
<th>state 1</th>
<th>state 4</th>
<th>state 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G = E^+$</td>
<td>$E = E^+T$</td>
<td>$E = E^+.T$</td>
<td>$E = E^+.T$</td>
</tr>
<tr>
<td>$E = E^+T$</td>
<td>$E = E^+.T$</td>
<td>$T = .T.Z$</td>
<td>$T = T.Z$</td>
</tr>
<tr>
<td>$E = T$</td>
<td>$T = .T.Z$</td>
<td>$T = .Z$</td>
<td>$T = T.Z$</td>
</tr>
<tr>
<td>$T = Z$</td>
<td>$T = T.Z$</td>
<td>$T = T.Z$</td>
<td>$T = T.Z$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state 2</th>
<th>state 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E = T.$</td>
<td>$G = E^+$</td>
</tr>
<tr>
<td>$T = T.Z$</td>
<td>$T = T.Z$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state 3</th>
<th>state 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = Z.$</td>
<td>$T = T.Z.$</td>
</tr>
<tr>
<td>$T = T.Z.$</td>
<td>$T = T.Z.$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state 8 (mid)</th>
<th>state 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G = E^+$</td>
<td>$E = E^+.T$</td>
</tr>
<tr>
<td>$G = E^+.T$</td>
<td>$E = T.$</td>
</tr>
<tr>
<td>$E = E^+.T$</td>
<td>$T = T.Z.$</td>
</tr>
<tr>
<td>$E = .T$</td>
<td>$T = .Z$</td>
</tr>
<tr>
<td>$T = .T.Z$</td>
<td>$T = T.Z.$</td>
</tr>
<tr>
<td>$T = T.Z$</td>
<td>$T = T.Z.$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = T.Z.$</td>
</tr>
<tr>
<td>$T = .Z$</td>
</tr>
</tbody>
</table>
The resulting tables are:

<table>
<thead>
<tr>
<th>in state</th>
<th>see</th>
<th>do</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (left)</td>
<td>E</td>
<td>goto 1</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>goto 2</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>goto 3</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
<td>goto 4</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>goto 5</td>
</tr>
<tr>
<td>2</td>
<td>#+</td>
<td>reduce E = T</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>goto 6</td>
</tr>
<tr>
<td>3</td>
<td>#+Z</td>
<td>reduce T = Z</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>goto 7</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>goto 3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>stop</td>
</tr>
<tr>
<td>6</td>
<td>#+Z</td>
<td>reduce T = TZ</td>
</tr>
<tr>
<td>7</td>
<td>#+</td>
<td>reduce E = E+T</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>goto 6</td>
</tr>
<tr>
<td>8 (mid)</td>
<td>E</td>
<td>goto 1</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>goto 9</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>goto 10</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>goto 5</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>goto 4</td>
</tr>
<tr>
<td>9</td>
<td>#+</td>
<td>stutter</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>goto 6</td>
</tr>
<tr>
<td>10</td>
<td>#+Z</td>
<td>stutter</td>
</tr>
</tbody>
</table>

Bad Input.

The algorithm should behave reasonably in the face of incorrect input. At a minimum it must (1) detect at least one error, (2) provide reasonable diagnostic information, and (3) terminate gracefully. Since the algorithm is based on an error recovery scheme, it is obvious how to incorporate recovery into it. We are satisfied, however, with the lesser demands stated above. To achieve them, we simply abandon processing any erroneous text discovered by other than the special leftmost processor, leaving the trouble to be met again by the parser immediately to its left. Ultimately, the leftmost parser will have to deal with the problem. It is a conventional LR parser, so its detection, diagnostic and termination algorithm can be invoked.

Scheduling.

The correctness of the algorithm does not depend upon the scheduling of the processors; the efficiency does. We have only intuitive and experimental knowledge about choosing the scheduling algorithms and present one as an example. We believe that the optimal choice will depend on matters such as processor switching time and the cost of synchronization which are beyond the scope of this paper.
Let \( L \) be the length of the input text. Start all \( N+1 \) processors, 0 to \( N \), simultaneously, with processor \( k \) aimed at token \( k*\frac{L}{N+1} \), that is, the processors are spaced approximately equally over the input. Each processor has read/write-access to a pointer marking its starting point (left boundary) and a pointer marking the current input token and readonly-access to a pointer marking its right boundary.

\[
\text{text: } \overline{XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX} \\
\text{! } \overline{!} \overline{!} \overline{!} \overline{!} \overline{!} \\
\text{left pointers: } \overline{!} \overline{!} \overline{!} \overline{!} \\
\text{processors: } P0 \quad P1 \quad P2 \quad P3 \quad P4
\]

Each processor proceeds until one of three situations arises:

1. a reduction must cross the left boundary or the tables fail to contain enough information to make a decision.
   Response: restart at the current token in the mid state and update the left pointer permitting the next parser to the left to encroach on some previously reserved textual territory. The local parse stack of this parser becomes input text for the parser to its left. (stutter)

2. a read or lookahead must cross the right boundary.
   Response: update the left pointer to the value of the right pointer for this region and quit. The parser to the left inherits all unfinished business. Ultimately all the parallel parsers will abandon the entire text to the leftmost parser.

3. an error is detected.
   Response: as in 2 above except for the leftmost parser which executes the normal LR error behavior.

Why it works.

To show that the parallel LR method works, we must show three things. Given a grammar with start symbol \( G \) and an input text \( x \):

1. If \( x \) is a sentential form, PARALR computes the parse tree which gives the derivation of \( G \) from \( x \).

2. If \( x \) is not a sentential form, PARALR detects it and issues a diagnostic.

3. PARALR terminates for an arbitrary number of processors fairly scheduled on arbitrary territories for arbitrary input.

The central reason that LR works is that the set of possible parse stacks is a regular language. For parallel LR the corresponding fact is that the set of final strings of a regular language is also regular, in particular, the tops of the set of parse stacks.
Correct parsing.

Recall that the mid state comes from the configuration set which is the union of all the configuration sets generated using the normal LR(1) algorithm, less configurations which have the marker on the extreme right. Each configuration set reachable from the mid state is a superset of some configuration set in the normal LR(1) machine. Eventually the configuration sets reached are in the LR(1) machine. If a unique action can be determined from a PARALR configuration set, it is the same action that would have been dictated by the normal LR(1) machine.

We have to make sure that an application of the wrong rule is not done. There are four possibilities:

a. A processor runs out of territory on the right while still in the middle of a rule. It releases its territory, which means that a processor to the left will have to worry about the reductions.

b. It is time to do a reduction, and there is only one configuration left which is marked on the extreme right. In this case there is no problem, since the reduction made is the same one that the LR(1) parser would have made.

c. It is time to do a reduction, and there is only one configuration which is marked on the extreme right, but the reducible string belongs at least partly to the processor to the left. In this case it stutters, releases the territory to the left of its current position to the processor to the left, hence makes no reductions nor, therefore, errors.

d. It is time to do a reduction, and there is a reduce-reduce conflict or a read-reduce conflict. The processor does not have enough information to handle this case, so it stutters and leaves the problem to a processor to the left.

Based on this analysis, we see that any rule application which is done is identical to that done by an LR(1) parser, hence we will get the same parse tree as the LR(1) parser would get.

Error detection.

Suppose the text contains an error that would be detected by an LR(1) parser. That means that the input token is neither a read transition nor a lookahead in the configuration set which is controlling the parser when the error is detected.

Termination.

If $N = 0$, then we have $N + 1 = 1$ processors, and the problem reduces to LR(1). If $N$ is greater than 0 the parallel parsers maintain the invariant as shown in 1 above. Furthermore each parser either continues to process text or abandons the task to processors to the left. At worst this wastes effort, but does not affect termination since the leftmost parser is known to terminate and it eventually gets all the text.
Performance

To measure the performance of the parallel algorithm, we developed a Pascal simulation. Scheduling was done by means of a simple round-robin scheme, with each "processor" doing one parse table lookup and resulting action and then the next one taking its turn. We used number of Pascal statements executed as a performance measure.

Random 1000 character expressions of the following grammar were used in the tests.

\[
\begin{align*}
G &= E^* \\
E &= T \\
E &= E \text{ addop } T \\
T &= F \\
T &= TF \\
F &= N \\
F &= (E) \\
F &= F | N \\
N &= x
\end{align*}
\]

Speed improvement increased almost linearly with number of processors. The critical factor in determining speedup is the size of each processor's territory. To investigate the effect of territory size, we used the measure

\[
K = \frac{\text{instr}(N)}{(N*\text{instr}(1))}
\]

where \( \text{instr}(i) \) = number of instructions executed with \( i \) processors

Intuitively, \( K \) is the "inflation factor" over ideal behavior. We calculate efficiency as

\[
E = \frac{1}{K}
\]

Figure 1 shows the relationship between efficiency and territory size.
Figure 1.
Relation between efficiency and territory size.
Conclusions.

We have demonstrated that the LR algorithm can function in parallel with augmented tables. This may be of significance when it is economical to bring a large number of processors into play for tasks of this kind.

Acknowledgements.

The work was motivated by a series of questions about finding potential parallelism posed by Severo Ornstein and Will Crowther at Xerox Palo Alto Research Center. Frank DeRemer, Tom Pennello and Bill McKeeman worked through most of the details with respect to LALR(1) shortly thereafter. The present authors have carried through the LR(1) analysis and implementations reported here.

References.


PARALR
A program for parallel parsing
program paralr(input, output, grammar, intext, inparam);
{

\textit{Wang Institute of Graduate Studies – May 1982}
Dan Ligett

Introduction

This program is an implementation of the algorithm described in "Parallel LR Parsing" (Wang Institute Technical Report TR-92-XX). The program takes a BNF description of a language and a text string as inputs, applies the PARALR algorithm to generate parsing tables and simulates the parsing of the text string by multiple independent processors. This program is intended to be a tool for experimenting with the PARALR algorithm.

This program requires three input files named 'grammar', 'text' and 'parameters'. 'grammar' contains a BNF description of the input grammar. 'text' contains an input string to be parsed, 'parameters' contains a number of flags that control the output generated by PARALR.

Ten different types of output can be printed by PARALR, the first ten lines of the 'parameters' file control which of these items are printed for a given execution of PARALR. If the first character of a line in 'parameters' is 'Y' then the corresponding output is printed. The ten types of output are:

1) A trace through the finite state machine reading of the grammar file.
2) An echo of the input grammar file.
3) A listing of the production rules generated from the grammar.
4) A listing of the head matrix generated from the production rules.
5) A listing of the configuration sets generated from the grammar.
6) A listing of the finite state machine generated for parsing.
7) An echo of the input text read.
8) A listing of the territories assigned to each processor.
9) A trace of the states encountered in parsing the input text.
10) A trace of the reductions applied in parsing the input text.

The eleventh line of the 'parameters' file contains the number of processors to be used in parsing the input text.
# Table of Contents

## HAF symbol package

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>initsymbols</td>
<td>9</td>
</tr>
<tr>
<td>blank</td>
<td>10</td>
</tr>
<tr>
<td>append</td>
<td>11</td>
</tr>
<tr>
<td>symbolequal</td>
<td>12</td>
</tr>
<tr>
<td>insertbnfsymbol</td>
<td>13</td>
</tr>
<tr>
<td>symtype</td>
<td>14</td>
</tr>
<tr>
<td>symname</td>
<td>15</td>
</tr>
<tr>
<td>writebnfsymbol</td>
<td>16</td>
</tr>
<tr>
<td>writebnfformat</td>
<td>17</td>
</tr>
<tr>
<td>numbnfsym</td>
<td>18</td>
</tr>
</tbody>
</table>

## Production package

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>initprod</td>
<td>20</td>
</tr>
<tr>
<td>createprod</td>
<td>21</td>
</tr>
<tr>
<td>lookupprod</td>
<td>22</td>
</tr>
<tr>
<td>getsymaftermark</td>
<td>23</td>
</tr>
<tr>
<td>getsymlattermark</td>
<td>24</td>
</tr>
<tr>
<td>getsymbeforemark</td>
<td>25</td>
</tr>
<tr>
<td>appendtoken</td>
<td>26</td>
</tr>
<tr>
<td>writeproductions</td>
<td>27</td>
</tr>
<tr>
<td>gethead</td>
<td>28</td>
</tr>
<tr>
<td>writeheads</td>
<td>30</td>
</tr>
</tbody>
</table>

## Procedure readgrammar

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeclass</td>
<td>34</td>
</tr>
<tr>
<td>writeaction</td>
<td>35</td>
</tr>
<tr>
<td>getchar</td>
<td>36</td>
</tr>
<tr>
<td>insertaction</td>
<td>37</td>
</tr>
<tr>
<td>lookupaction</td>
<td>38</td>
</tr>
<tr>
<td>readgrammar - mainline</td>
<td>39</td>
</tr>
</tbody>
</table>

## Configuration package

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>50</td>
</tr>
<tr>
<td>findconfinsert</td>
<td>51</td>
</tr>
<tr>
<td>createconfig</td>
<td>52</td>
</tr>
<tr>
<td>mergelast</td>
<td>53</td>
</tr>
<tr>
<td>addconfig</td>
<td>54</td>
</tr>
<tr>
<td>deleteconfig</td>
<td>55</td>
</tr>
<tr>
<td>configequal</td>
<td>56</td>
</tr>
<tr>
<td>advanceconfig</td>
<td>58</td>
</tr>
</tbody>
</table>
const ci = 's';
ss = 's';
maxset = 100;
maxsym = 1000;
maxstring = 12;
maxhead = 200;
maxtext = 100000;
empty = -9;
maxproc = 1000;

138 type parseaction = (pass, apply, error, stutter);  // Possible parser actions
140 symboltype = (term, nonterm);  // Possible BNF symbol types
142 string = array [1..maxstring] of char;  // BNF symbols
144 prodpnt = "prod;"  // Pointer to productions
146 rhssymistpnt = "rhssymist;"  // Pointer to rhs symbols
148 configsetpnt = "configset;"  // Pointer to configuration set
150 statepnt = "state;"  // Pointer to states
152 bntsymistpnt = "bnfsymist;"  // Pointer to bnf symbols
154 prod = record
   id : integer;  // Id of production
   lnsid : integer;  // Id of nonterminal
   rhs : rhssymistpnt;  // Link to right hand side
   next : prodpnt  // Link to next production
end;
161 rhssymist = record
   id : integer;  // Id of rhs symbol
   next : rhssymistpnt  // Link to next symbol of rhs
end;
166 lasetty = set of 1..maxset;  // Look ahead symbols
168 configset = record
   id : integer;  // Production Id
   mark : integer;  // Location of mark. 1 --> in front of first
   laset : lasetty;  // List of look ahead symbols
   next : configsetpnt  // Link to next member
end;
state = record
    snum : integer;      \{ The state number \}
    cset : configsetpnt; \{ Pointer to configuration set \}
    next : statepnt     \{ Link to next state \}
end;

entry = record
    statenum : integer; \{ Current state number \}
    token : integer; \{ Current input symbol Id \}
    laset : lasettype; \{ Look ahead set \}
    action : parseaction; \{ Pass or Apply \}
    prod : integer; \{ Production to apply \}
    newstate : integer \{ State to go to next \}
end;

onfsm = record
    symid : integer; \{ Number of this bnf symbol \}
    name : string; \{ Actual symbol \}
    stype : symboltype; \{ Terminal/nonterminal \}
    next : onfsmplnt \{ Link to next bnf symbol \}
end;

procdef = record
    active : boolean; \{ Processor status \}
    curtext : integer; \{ Current character pointer \}
    left : integer; \{ Leftmost character assigned \}
    right : integer \{ Rightmost character assigned \}
end;
<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>204</td>
<td>`{                      Global Variables                   }</td>
</tr>
<tr>
<td>205</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>var grammar : text;           (File of production rules)</td>
</tr>
<tr>
<td>207</td>
<td>intext : text;                (File of input text)</td>
</tr>
<tr>
<td>208</td>
<td>inparam : text;               (File with execution parameters)</td>
</tr>
<tr>
<td>209</td>
<td>cll : integer;                (Symbol id for &quot;chicken foot&quot;)</td>
</tr>
<tr>
<td>210</td>
<td>ssid : integer;               (Symbol id for internal start symbol)</td>
</tr>
<tr>
<td>211</td>
<td>mstate_num : integer;         (State number of midstate)</td>
</tr>
<tr>
<td>212</td>
<td>voff : boolean;               (Verbose flag for BNF reading)</td>
</tr>
<tr>
<td>213</td>
<td>vecno : boolean;              (Verbose flag for BNF echo)</td>
</tr>
<tr>
<td>214</td>
<td>vprod : boolean;              (Verbose flag for production rules)</td>
</tr>
<tr>
<td>215</td>
<td>vhead : boolean;              (Verbose flag for head matrix)</td>
</tr>
<tr>
<td>216</td>
<td>vstate : boolean;             (Verbose flag for configurations)</td>
</tr>
<tr>
<td>217</td>
<td>vtext : boolean;              (Verbose flag for echo of text)</td>
</tr>
<tr>
<td>218</td>
<td>vassign : boolean;            (Verbose flag for processor assignment)</td>
</tr>
<tr>
<td>219</td>
<td>vtrace : boolean;             (Verbose flag for trace of parse)</td>
</tr>
<tr>
<td>220</td>
<td>vreduce : boolean;            (Verbose flag for trace of reductions)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>228</td>
<td>`{                      &quot;Local&quot; to BNF symbol package                  }</td>
</tr>
<tr>
<td>229</td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>bnsymroot : bnf_sym_listpnt;  (BNF symbol listhead)</td>
</tr>
<tr>
<td>231</td>
<td>lastsymid : integer;          (Latest symbol id used)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>236</td>
<td>`{                      &quot;Local&quot; to production package                   }</td>
</tr>
<tr>
<td>237</td>
<td></td>
</tr>
<tr>
<td>238</td>
<td>prdroot : prodpnt;            (Productions listhead)</td>
</tr>
<tr>
<td>239</td>
<td>lastpid : integer;            (Id of latest production rule)</td>
</tr>
<tr>
<td>240</td>
<td>headmatrix : array [1..maxhead,1..maxhead] of boolean;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>245</td>
<td>`{                      &quot;Local&quot; to state package                   }</td>
</tr>
<tr>
<td>246</td>
<td></td>
</tr>
<tr>
<td>247</td>
<td>laststate : integer;          (Latest state number)</td>
</tr>
</tbody>
</table>
250) {
251      ***************** "Local" to finite state machine package **************
252    )
253    fsmTable : array [1..maxism] of fsmentry;  {Finite State Machine table}
254    lastfsm : integer;                          {Latest slot used}
255    keys : array [0..maxism] of integer;       {Keys for fsm table}
256    )
257      ***************** "Local" to LR(1) package ****************
258    )
259    parsestates : array [1..maxtext] of integer; {Stack of states}
260    sentform : array [1..maxtext] of integer;  {Symbols representing input text}
261    lastchar : integer;                        {Last character to parse}
262    processor : array [1..maxproc] of procdef; {Processors for parsing}
263    numproc : integer;                        {Number of processors available}
264    savproc : integer;                        {Last processor scheduled}
265    )
266      ***************** "Local" to Main Program ***************
267    )
268    c : char;                                {Character from parameter file}
269    startId : integer;                       {Id of starting production}
270    startiset : lasettyp;                    {Starting look ahead set}
271    sconfig : configsetpnt;                  {Starting configuration}
272    statemt : statepnt;                      {States list head}
273    grammarok : boolean;                    {Grammar parsed ok flag}
274    headok : boolean;                       {Heads ok flag}
275    succok : boolean;                       {Successors ok flag}
276    lr1ok : boolean;                        {Grammar is lrl flag}
277    parseok : boolean;                      {Parse succeeded flag}
278    s : statepnt;                           {Points at states for successors}
279    midstate : statepnt;                    {Midstate state}
280    midok : boolean;                       {Midstate successors ok}
The ANF symbol package

These procedures manipulate the symbols defined during reading the ANF.

initWithsymbols
blank
append
 threesomequal
insertnfsymbol
syntype
symname
writeonfsymbol
writeonfformat
numnfsym
numnfsym
procedure initSymbols;
{
    Pre: None,
    Post: The symbol package is ready for use.
}
begin
    nSymRoot := nil {Initialize list to empty}
end (initSymbols);
procedure blank(var s: string);
{
  Pre: None,
  Post: The symbol 's' is set to blanks.
}
var l: integer;
begin
  for l := 1 to maxstring do s[l] := ''
end (blank);
procedure append(c:character; var s:string);
{
    Pre: None;
    Post: The first blank character in 's' is replaced by 'c'.
          If 's' has no blank characters, no changes are made.
}

var i : integer;
begin
    i := 1;
    while (i <= maxstring) and (s[i] <> ' ') do i := i + 1; {Search for blank}
    if i <= maxstring then s[i] := c {Fill in character}
end {append};
function symboolequal(a:string; b:string):boolean;
{
    Pre: None.
    Post: "symboolequal" is true if 'a' and 'b' represent the
        same string; otherwise "symboolequal" is false.
}

var match : boolean; {String match flag}
i     : integer; {Assume strings match}

match := true; {Initialize character count}
i := 0;
while (i < maxstring) and match do
begin
    i := i + 1; {Update character count}
    if a[i] <> b[i] then match := false {Check strings for equality}
end;
symboolequal := match
end {symboolequal};
procedure insertbnfsymbol(\textbf{symbol} : \textbf{string}; \textbf{type} : \textbf{symbol type}; \textbf{var} \textbf{symbolid} : \textbf{integer});

\textbf{Pre}: none.
\textbf{Post}: If \textbf{"symbol"} is already in the list of symbols
found during the reading of the file \textbf{"grammar"} then
\textbf{"symbolid"} is set to the original symbol id assigned.
If \textbf{"symbol"} is not in the list of symbols then it is
inserted into the list; \textbf{"symbolid"} is the symbol
id assigned.

\begin{verbatim}
301 var Newsymbol : BNFSYMListItem;
302  spnt : BNFSYMListItem;

304 begin
305 if bnfsymroot = nil then
306 begin
307   new(Newsymbol);
308   lastsymid := 1;
309   Newsymbol^ .symid := lastsymid;
310   Newsymbol^ .name := symbol;
311   Newsymbol^ .type := stype;
312   Newsymbol^ .next := nil;
313   Newsymbol^ .symid := lastsymid;
314   bnfsymroot := Newsymbol
315 end
316 else begin
317   spnt := bnfsymroot;
318   while (spnt^ .next <> nil) and not symbolequal(symbol,spnt^ .name) do
319     spnt := spnt^ .next;
320   if not symbolequal(symbol,spnt^ .name) then
321     begin
322       new(Newsymbol);
323       lastsymid := lastsymid + 1;
324       Newsymbol^ .symid := lastsymid;
325       Newsymbol^ .name := symbol;
326       Newsymbol^ .type := stype;
327       Newsymbol^ .next := bnfsymroot;
328       bnfsymroot := Newsymbol;
329     end
330     else
331       symbolid := spnt^ .symid
332 end;
end (insertbnfsymbol);
\end{verbatim}
function symtype(id:Integer):symboltype;
{
  Pre: "id" is the symbol id of an element of the right hand side of a production rule.
  Post: "symtype" is the type (terminal/nonterminal) of the symbol identified by 'id'.
}
var spnt: onsym*istpnt;
begin
  spnt := bntsyp*root;
  while spnt^.symid <> id do spnt := spnt^.next;
  symtype := spnt^.stype
end {symtype};
procedure synname(id:integer; var s:string);
{
  Pre: 'id' is the symbol id of an element of the right
  hand side of a production rule.
  Post: 's' is the name of the symbol identified by 'id'.
}

var spnt : builtinsymptnt; {Point into list of symbols}
begin
  spnt := bnsymroot; {Point at first symbol}
  while spnt^.symid <> id do spnt := spnt^.next; {Search for match}
  s := spnt^.name {Return symbol name}
end {synname};
procedure writebnfsymbol(symid:integer);
{
Pre: 1 <= symid <= lastsym. (i.e., A symbol corresponding
to 'symid' exists).
Post: The 'maxstring' characters corresponding to the symbol
with symbol id 'symid' are printed. If 'symid' is a
terminal then it is printed in quotes; otherwise two
blanks are printed at the end, so a total of
"maxstring" + 2 characters are always printed.
}
var symstring : string;
n : integer;
m : integer;
blankfound : boolean;

begin
  symname(symid,symstring);
  if symtype(symid) = term then
    begin
      blankfound := false;
      m := 1;
      while (m <= maxstring) and not blankfound do
        if symstring[m] <> '"'
          then
            begin
              write(symstring[m]);
              m := m + 1
            end
        else
          blankfound := true;
      write('"');
      for n := 1 to maxstring do write('"')
    end
  else
    begin
      for n := 1 to maxstring do write(symstring[n]);
      write('"')
    end
end {writebnfsymbol};
procedure writeunformat(symid: integer);
{
  Pre:  1 <= symid <= lastsym. (i.e., A symbol corresponding
to "symid" exists).
  Post: The characters corresponding to the symbol with symbol
        If "symid" are printed; trailing blanks are omitted.
        If "symid" is a terminal then it is printed in quotes.
        A blank is printed after the symbol is printed.
}
var symstring : string;
  i : integer;

begin
  if symtype(symid) = term then write(""");
  symname(symid,symstring);
  for i := 1 to maxstring do
    if symstring[i] <> "" then write(symstring[i]);
  if symtype(symid) = term
    then write(""")
  else write(""")
end {writeunformat};
function numbntsymb:integer;
{
    Pre: None.
    Post: "numbntsymb" is the total number of symbols defined by the bnf grammar read.
}
begin
numbntsymb := lastsym;
end {numbntsymb};
The production package

These procedures manipulate production rules.

- initprod
- createprod
- lookupid
- getsymaftermark
- getsymbeforemark
- appendto
- writeproductions
- gethead
- writen
d

```c
{

}
```
procedure initprod;
{
    Pre:  None.
    Post: The production package is ready to use.
}

begin
  lastpid := 0;
  prodroot := nil
  end (initprod);
procedure createprod(lnsid:integer; var newprod:prodpt);
{
  Pre:        'lastoid' is the last id that was assigned to a production rule. 'prodroot' is the root of a linked list of production rules.
  Post:      A new production rule is entered into the linked list with root 'prodroot'; the new entry represents a production rule with a left hand side 'lnsid'. 'lastoid' is incremented, and the updated 'lastoid' is assigned to the new production. 'prodroot' and 'newprod' point at the new production.

begin
  new(newprod);
  lastpid := lastpid + 1;
  newprod^.id := lastpid;
  newprod^.lnsid := lnsid;
  newprod^.rhs := nil;
  newprod^.next := prodroot;
  prodroot := newprod
end {createprod};
procedure lookupprod(pid:integer; var p:prodtype);
{
  Pre:  A production with id 'pid' exists.
  Post: 'p' will point to the production that has 'pid' as its id.
}

begin
  p := prodroot;
  while p^.id <> pid do p := p^.next;
end (lookupprod);
procedure getsymaftermark(c:configsetptnt; var rhstoken:integer;
               var rhstype:symtype; var ok:boolean);
{
The first configuration of 'c' must be a valid configuration.
'rhstoken' is set to the token of the symbol found immediately
after the scan marker in the first configuration of 'c'.
'rhstype' is set to the type (terminal or nonterminal) of the
symbol found immediately after the scan marker in the first
configuration of 'c'.
}
{
var rhsptn : rhsymsetptnt;
begin
  1 : integer;
p : prodptnt;
begin
  lookupprod(c^.ld,p);
  rhsptn := p^.rhs;
  for i := 1 to c^.mark - 1 do
    rhsptn := rhsptn^.next;
  if rhsptn = nil
    then ok := false
    else
      begin
        ok := true;
        rhstoken := rhsptn^.ld;
        rhstype := symtype(rhsptn^.ld)
end;
end (getsymaftermark);
procedure getsymaftermark(c:configsetpnt; var rhstoken:integer;
  var rhstype:symtype; var ok:boolean);
{
  This procedure finds the symbol 1 past the symbol after
  the scan marker.
}

var copy : configsetpnt;
begin
  new(copy);
  copy^.id := c^.id;
  copy^.mark := c^.mark + 1;
  copy^.laset := {};
  copy^.next := nil;
  getsymaftermark(copy,rhstoken,rhstype,ok);
  dispose(copy);
end {getsymaftermark};
procedure getsymbeforemark(configsetpnt: integer; var rhstoken: boolean);
begin
   copy := new(configsetpnt);
   copy.id := c.id;
   copy.mark := c.mark := 1;
   copy.lset := c.lset := nil;
   getsymaftermark(copy, rhstoken, rhstype, ok);
   dispose(copy);
end {getsymbeforemark};
procedure appendorhs(sid:integer; p:prodptnt);
{
    Pre: None.
    Post: A new symbol with Id 'sid' is entered at the end of
          the right hand side of production rule 'p' points at.
}
var newsymbool : rnssymlistptnt;
lastsymbool : rnssymlistptnt;
begin
    newsymbool.id := sid;
    newsymbool.next := nil;
    if p^.rhs = nil
    then p^.rhs := newsymbool
    else
        lastsymbool := p^.rhs;
        while lastsymbool^.next <> nil do
            lastsymbool := lastsymbool^.next;
            lastsymbool^.next := newsymbool
    end
end {appendorhs};
procedure writeproductions;
{
    Pre: None.
    Post: The productions in the linked list with root 'prodroot'
          are printed.
}

var p : prodnt;
    rns : rhssymlstptnt;
{Point into list of productions}

begin
    page(output);
    writeln("Production Rules");
    writeln("Number Production");
    writeln("--------------------------------------------------",
            "--------------------------------------------------");
{Point at first production}
    p := prodroot;
    while p <> nil do
        begin
            writeln(p^.id, ", ");
            writeln(p^.lhsid);
            writeln(" --> ");
            rns := p^.rns;
            while rns <> nil do
                begin
                    writeln(rns^.id);
                    rns := rns^.next
                end;
            writeln;
            p := p^.next
        end
end (writeproductions);
procedure getheads(var success:boolean);
{
    Pre:  "prodroot" is the root of a list of productions.
    All the symbols used in the productions are defined.
    Post: If 'success' is true then 'headmatrix' reflects what
    symbols are headsymbols for other symbols.
    }

var updated : boolean;
rowsym : integer;
colsym : integer;
canSYM : integer;
p : prodint;
maxSYM : integer;

begin
success := true;
maxSYM := numbnfsym;
if maxSYM > maxhead
    then begin
        success := false;
        writeln("******** Too many grammar symbols = ",
        'Unable to calculate calculate head set ********");
    end
else begin
    for rowsym := 1 to maxSYM do
        for colsym := 1 to maxSYM do
            headmatrix[rowsym,colsym] := false;
    p := prodroot;
    while p <> nil do
        begin
            if p^.rhs <> nil then
                headmatrix[prodroot^.lhsid,p^.rhs^.id] := true;
            p := p^.next
        end;

}
updated := true;  \{Head matrix has been updated\}
while updated do
  begin
    updated := false;  \{Assume no updates made\}
    for rowsym := 1 to maxsym do
      for colsym := 1 to maxsym do
        if (symtype(rowsym) = nonterm) and (symtype(colsym) = nonterm)
          and headmatrix(rowsym, colsym)
        then
          for candsym := 1 to maxsym do
            if headmatrix(colsym, candsym)
              and not headmatrix(rowsym, candsym)
            then
              begin
                updated := true;
                headmatrix(rowsym, candsym) := true
              end
          end
      end
    end
    for rowsym := 1 to maxsym do
      headmatrix(rowsym, rowsym) := true;  \{Set diagonal true for reflexive transitive completion\}
  end  \{getheads\}
procedure writenodes;

begin
  { "headmatrix" has been defined.
  A report is printed that shows the head symbols of every symbol defined in the grammar. }

const rowsperpage = 35;
colsperpage = 43;

var colstring : string;
rowsym : integer;
colsym : integer;
firstcol : integer;
lastcol : integer;
umsections : integer;
section : integer;
1 : integer;

begin
  numsections := ((numnodes - 1) div colsperpage) + 1;

  for section := 1 to numsections do
  begin
    firstcol := 1 + (section - 1) * colsperpage;
    if (firstcol + colsperpage) <= numnodes
    then lastcol := firstcol + colsperpage
    else lastcol := numnodes;

    for rowsym := 1 to numnodes do
    begin
      if (rowsym mod rowsperpage) = 1
      then
        begin
          writeln(output);  
          writeln('');
          writeln;
          for i := 1 to maxstring do
          begin
            {Write a character from each string across page }
          end;

          {Repeat for each column}
          begin
            writeln(colstring[i], '');
            {Get string for this symbol}
            writeln('');
            {Write part of string}
          end;
          writeln;
          {Finish one line of heading}
write('Symbol    Type    Symbol  | ');
write('--------------------------------------------------------');
write('--------------------------------------------------------');
end;

write(rowsym:5, ' ');
if symtype(rowsym) == term then write('term   ');  (*write symbol id*)
else write('nonterm  ');  (*write symbol type*)
write(unfrowsym);  (*write string*)
write('| ');

for colsym := firstcol to lastcol do  (*Repeat for each column in this section*)
  if headmatrix(rowsym,colsym) then write('#   ');  (*Show that symbol is head symbol*)
  else write('   ');  (*Show that symbol isn't head symbol*)
  writeln
end
end  (*writeheads*)
{  
  Procedure readGrammar defines a number of local procedures to write labels corresponding to its defined types, to read and classify the input grammar, and to manipulate the finite state machine that defines the syntax of the production rules read in.

  writeclass
  writeaction
getchar
insertaction
lookupaction

procedure readGrammar(var startpld:integer; var success:boolean);

Pre: None.
Post: If 'success' is true then production rules have been built from the BNF read from file 'grammar'. 'startpld' points at the special production rule built with the internal start symbol as the left hand side. If 'success' is false then the text in file 'grammar' contains an error and an error message has been printed.
583  type bnfsmpnt = 'bnfs';
584  class = (entf,
585          endl,
586          space,
587          lt,
588          gt,
589          var,
590          dash,
591          letter,
592          digit,
593          escape,
594          special,
595          quote);
596
597  action = (null,
598          startsymbol,
599          additosymbol,
600          endlhs,
601          duplhs,
602          makelem,
603          endterm,
604          endnonterm,
605          endtermaccept,
606          endtermduplhs,
607          blanksymbol,
608          endnontermblank,
609          endnontermduplhs,
610          endnontermaccept,
611          accept,
612          error);
613
614  bnfs = record
615       state : integer;
616       class : class;
617       newstate : integer;
618       act : action;
619       next : bnfsmpnt
620       end);
621
622  var actionroot : bnfsmpnt;
623  newprod : prodpnt;
624  state : integer;
625  newstate : integer;
626  inchar : char;
627  act : action;
628  symbol : string;
629  lnside : integer;
630  termid : integer;
631  nontermid : integer;
632  xonfpnt : bnfsmpnt;
633  yonfpnt : bnfsmpnt;
634  sp : prodpnt;
procedure writeclass(c: class);
{
  Pre:  None;
  Post: 7 characters are written to represent the class of 'c',
}
begin
  case c of
  endf:    write("  endf");
  endl:    write("  endl");
  space:   write("  space");
  lt:      write("  lt");
  gt:      write("  gt");
  bar:     write("  bar");
  #ash:    write("  ash");
  letter:  write("  letter");
  digit:   write("  digit");
  escape:  write("  escape");
  special: write("  special");
  quote:   write("  quote");
end (case)
end (writeclass);
procedure writeaction(a: action);
{
    Pre: None.
    Post: 16 characters are written to represent the action 'a'.
}
begin
    case a of
        null: write("null");
        startsymbol: write("startsymbol");
        addtosomebol: write("addtosomebol");
        endins: write("endins");
        dupins: write("dupins");
        maketerm: write("maketerm");
        endterm: write("endterm");
        endnonterm: write("endnonterm");
        endtermaccept: write("endtermaccept");
        endtermdupins: write("endtermdupins");
        blanksymbol: write("blanksymbol");
        endnontermblank: write("endnontermblank");
        endnontermaccept: write("endnontermaccept");
        endnontermdupins: write("endnontermdupins");
        accept: write("accept");
        error: write("error");
    end (case)
end {writeaction};
procedure getchar(var ci:class; var c:char);
{
  Pre: File 'grammar' has been opened.
  Post: 'ci' is set to the class of the input character,
        'c' is the input character, unless 'ci' is 'endl'
        or 'endf'. The input character is echoed.
}

begin
  if eof(grammar)
  then cl := end
     {Class is end-of-file}
  else
    if eoin(grammar)
    then
      begin
        read(grammar,c);
        if vecho then writeln;
        cl := endl    {Class is end-of-line}
      end
    else
      begin
        read(grammar,c);
        if vecho then write(c);
        cl := special;       {Echo input character}
      end
    end
  if c in ['0'..'9'] then cl := digit;
  if (c in ['a'..'z']) or (c in ['A'..'Z']) then cl := letter;
  if c = ' ' then cl := space;
  if c = ',' then cl := lt;
  if c = '>' then cl := gt;
  if c = '/' then cl := bar;
  if c = '-' then cl := dash;
  if c = '#' then cl := escape;
  if c = '"' then cl := quote
end;
procedure insertaction(s:integer; c:class; n:integer; a:action);
{
  Pre: None,
  Post: A new entry is added to the linked list with root
        'actionroot'; the entry is made from ‘s’, ‘c’,
        ‘n’, and ‘a’.
}

var newaction: pointer;

begin
  new(newaction);
  newaction^.state := s;
  newaction^.class := c;
  newaction^.nextstate := n;
  newaction^.act := a;
  if actionroot = nil
    then newaction^.next := nil
  else newaction^.next := actionroot;
  actionroot := newaction
end {insertaction};
procedure lookupaction(st:integer; c: class; var n:integer; var a: action);
1049    {
1050        Pre: None.
1051        Post: If an entry corresponding to state 's' and class 'c'
1052              has been inserted in the linked list of finite state
1053              machine states with root 'actionroot' then 'n' and 'a'
1054              are the new-state and the action associated with this
1055              state; otherwise 'a' is set to 'error'.
1056    }
1057
1058    var p : bftfsmpt;
     {Point into list}
1060    begin
1061        p := actionroot;
1062        {Point at first entry}
1063        while (p <> nil) and not ((p^.state = s) and (p^.class = c))
1064            do p := p^.next;
1065            {Search for match}
1066        if p <> nil
1067            then
1068                n := p^.newstate;
1069                a := p^.act
1070                {Get new state}
1071            else
1072                a := error
1073                {No match found}
1074    end {lookupaction};
This is the main program for readgrammar.

begin

if echo then writeln("Input Grammar");
if echo then writeln("-----------------------------");
actionroot := nil;  {list of fsm entries is empty}
Nonterminals are enclosed in "<" and ">">" delimiters. The left hand side of a production rule is separated from the right hand side by "-->". The symbol "|" is interpreted as 'or'. Special symbols, such as "<", ">", and "|" may be included in a production as terminals if they are preceded by a "<". The following examples illustrate these points:

<number> --> <digit><digit> | <digit>
<digit> --> 0 | 1 | 2 | 9
<less> --> "<
<great> -->">
<space> --> " 
<cmgr> --> <less> | <less>= |
<great> | =<great>
```
1102 {
1103 <<<<<<< This Style of U&F is "commented out" >>>>>>>>

<table>
<thead>
<tr>
<th>Current State</th>
<th>Class of Input</th>
<th>New State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1108 insertaction(1, endf, 13, accept);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1109 insertaction(1, endl, 1, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110 insertaction(1, space, 1, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111 insertaction(1, lt, 2, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1112 insertaction(2, letdig, 3, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1113 insertaction(3, gt, 4, endlhs);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1114 insertaction(3, letdig, 3, addtosymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1115 insertaction(4, space, 4, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1116 insertaction(4, dash, 5, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1117 insertaction(5, dash, 6, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1118 insertaction(6, gt, 7, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1119 insertaction(7, endf, 13, accept);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1120 insertaction(7, endl, 1, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1121 insertaction(7, space, 7, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1122 insertaction(7, lit, 9, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1123 insertaction(7, bar, 8, duplhs);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1124 insertaction(7, dash, 11, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1125 insertaction(7, letdig, 11, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1126 insertaction(7, special, 11, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1127 insertaction(7, quote, 12, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1128 insertaction(8, endl, 7, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1129 insertaction(4, space, 8, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1130 insertaction(4, lit, 9, null);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1131 insertaction(8, dash, 11, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1132 insertaction(8, letdig, 11, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1133 insertaction(8, special, 11, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1134 insertaction(8, quote, 12, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1135 insertaction(9, letdig, 10, startsymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1136 insertaction(10, gt, 7, endnonterm);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1137 insertaction(10, letdig, 10, addtosymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1138 insertaction(11, endf, 13, endtermaccept);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1139 insertaction(11, endl, 1, endterm);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1140 insertaction(11, space, 7, endterm);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1141 insertaction(11, lit, 9, endterm);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1142 insertaction(11, bar, 8, endtermduplhs);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1143 insertaction(11, dash, 11, addtosymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1144 insertaction(11, letdig, 11, addtosymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1145 insertaction(11, special, 11, addtosymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1146 insertaction(11, quote, 11, addtosymbol);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1147 insertaction(12, space, 7, make term);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1148 insertaction(12, lit, 7, make term);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1149 insertaction(12, gt, 7, make term);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1150 insertaction(12, bar, 7, make term);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1151 insertaction(12, dash, 7, make term);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1152 insertaction(12, letdig, 7, make term);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1153 insertaction(12, special, 7, make term);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1154 insertaction(12, quote, 7, make term);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 1155 } 
```
{nonterminals aren't enclosed in delimiters. The left hand side of a
production rule is separated from the right hand side by "-->". The symbol
"|" is interpreted as 'or'. Terminals are enclosed in double quotes. Special
symbols, such as " may be included in a terminal if preceded by ". The
following examples illustrate these points:

  number --> digit digit | digit
  digit --> "0" | "1" | "2" | "9"
  less --> "<"
  great --> ">
  space --> " 
  quote --> ""
  cmp --> less | less= |
  great | =great
}

This style of HUF uses quotes around terminals, and a # as an escape character, for including things like " inside of terminals.

<table>
<thead>
<tr>
<th>Current State</th>
<th>Class of Input</th>
<th>New State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>endf,</td>
<td>11</td>
<td>accept</td>
</tr>
<tr>
<td>11</td>
<td>endi,</td>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>space,</td>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>letter,</td>
<td>2</td>
<td>startsymbol</td>
</tr>
<tr>
<td>11</td>
<td>digit,</td>
<td>2</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>letter,</td>
<td>2</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>space,</td>
<td>3</td>
<td>endlns</td>
</tr>
<tr>
<td>11</td>
<td>dash,</td>
<td>4</td>
<td>endlns</td>
</tr>
<tr>
<td>11</td>
<td>space,</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>dash,</td>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>space,</td>
<td>5</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>gt,</td>
<td>6</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>bar,</td>
<td>7</td>
<td>duplichs</td>
</tr>
<tr>
<td>11</td>
<td>letter,</td>
<td>8</td>
<td>startsymbol</td>
</tr>
<tr>
<td>11</td>
<td>space,</td>
<td>6</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>quote,</td>
<td>9</td>
<td>blanksymbol</td>
</tr>
<tr>
<td>11</td>
<td>endi,</td>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>endf,</td>
<td>11</td>
<td>accept</td>
</tr>
<tr>
<td>11</td>
<td>endi,</td>
<td>6</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>letter,</td>
<td>8</td>
<td>startsymbol</td>
</tr>
<tr>
<td>11</td>
<td>space,</td>
<td>7</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>quote,</td>
<td>9</td>
<td>blank</td>
</tr>
<tr>
<td>11</td>
<td>endi,</td>
<td>1</td>
<td>endnonterm</td>
</tr>
<tr>
<td>11</td>
<td>endf,</td>
<td>11</td>
<td>endnontermaccept</td>
</tr>
<tr>
<td>11</td>
<td>quote,</td>
<td>6</td>
<td>endterm</td>
</tr>
<tr>
<td>11</td>
<td>escape,</td>
<td>10</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>space,</td>
<td>9</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>li,</td>
<td>9</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>letter,</td>
<td>9</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>digit,</td>
<td>9</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>gt,</td>
<td>9</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>dash,</td>
<td>9</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>bar,</td>
<td>9</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>special,</td>
<td>9</td>
<td>addtosymbol</td>
</tr>
<tr>
<td>11</td>
<td>quote,</td>
<td>9</td>
<td>maketerm</td>
</tr>
<tr>
<td>11</td>
<td>escape,</td>
<td>9</td>
<td>maketerm</td>
</tr>
<tr>
<td>11</td>
<td>space,</td>
<td>9</td>
<td>maketerm</td>
</tr>
<tr>
<td>11</td>
<td>li,</td>
<td>9</td>
<td>maketerm</td>
</tr>
<tr>
<td>11</td>
<td>letter,</td>
<td>9</td>
<td>maketerm</td>
</tr>
<tr>
<td>11</td>
<td>digit,</td>
<td>9</td>
<td>maketerm</td>
</tr>
<tr>
<td>11</td>
<td>gt,</td>
<td>9</td>
<td>maketerm</td>
</tr>
<tr>
<td>11</td>
<td>dash,</td>
<td>9</td>
<td>maketerm</td>
</tr>
</tbody>
</table>
1232  reset(grammar,"grammar");
1233  initSymbols;
1234  initProds;
1235  state := 1;
1236  repeat
1237     getChar(inclass,inchar);
1238     lookupAction(state,inclass,newstate,act);
1239     if then
1240         begin
1241             write(" state = ",state4," class = ");
1242             writeclass(inclass);
1243             write(" newstate = ",newstate4);
1244             write(" action = ");
1245             writeAction(act);
1246         writeln;
1247         end;
1248     case act of
1249         accept,
1250             null;
1251         blanksymbol:
1252             blank(symbol);
1253         startsymbol:
1254             begin
1255                 blank(symbol);
1256                 append(inchar,symbol)
1257             end;
1258         addtosymbol:
1259             begin
1260                 append(inchar,symbol);
1261             end;
1262         endings:
1263             begin
1264                 insertNonsymbol(symbol,nonterm,lhsid);
1265                 createprod(lhsid,newprod)
1266             end;
1267         endnonterm=accept,
1268         endnonterm:
1269             begin
1270                 insertNonsymbol(symbol,nonterm,nontermid);
1271                 appendNonterm(nontermid,newprod)
1272             end;
1273         endnonterm=blank:
1274         begin
1275                 insertNonsymbol(symbol,nonterm,nontermid);
1276             end;
if act = error
then success := false
else success := true;

Delete the finite state machine used to build the production rules.

xbnpnt := actionroot;
while xbnpnt <> nil do
begin
ybnpnt := xbnpnt;
xbnpnt := xbnpnt^.next;
dispose(ybnpnt)
end;

{Point at first entry}
{Copy pointer to entry}
{Point at next entry}
{Release memory}
The special production rule `<s> --> <G> cf` is added to the grammar, where `<s>` is the internal start symbol, `cf` is the "chicken foot", and `<G>` is the user's start symbol (the left hand side of the first rule read).

1365 blank(symbol);
1366 append(ss,symbol);
1367 insertnfssymbol(symbol,nonterm,ssid);

1369 blank(symbol);
1370 append(cf,symbol);
1371 insertnfssymbol(symbol,term,cfid);

1373 createprou(ssid,sp);
1374 appendterms(1,sp);
1375 appendterms(cfid,sp);

1377 startpid := sp","id

1379 end (readgrammar);
Configuratoin package

*** These procedures manipulate individual configurations and configuration sets.

first
findconfiginset
createconfig
mergelasets
addconfigs
deleteconfig
configequal
advanceconfig
procedure first(symbolid: integer; var lahead: lasettyp);
{
  Pre: "symbolid" is the symbol id of a terminal or nonterminal
  present in "headmatrix".
  Post: "lahead" is the set of symbols that are possible
  look ahead symbols for "symbolid".
}

var canusym: integer;  \{Symbol id of a symbol that is a candidate for the look ahead list\}

begin
  lahead := {};
  for canusym := 1 to numnfsym do
    if headmatrix[symbolid, canusym] then lahead := lahead + {canusym}  \{Add new symbol to set\}
end (first);
procedure findconfiginset(pid:integer; mark:integer; cset:configsetpnt);
    var cfound:configsetpnt; var found:boolean);

{ Pre: None. 
Post: If 'found' is false then there is no configuration in 
set 'cset' with production id "pid" and mark "mark". If 
'found' is true then 'cfound' points at the configuration 
with production id "pid" and mark "mark".

ueqin

  cfound := cset; \{Point at first configuration\}
  found := false; \{Match not found yet\}
  while (cfound <> nil) and not found do \{Check every entry in set\}
      if (cfound^.id = pid) and (cfound^.mark = mark)
          then found := true \{Indicate match found\}
      else cfound := cfound^.next; \{Point at next configuration\}
  end (findconfiginset);
procedure createconfig(prodid:integer; mark:integer; laset:lasettyp);
var c:configsetpnt);
{
Post: 'c' points to a new configuration describing a production
with production id 'prodid', scan marker 'mark', and
look ahead set 'laset'.

begin
new(c);
\{Create empty configuration\}
\{Fill in production id\}
\{Fill in scan marker\}
\{Fill in look ahead set\}
\{This is last configuration in set\}
begin (createconfig)\}
procedure mergelasets(var lasetout:lasettyp; laset:lasettyp;
var updated:boolean);
{
    Pre: None.
    Post: The look ahead symbols in set 'laset' are merged with
          the look ahead symbols already present in the look ahead set
          'lasetout'. 'updated' is true only if items were moved
          from 'laset' to 'lasetout'.
}

var temp : lasettyp;

begin
    temp := lasetout + laset;
    if temp <> lasetout then updated := true
    else updated := false;
    lasetout := temp
end {mergelasets};
procedure addconfig(cset:configsetpnt; lhsid:integer; laset;lasettyp);
  var updated:boolean;
  {
    All the productions that have the symbol 'lhsid' as a
    left hand side are turned into configurations and
    added to the end of 'cset'. Each of the new configurations
    has the scan marker in front of the first character
    of the right hand side. Each of the new configurations has
    the look ahead symbols in 'laset', 'updated' is true only
    if 'cset' was modified.
  }
  var newconfig : configsetpnt;
  s : configsetpnt;
  canprod : prodpnt;
  oldconfig : configsetpnt;
  found : boolean;
  mergeupdate : boolean;
  lasetcopy : lasettyp;

begin
  updated := false;
  s := cset;
  if s <> nil then
    begin
      while s^.next <> nil do s := s^.next;
      canprod := prodroot;
      while canprod <> nil do
        begin
          if canprod^.lhsid = lhsid then
            begin
              findconfiginset(canprod^.id,1,cset,oldconfig,found);
              if found then
                begin
                  mergelsets(oldconfig^.laset,laset,mergeupdate);
                  if mergeupdate then updated := true
                end;
              end
            else
              begin
                updated := true;
                lasetcopy := laset;
                createconfig(canprod^.id,1,lasetcopy,newconfig);
                s^.next := newconfig;
                s := newconfig
              end
            end;
          end;
        end;
      canprod := canprod^.next
    end
end {addconfig}
procedure deleteconfig(x:configsetpnt; var z:configsetpnt; var found: boolean);
{
If the first configuration in 'x' is not present in 'z' then
'found' is set to false. If a configuration in 'z' corresponds
to the first configuration in 'x' then the configuration in
'z' is deleted.
}

var lastz : configsetpnt;
curz : configsetpnt;

begin
lastz := nil;
curz := z;
found := false;

while (curz <> nil) and not found do
{Check every entry in z}
if (curz^.id = x^.id) and (curz^.mark = x^.mark)
and (curz^.laset = x^.laset)
then
found := true
{Indicate match found}
else
begin
lastz := curz;
curz := curz^.next
{Lastz lags behind current pointer}
end;

if found then
begin
if lastz = nil
then z := curz^.next
else lastz^.next := curz^.next;
dispose(curz);
end
end (deleteconfig);
procedure config_equal(a,b:configsetpnt; var equal:boolean);

if 'equal' is true if 'a' and 'b' have the same members.

var x : configsetpnt;
y : configsetpnt;
z : configsetpnt;
delont : configsetpnt;
lastconfig : configsetpnt;
newconfig : configsetpnt;
din : boolean;

begin
  x := a;
y := b;
lastconfig := nil;

begin
  assue success
  point at first member of a
  point at first member of b
  No last configuration yet

  (Duplicate one of the input configuration sets.

  while y <> nil do
  begin
    new(newconfig);
    newconfig^.id := y^.id;
    newconfig^.mark := y^.mark;
    newconfig^.laset := y^.laset;
    newconfig^.next := nil;
  if lastconfig = nil
     then z := newconfig
     else lastconfig^.next := newconfig;
    lastconfig := newconfig;
y := y^.next
  end;
end;

(points into set a)
(points into set b)
(points into copy of b)
(points at entry to delete)
(points at previous configuration)
(points at new configuration)
(false if couldn't delete)
{  
  delete configurations one at a time from the duplicate configuration set. Make sure every member of 'x' has a corresponding member in the duplicate. 
}

while (x <> nil) and din do 
  begin
  deleteconfig(x,z,ain);
  x := x^.next
  end;

if 'x' and 'z' didn't have exactly the same members, then 'z' may still contain some entries which must be deleted.

if (x <> nil) or (not din) or (z <> nil) then 
  begin
    equal := false;
    while z <> nil do 
      begin
        delpnt := z;
        z := z^.next;
        dispose(delpnt)
      end
    end
  else 
    equal := true
end {configequal};
procedure advanceconfig(cset:configsetpnt; var newcs:configsetpnt);

begin
  newcs := cset;
  newcs.id := cset.id;
  newcs.mark := cset.mark + 1;
  newcs.lset := cset.lset;
  newcs.next := nil
end {advanceconfig};
State package

This procedures are used to manipulate the
the configuration states.

initstates
createstate
deletestate
writestates
createmidstate
procedure initstates;
{
  Pre: None.
  Post: The state package is ready for use.
}

begin
  laststate := 0
  (Initialize number of states)
end (initstates);
procedure createstate(cstate; configsetptn; var newstate; stateptn);
()
    Pre: 'Initstate' has been called.
    Post: 'newstate' points at a new state that corresponds
to the configuration set 'cstate'.
)
begin
    newstate := laststate;
    newstate.snum := laststate + 1;
    newstate.cset := cstate;
    newstate.next := nil;
end {createstate};
procedure deletestate(s:statepnt);
{
    Pre: None.
    Post: All the configurations linked to 's' are deleted.
}

var c : configsetpnt;
d : configsetpnt;
ok : boolean;

begin
    c := s^.cset;
    while c <> nil do
        begin
            d := c;
            c := c^.next;
            deleteconfig(d,d,ok);
        end
end {deletestate};
procedure writestates(stateroot:statепnt);
{ Pre: none.
 Post: All the states in the list with root 'stateroot' are listed.
}

var p : prodпnt;
rhsпnt : rhs+listпnt;
rnscount : integer;
stateпnt : statепnt;
c : configsetпnt;
i : integer;
nамe : String;
l : integer;
statenum : integer;
chars : integer;

begin
page(output);
writeln("Configurations Sets");
writeln("-------------------------");
writeln;
writeln;
s := stateroot;
statenum := -1;
while s <> nil do
begin
statenum := statenum + 1;
writeln("******* State ",statenum," *******");
end;
writeln;
c := s^.cset;
while c <> nil do
begin
lookupprod(c^.id,p);
writeofssymbol(p^.lhs.id);
write(" -> ");
chars := 17;
rhsпnt := p^.rhs;
rnscount := 0;
while rhsпnt <> nil do
begin
rnscount := rnscount + 1;
if rnscount = c^.mark then
begin
writeln(" ");
chars := chars + 2
end;
if symtype(rhsпnt^.id) = term then
begin
writeln(" ");
chars := chars + 1
end;
symname(rhsпnt^.id,name);
for i := 1 to maxstring do
  if name[i] <> ' ' then
    begin
      write(name[i]);
      (Print symbol's name)
      chars := chars + 1
      (Count a character)
      end;
    if symtype(rhspnt^.id) = term
    then
      begin
        write('"');
        (Print terminal close quote)
        chars := chars + 2
        (Count 2 characters)
      end;
    else
      begin
        write('"');
        (Separate nonterminals)
        chars := chars + 1
        (Count a character)
        end;
    rhspnt := rhspnt^.next
    (Point at next symbol)
  end;
if rhscount + 1 = c^.mark then
  begin
    write('"');
    (Print scan marker)
    chars := chars + 1
    (Count a character)
  end;
if c^.laset <> [] then
  begin
    for i := chars to 50 do write(" ");
    for i := 1 to numnfsym do
      if i in c^.laset then writeonfformat(1);
      (Print symbol's name)
      (Finish line)
    writeln
    end;
  c := c^.next
  (Point at next configuration)
end;
s := s^.next
  (Point at next state)
end (writestates);
procedure createMidState(var midstate: stateptnt);
{
    Pre:  "stateroot" is the root of a list of states produced
          by applying the $LH(1)$ algorithm to the input grammar.
    Post: "midstate" is a set of configurations that is
          the union of all the configuration sets produced
          by the standard $LH(1)$ algorithm - except that all
          the configurations that represent reductions are
          excluded.
}

var cfound : configsetptnt;
found : boolean;
curstate : stateptnt;
curconfig : configsetptnt;
newcs : configsetptnt;
imidconfigs : configsetptnt;
s : integer;
symboltype;
readconfig : boolean;

begin
    midconfigs := nil;
curstate := stateroot;
while curstate <> nil do
begin
    curconfig := curstate^.cset;
while curconfig <> nil do
begin
    getsymbolmark(curconfig,s,t,readconfig);(Try to get symbol after mark)
    if readconfig then (Reduces are ignored)
    begin
        findconfiginset(curconfig^.id,curconfig^.mark,midconfigs,cfound,found);
        if found then
            cfound^.laset := cfound^.laset + curconfig^.laset
        else
            begin
                new(newcs);
                newcs^.id := curconfig^.id;
                newcs^.mark := curconfig^.mark;
                newcs^.laset := curconfig^.laset;
                newcs^.next := midconfigs;
                midconfigs := newcs (New configuration is now first in set)
            end;
    end;
curconfig := curconfig^.next
end;
curstate := curstate^.next
end;
createState(midconfigs,midstate)
createMidState;
1670 { ***
1671 """"
1672 """"
1673 ***
1674 *** Finite state machine package ***
1675 ***
1676 *** These procedures are used to manipulate the finite ***
1677 *** state machine that is produced to parse the input ***
1678 *** text.
1679 ***
1680 *** initfsm ***
1681 *** createfsmstate ***
1682 *** fixupfsmtable ***
1683 *** renumberfsmtable ***
1684 *** checkfsmtable ***
1685 *** buildfsmkeys ***
1686 *** writefsm ***
1687 *** lookufsm ***
1688 ***
1689 ***
1690 ***
1691 } ***
procedure initfs;
{
    Pre: None.
    Post: The finite state machine package is ready for use.
}

begin
lastfsm := 0
(Initialize number of states)
end {initfs};
procedure create_fsm_state(curstate:integer; symbold:integer; la1t:laset; la1ttypetypes; 
statetype:parseaction; prod:integer; newstate:integer; 
var success:boolean); 
1906 { 
1907 Pre: "initfs" has been called. 
1908 Post: If "success" is true then a new entry corresponding 
1909 to "curstate", "symbold", "laset", "statetype", "prod", 
1910 and "newstate" has been added to the finite state machine, 
1911 If "success" is false then the table is too small; 
1912 an error message is printed in this case. 
1913 } 

1915 var found : boolean; 
1916 l : integer; 
1917 ok : boolean; 

1919 begin 
1920 l := 0; 
1921 found := false; 
1922 while (l < lastfs) and not found do 
1923 begin 
1924 l := l + 1; 
1925 if (curstate = fsmtab[l],statenum) and 
1926 (statetype = fsmtab[l],action) 
1927 then 
1928 if statetype = pass 
1929 then 
1930 begin 
1931 if (symbold = fsmtab[l],token) and 
1932 (newstate = fsmtab[l],newstate) 
1933 then 
1934 begin 
1935 found := true; 
1936 mergelsets(fsmtab[l],laset,laset,ok); 
1937 end 
1938 end 
1939 else 
1940 if prod = fsmtab[l],prod then 
1941 begin 
1942 found := true; 
1943 mergelsets(fsmtab[l],laset,laset,ok); 
1944 end; 
1945 end;
if not found then begin
  success := true;
  lastism := lastism + 1;
  if lastism > maxfsm then begin
    success := false;
    writeln('***** Grammar is too complex to fit in fsm table *****')
  end
else begin
  lastfsm := lastfsm + 1;
  fs@table(lastfsm).statenum := curstatenum;
  fs@table(lastfsm).token := symbolid;
  fs@table(lastfsm).laset := laset;
  fs@table(lastfsm).action := statetype;
  fs@table(lastfsm).prod := prod;
  fs@table(lastfsm).newstate := newstate
end
end (createfsmstate);
procedure fixupfsmtable(sdel:integer; srep:integer);
1971  {
1972      The state numbered 'sdel' is being deleted because it
1973      duplicates the state numbered 'srep'. Fix the fsmtbl
1974      table to reflect the replacement.
1975      'midstatenumber' is updated to reflect deleted states,
1976  }
1977  var i : integer;
1978  begin
1979  if sdel = midstatenumber then midstatenumber := srep;
1980  for i := 1 to laststate do
1981     begin
1982     if fsmtbl[i].statenumber = sdel then fsmtbl[i].statenumber := srep;
1983     if fsmtbl[i].newstate = sdel then fsmtbl[i].newstate := srep
1984     end
1985  end;  //fixupfsmtable;
procedure renum(ers*table);  
{

The report of the finite state machine should use consecutive
numbers for the states.
}
var i : integer;
curstate : integer;

begin
  curstate := 0; {first state expected}
  for i := 1 to lastfsm do
    if fsmstate[i],statenum <> curstate then
      if fsmstate[i],statenum = curstate + 1
      then curstate := curstate + 1
      else
        begin
          curstate := curstate + 1;
          fixupfsmtable(fsmstate[i],statenum,curstate)
        end;
end {renum(ers*table)};
procedure checkfsmtab(e(midstatenum:integer; var success:boolean));
{
Pre: 'fsmtab' contains entries in its first
'lastfsm' places,
Post: States that don't have enough information to
determine what reduction should be applied are
marked as 'error' if they are in the standard
unital machine, or 'stutter' if they are the midstate
or any of its successors. If any state is marked
as 'error' then 'success' is false; otherwise
'success' is true.
}
var candentry: integer; (Candidate for error/stutter)
cureentry: integer; (Current fsm entry)
curstatenum: integer; (Current entry's state number)

begin
success := true; (Assume success)
cureentry := 1; (Start with first entry)

{ All the entries for a state are adjacent to each other in 'fsmtab'.
  Check every entry against all the other entries in the same state to
  see if an 'error' or 'stutter' condition exists.
}
while cureentry <= lastfsm do
  (Check every entry for error/stutter)
  begin
    curstatenum := fsmtab[cureentry].statenum; (Get current state number)
    candentry := cureentry + 1; (First candidate follows current entry)
    while curstatenum = fsmtab[candentry].statenum do
      begin (Check all entries in same state)
        { Two entries for the same state that signify reductions
          require attention if their look ahead sets intersect.
        }
        if (fsmtab[cureentry].action <> pass)
          and (fsmtab[candentry].action <> pass)
          and (fsmtab[cureentry].last <> fsmtab[candentry].last <> []))
          then
            if curstatenum < midstatenum
            then
              begin
                fsmtab[cureentry].action := error; (Not enough information to pick correct entry)
                fsmtab[candentry].action := error; (Not enough information to pick correct entry)
              end
            else
              begin
                fsmtab[cureentry].action := error; (Not enough information to pick correct entry)
                fsmtab[candentry].action := error; (Not enough information to pick correct entry)
              end
            end
          end
        else
          begin
            candentry := candentry + 1; (Point at next candidate)
          end
        cureentry := cureentry + 1; (Point at next entry)
      end
    end
  end (checkfsmtab);
procedure build_fsm_keys;
{
Pre: 'fsmtabell' to 'fsmtabellast' are defined
and represent a set of consecutive states numbered
from 0 to 'fsmtabellast',statenum'.
Post: 'keys[0]' to 'keys[fsmtabellast],statenum' are
defined and contain the index into 'fsmtabell' of
the first entry for a given state - 'key[i]' is
the first entry in 'fsmtabell' for state 'i'.}

var s : integer;
curindex : integer;

begin
curindex := 1;
for s := 0 to fsmtabellast,statenum do
begin
keys[s] := curindex;
while fsmtabell[curindex],statenum = s do
curindex := curindex + 1
end
end (build_fsm_keys);
procedure writefsms;
{
Print the entries that comprise the Finite State Machine.
}
var l : integer;
l : integer;
begin
page(output);
{write a form feed}
writeln('Finite State Machine Generated From Grammar');
writeln('Current Symbol Action State Number');
writeln('Look Ahead Set');
 writeln('-----------------------------------');
 writeln('-----------------------------------');
 writeln;
for i := 1 to lastfsm do
begin
writeln(fsmtable[i],statenum:6,'');
if fsmtable[i].statenum = midstatenum
then writeln('mid ')
else writeln(' ');
case fsmtable[i].action of
pass:
begin
writeائفsymbol(fsmtable[i],token);
write('read',fsmtable[i],newstate:5,'')
end;
apply:
write('reduce',fsmtable[i],prod:11);
error:
write('error',fsmtable[i],prod:11);
stutter:
write('stutter',fsmtable[i],prod:11);
end (case);
end (if)
if (fsmtable[i].lstat <> [])
and ((fsmtable[i].action = apply) or (fsmtable[i].action = stutter))
then
begin
writeln('');
for i := 1 to numnfsym do
if l in fsmtable[i].lstat then writeln(format(',lstat'));
end;
writeln
end
end (writefsms);
procedure lookupFSM(st: integer; c: integer; var a: parseaction; var n: integer;)
begin
  var p: integer; var success: boolean;
  if there is no appropriate entry,
  for i := 1 to found do
    if keys[i] = c then
      begin
        a := fsmtable[i].action;
        n := fsmtable[i].newstate;
        p := fsmtable[i].prod;
        found := true;
      end;
    else lastentry := i + 1;
  end;
end;
it not found then
begin
writeln('***** No finite state machine entry for state ',$15,' symbol ','c,
'*****');
success := false
end
end {lookup()};
LH(1) package

These routines use the other packages to generate a finite state machine from the input NNF grammar and use this finite state machine to parse the input text.
procedure setcomplete(s:statept);
  ("cset" is completed.

  var current : confisetpt;
  rnstoken : integer;
  rntype : stype;
  ok : boolean;
  rnstokeni : integer;
  rntypeli : stype;
  updated : boolean;
  addupdated : boolean;
  firstrnstokeni : lasettype;

begin
  repeat
    updated := false;
    current := s",cset;
    while current <> nil do
      begin
        getsymaftermark(current,rnstoken,rntype,ok);
        if ok then
          if rntype = nonterm then
            begin
              getsymaftermark(current,rnstokeni,rntypeli,ok);
              if ok then
                begin
                  first(rnstokeni,firstrnstokeni);
                  addconfigs(s",cset,rnstoken,firstrnstokeni,
                  addupdated,addupdated);
                  if addupdated then updated := true
                end
              else
                begin
                  addconfigs(s",cset,rnstoken,current",laset,
                  addupdated,addupdated);
                  if addupdated then updated := true
                end
            end
          end
        end
      until not updated
  end (setcomplete);
procedure successor(stateroot:statepnt; instate:statepnt; var success:boolean);
{
  This procedure calculates all the successor states to the
  state 'instate' and adds them to the list headed by
  'stateroot'. After all the successor states have been
  built entries are made in the finite state machine tables,
  and then duplicate states are deleted. 'Success' is false
  if the finite state machine's table has overflowed; an error
  message has been printed to note the error.
}

var curcon : configsetpnt;
newcs : configsetpnt;
rsctype : symboltype;
olddone : integer;
oldtype : symboltype;
s : statepnt;
newstateroot : statepnt;
found : boolean;
ok : boolean;
old : statepnt;
deleted : boolean;
exts : statepnt;
lst : statepnt;
equal : boolean;
pasconfly : boolean;
curset : lasettyp;
advancedover : integer;

(Points into configuration set)
(Points at new configuration set)
(Type for root symbol)
-Token for a state)
(Points into list of states)
(Root of list of new states)
(Found a new state flag)
(Mark is past end of production)
(Points at list of old state in list)
(State deleted flag)
(Points at next state)
(Points at recent state in list)
(States equal flag)
(Indicate that configuration reflects a 'pass')
(Points at duplicate of a lookahead set)
(Symbol Id of symbol advanced over to get to a new state)

Each possible transition to one of the new states is
recorded in the finite state machine tables.

The symbol in front of the scan marker of the first configuration
in a state indicates what symbol causes a transition to the new
state. The finite state machine is used to record that a
transition is possible from the current state to each of the newly
derived states.

Each configuration in the newly derived states indicates that either
a 'pass' (read) or 'apply' (reduce) should be performed. These two
types of actions are distinguished by the position of the scan
mark - if the scan mark is past the last symbol of a configuration
then the configuration represents an 'apply'.

The lookahead set for each configuration may be required to
distinguish between possible actions - this information is therefore
passed on to the finite state machine.
begin
newstateroot := nil;
curcon := instate\*,cset;
while curcon <> nil do
begin
getsymbolmark(curcon,advancedover,instype,passconfig);
if passconfig then
begin
first(advancedover,curlaset);
advanceconfig(curcon,newcs);
found := false;
s := newstateroot;
while (s <> nil) and not found do
begin
getsymbolbeforemark(s\*,cset,oldtoken,oldtype,ok);
if advancedover = oldtoken then
begin
createsmstate(instate\*,snun,advancedover,curlaset,
newcs\*,next := s\*,cset;
s\*,cset := newcs;
founds := true;
end;
s := s\*,next;
end;
end;
if not found then
if newstateroot = nil then
begin
createstate(newcs,newstateroot);
createfsmstate(instate\*,snun,advancedover,curlaset,
pas,0,newstateroot\*,snun,succ);
end;
else
begin
s := newstateroot;
while s\*,next <> nil do s := s\*,next;
createstate(newcs,s\*,next);
createfsmstate(instate\*,snun,advancedover,curlaset,
pas,0,s\*,next\*,snun,succ);
end;
end;
end;
else
begin
curlaset := curcon\*,laset;
createfsmstate(instate\*,snun,0,curlaset,
apply,curcon\*,id,0,succ);
end;
end;
curcon := curcon\*,next
end;
2343 }  
2344   Complete each of the new states.
2345 }  
2346 s := newStatert;  
2347 while s <> nil do  
2348 begin  
2349 setcomplete(s);  
2350 s := s^.next  
2351 end;  
2352  
2353 if success then  
2354 begin  
2355 delete new states that are duplicates.  
2356 }  
2357 old := statert;  
2358 while old <> nil do  
2359 begin  
2360 s := newStatert;  
2361 last := nil;  
2362 deleted := false;  
2363 while (s <> nil) and (not deleted) do  
2364 begin  
2365 nexts := s^.next;  
2366 configuequal(s^.cset,old^.cset,equal);  
2367 if equal then  
2368 begin  
2369 fixupfshtable(s^.snum,old^.snum);  
2370 if last = nil  
2371 then newStatert := s^.next;  
2372 delete state(s);  
2373 dispose(s);  
2374 deleted := true  
2375 end  
2376 else last := s;  
2377 s := nexts  
2378 end;  
2379 old := old^.next  
2380 end;  
2381  
2382 s := statert;  
2383 while s^.next <> nil do s := s^.next;  
2384 s^.next := newStatert  
2385 end;  
2386 end (successor);
procedure initstack;

    (Pre: None,
     Post: The stack package is ready for use.)

    var i : integer;

    begin
    for i := 1 to maxtext do
        parsestates[i] := empty
    end (initstack);
procedure readsymbols;

Pre: 'bntsymb0ot' is the root of a list of terminals and
nonterminals defined by the grammar read from
file 'grammar'.
Post: 'sentform' is filled with symbols representing the
input text. All unused positions are set to 'cfd',
'lastchar' indicates the last symbol of text.

var symbol : string;
c : character;
match : boolean;
spnt : bntsymblistpnt;
i : integer;
cursym : integer;

begin
reset(intext,'text');
if vtext then
begin
page(output);
writein('-------------------------------------------------------------------------------'

Input Text'

'-------------------------------------------------------------------------------'

end;
for i := 1 to maxtext do sentform(i) := cfd;
cursym := 0;
while not eof(intext) do
  begin
    cursym := cursym + 1;
    (Point at next symbol)
    match := false;
    (No match found yet)
    i := 0;
    (No characters added yet)
    blank(symbol);
    (Set symbol to blanks)
    while (i < maxstring) and not match and not eof(intext) do
      if eoln(intext)
        then
          begin
            read(intext,c);
          (?Throw away dummy character)
            if vtext then writeln
            begin
            end
          end
        else
          begin
            i := i + 1;
            (Count a character)
            read(intext,c);
            (Read a character)
            if vtext then write(c);
            (Append new character to symbol)
            append(c,symbol);
            (Point at first symbol)
            spnt := bnfexprroot;
            while (spnt^.next <> nil) and not
              (symbolequal(symbol,spnt^.name) and (spnt^.type = term)) do
                spnt := spnt^.next;
                (Try to find match)
            if symbolequal(symbol,spnt^.name) and (spnt^.type = term)
              then
                begin
                  sentform(cursym) := spnt^.symid;
                (?Return symbol Id of match)
                  match := true
                (?Note match found)
                end
            end
          end
        end
      if vtext then writeln;
      lastchar := cursym + 1
      (Record last character to parse)
    end (readsymbols);
function cursymbol (curproc:integer):integer;
{
    Pre: processor[curproc].curtext' points at the current input symbol (or an empty symbol to its left). The current state is the one that corresponds to the current symbol -- which may be later in the text.
    Post: 'symbol' is the current symbol.
}

begin
    while sentform[processor[curproc],curtext] = empty do
        processor[curproc].curtext := processor[curproc].curtext + 1; {Find current symbol}
    cursymbol := sentform[processor[curproc],curtext]; {Return top state}
end (cursymbol);
function curstate(curproc:integer):integer;
{
  Pre:  'processor(curproc),curtext' points at the current input
        symbol (or an empty symbol to its left). The current state
        is the one that corresponds to the current symbol - which
        may be later in the text.
  Post: 'curstate' is the current state.
}

begin
  while sentforw(processor(curproc),curtext) = empty do
      processor(curproc),curtext := processor(curproc),curtext + 1; {Find current symbol}
  curstate := parsestates(processor(curproc),curtext) {Return top state}
end {curstate};
procedure assignproc;
{
Pre:  'lastchar' is the last position in 'sentform' that
      is to be used in parsing it is a 'chicken foot',
Post: The 'numproc' processors are assigned segments of
      'sentform' to parse.
}

var charsperproc : integer;
leftover : integer;
proc : integer;

begin
readin(inpara,m,numproc);

if numproc < 1 then numproc := 1;
if numproc > maxproc then numproc := maxproc;
if lastchar < numproc then numproc := lastchar;
charsperproc := (lastchar) div numproc;
leftover := (lastchar) mod numproc;

processor[i].active := true;
processor[i].left := 1;
processor[i].curtext := processor[i].left;
if leftover >= 1
then processor[i].right := charsperproc + 1
else processor[i].right := charsperproc;
parsestates(processor[i],left) := 0;

for proc := 2 to numproc do
begin
processor[proc].active := true;
processor[proc].left := processor[proc - 1].right + 1;
processor[proc].curtext := processor[proc].left;
if leftover >= proc
then processor[proc].right := processor[proc].left + charsperproc
else processor[proc].right := processor[proc].left + charsperproc - 1;
parsestates(processor[proc],left) := midstatenum
end;
2585 It was so then
2586 begin
2587 writeln('assignment of processors');
2588 writeln('number of characters in input text:', lastchar);
2589 for proc = 1 to numproc do
2590 writeln('processor(', proc, '):', processor[proc]);
2591 savproc = 0;
2592 for proc = 1 to numproc do
2593 processor[(proc)right] = processor[(proc)right] + savproc;
2595 end;
2596 end;
procedure schedule (var curproc: integer);
{
    Pre:  0 <= 'savcurproc' <= 'numproc',
    Post: (1 <= 'savcurproc' <= 'numproc') and ('savcurproc' = 'curproc')
    ('processor[curproc].active' = true).

    One of the active processors is chosen.
}
begin
repeat
    begin
        savcurproc := savcurproc + 1;
        {Try to use next processor}
        if savcurproc > numproc then savcurproc := 1
        {Use first processor again}
        until processor[savcurproc].active;
        {Skip inactive processor}
        curproc := savcurproc
        {Return value of processor assigned}
end (schedule);
procedure killproc(curproc:integer);
{
    Pre: (1 <= curproc <= numproc) and (processor[curproc].active = true).
    Post: 'processor[curproc].active' is false. The territory belonging
to this processor is given to its nearest active left-hand
neighbor. If no such neighbor exists, the leftmost processor
takes over the territory, and is restarted.
    If 'curproc' is 1 then the leftmost processor is backed up to
the state it was in before it poached on its neighbor's
territory, so that it may begin in the proper state when it
is restarted.
}

var found : boolean;
proc : integer;
begin
    processor[curproc].active := false;
    if curproc = 1
    then begin
        processor[curproc].curtext := processor[curproc].curtext - 1;
        while sentform(processor[curproc].curtext) = empty do
            processor[curproc].curtext := processor[curproc].curtext - 1
    end
    else begin
        proc := curproc;
        found := false;
        while (proc >= 2) and not found do
            begin
                proc := proc - 1;
                if processor[proc].active then found := true
            end;
        if found
            then begin
                processor[proc].right := processor[curproc].right
            end
            else begin
                processor[1].right := processor[curproc].right;
                processor[1].active := true
            end
    end
end (killproc);
procedure stutterproc(curproc:integer);
{
  Pre: (1 <= "curproc" <= numproc) and
  (processor[curproc].active = true).
  Post: The processor identified by "curproc" is restarted on
  its current symbol, but in the midstate, its lefthand
  neighbor takes over the territory it abandons.
}

var proc : integer;
found : boolean;

begin
while sentfor(processor[curproc],curtext) = empty do
  processor[curproc].curtext := processor[curproc].curtext + 1;
if processor[curproc].curtext >= processor[curproc].right
  then
    killproc(curproc)
  else
    begin
      processor[curproc].left := processor[curproc].curtext;
      parsestates(processor[curproc].left) := midstatenum;
    end;
  proc := curproc;
  found := false;
  while (proc >= 2) and not found do
    begin
      proc := proc - 1;
      if processor[proc].active then found := true
    end;
  if found
    then
      processor[proc].right := processor[curproc].left - 1
    else
      begin
        processor[1].right := processor[curproc].left - 1
        processor[1].active := true
      end
end (stutterproc);
2712 function reduceWithoutPach(curproc: integer; rhs: rnsym*lstpnt): boolean;
2713 {
2714     Pre: 1 <= 'curproc' <= 'numproc',
2715     'rhs' points at the first right hand symbol of a production,
2716     Post: 'reduceWithoutPach' is true if a symbol in 'curproc's'
2717     territory between 'processor[curproc],left' and
2718     'processor[curproc],curtext' exists for every symbol
2719     in the 'rhs' list.
2720 }

2722 var rhscounl : integer;
2723     currhs : rnsym*lstpnt;
2724     textpnt : integer;
2725     symcount : integer;

2727 begin
2728 rhscounl := 0;
2729 currhs := rhs;
2730 while currhs <> nil do
2731     begin
2732         rhscounl := rhscounl + 1;
2733         currhs := currhs^.next
2734     end;
2735 symcount := 0;
2736 textpnt := processor[curproc],curtext - 1;
2737 while (textpnt >= processor[curproc],left) and (symcount < rhscounl) do
2738     begin
2739         if sentform(textpnt) <> empty then symcount := symcount + 1;
2740         textpnt := textpnt - 1
2741     end;
2742 if symcount >= rhscounl
2743 then reduceWithoutPach := true
2744 else reduceWithoutPach := false
2745 end (reduceWithoutPach);
procedure pushstate(curproc:integer; state:integer; var poach:boolean);
  var success:boolean);
{
  Pre: "initstack" has been called.
  Post: If "success" is true and "poach" false then "state" has
  been stacked, and "processor[curproc],curtext" is advanced
  to the next input symbol, which becomes the new current symbol,
  and "state" is stored, becoming the new current state.
  If "success" is false then an error message has been
  printed.
  If "poach" is true then "pushstate" couldn’t operate without
  encroaching on another processor’s territory; no action has
  been taken.
}
begin
  if processor[curproc],curtext >= maxtext
    then begin
      success := false;
      writeln("***** Parse stack overflow *****")
      end
  else begin
    success := true;
    while sentforms(processor[curproc],curtext) = empty do
      processor[curproc],curtext := processor[curproc],curtext + 1;
    repeat
    processor[curproc],curtext := processor[curproc],curtext + 1
    until sentforms(processor[curproc],curtext) <> empty;
    if processor[curproc],curtext > processor[curproc],right
      then poach := true
      else begin
        parsestates(processor[curproc],curtext) := state
        poach := false;
      end
  end
end {pushstate};
procedure popstate(cuproc, integer; var state, integer; var poach, boolean);
    var success, boolean;
{
    'initstack' has been called.
    'success' is true and 'poach' is false then 'state' has
    been removed from the state stack, and
    'processor(cuproc),curtext' has been moved back; 'state'
    is the state uncovered from the stack. A new current state
    and current symbol are defined by the updated
    'processor(cuproc),curtext'.
    If 'success' is false then an error message has been
    printed.
    If 'poach' is true then 'popstate' couldn't operate without
    encroaching on another processor's territory; no action has
    been taken.

    begin
    if processor(cuproc),curtext < 1
    then begin
        writeln('**** Parse stack underflow ****');
        success := false
    end
    else begin
        success := true;
        processor(cuproc),curtext := processor(cuproc),curtext - 1;
        while sentform(processor(cuproc),curtext) = empty do
        processor(cuproc),curtext := processor(cuproc),curtext - 1;
        if processor(cuproc),curtext < processor(cuproc),left
        then poach := true
        else begin
            poach := false;
            state := parsestates(processor(cuproc),curtext);
            parsestates(processor(cuproc),curtext) := empty;
            sentform(processor(cuproc),curtext) := empty
        end
        poach := true;
end
end (popstate);
procedure stackinsert(curproc:integer; state:integer; symbol:integer);
{
    Pre:  'poopstack' has cleared a stack location at 'curtext'.
          There may be other clear states to the right of the
current state.
    Post: 'state' and 'symbol' are filled in as the current state
          and current symbol; they are filled in in the rightmost
          empty slot before the next symbol. 'curtext' is updated
          to reflect the location of the new symbol.
}
begin
  while sentfor(processor[curproc],curtext) = empty do
    processor[curproc],curtext := processor[curproc],curtext + 1;
    processor[curproc],curtext := processor[curproc],curtext - 1;
    parsestates[processor[curproc],curtext] := state;
    sentfor(processor[curproc],curtext) := symbol
end {stackinsert};
procedure applyprod(curproc:integer; p:prodnt; var success:boolean);
{
  'curproc' identifies a processor for which a production
  'p' can be performed without danger of poaching on its
  left-hand neighbor's territory.

  'success' is false then the application of the
  production rule failed; otherwise, the nonterminal from
  the left hand side of the production rule 'p' has
  replaced the symbols on the stack that correspond to
  the symbols in the right hand side of 'p'.
}

label 9999;

var poach : boolean;
s : integer;
rhs : prodnt;

begin
  rhs := p^.rhs;
  while rhs <> nil do
    popstate(curproc,s,poach,succ);
    if not success then goto 9999;
    rhs := rhs^.next
  end;
  stackinsert(curproc,s,p^.lhs);
  goto 9999;
end {applyprod};
procedure writetrace(curproc:integer; act:parseaction; newstate:integer; prodnum:integer);
{
// Print intermediate results of parse.
}

begin
write(curproc,processor[curproc].left,7,processor[curproc].right,7,
processor[curproc].curtext,7,' ');
writesymbol(symbol[curproc]);
write(' ',symbol[curproc],curstate[curproc]);
case act of
  pass: write(' read ');
  apply: write(' reduce ');
  stutter: write(' stutter ');
  error: write(' error ')
end (case);
write(newstate,prodnum,7,' ');
writein
end (writetrace);
procedure writereduction(curproc:integer);
{
   write the top item of the parse stack and its context.
}

var lowstack : integer;
highstack : integer;
s : integer;

begin
   write(curproc," ");
   lowstack := processor(curproc),curtext;

   for s := 1 to 8 do
      begin
         while (sentform(lowstack) = empty) and (lowstack > 1) do
            lowstack := lowstack - 1;
         if lowstack > 1 then lowstack := lowstack - 1
      end;
   highstack := processor(curproc),curtext;

   for s := 1 to 7 do
      begin
         while (sentform(highstack) = empty) and (highstack <= lastchar) do
            highstack := highstack + 1;
         if highstack <= lastchar then highstack := highstack + 1
      end;
   for s := lowstack to processor(curproc),curtext = 1 do
      begin
         if sentform(s) <> empty then writeunifformat(sentform(s));
      end;
   write(" ");
   s := processor(curproc),curtext;
   while sentform(s) = empty do s := s + 1;
   writeunifformat(sentform(s));

   for s := processor(curproc),curtext + 1 to highstack do
      begin
         if sentform(s) <> empty then writeunifformat(sentform(s));
      end;
   writeln
end {writetrice};
procedure parse(var success:boolean);
{
    This procedure uses the finite state machine produced by
    applying the LRk algorithm to the input grammar.
}
label 9999;

var newstate : integer;
act : parseaction;
prodnum : integer;
p : prodtnt;
curproc : integer;
poach : boolean;

begin
    repeat
        begin
            scheckproc(curproc);
            lookastm(curstate(curproc),cursymbol(curproc),act,newstate,prodnum,succ);
            if not success then goto 9999;
            if vtrace then writetrace(curproc,act,newstate,prodnum);
            case act of
            pass:
                begin
                    pushstate(curproc,newstate,poach,succ);
                    if not success then goto 9999;
                    if poach then killproc(curproc)
                end;
            apply:
                begin
                    lookupprod(prodnum,p);
                    if reducewithoutpoach(curproc,p,rhs) then
                        begin
                            applyprod(curproc,p,succ);
                            if not success then goto 9999;
                            if vreduce then writereduction(curproc)
                        end
                    else
                        stutterproc(curproc)
                    end;
            stutter:
            stutterproc(curproc);
            error:
                succ := false
        end (case);
    until not success or (cursymbol(curproc) = ssid);
end {parse};
readgrammar(startpid,grammarok);
{Read the grammar rules}

if vorod then writeproductions;
{Write out productions}

if not grammarok
then
begin
writeln;
writeln("***** Grammar is no good; parsing aborted, *****");
writeln
end
else
begin
getheads(headsok);
if vhead then writeheads;
if not headsok
then
begin
writeln;
writeln("***** Head matrix no good; parsing aborted, *****");
writeln
end
else
begin
initism;
{Initialize fsm package}
initstates;
{Initialize state package}
{The special production rule is made into the first configuration, and
the first configuration is made into the first state.}
startlaset := {cfid};
{"Chicken foot" is look ahead symbol}
createconfig(startpid,1,startlaset,sconfig);{Create first configuration}
createstate(sconfig,stateroot);         {Create new state}
setcomplete(stateroot);                 {Complete first state}
{  
  ************ Start the lr(1) process in motion. ************
  
  s := stateroot;
  sucok := true;
  while (s <> nil) and sucok do
    begin
      successor(stateroot, s, sucok);
      s := s^.next
    end;
  
  if not sucok
    then
      begin
        writeln;
        writeln("*********** Could not compute successors; parsing aborted. ***********");
        writeln;
      end
    else
      begin
        Create the midstate and compute all the successors to it.
        
        createMidstate(midstate);
          {Create midstate configurations}

        s := stateroot;
        while s^.next <> nil do s := s^.next;
        s^.next := midstate;
          {Point at first state}
          {Attach midstate to list}

        s := midstate;
        midok := true;
        while (s <> nil) and midok do
          begin
            successor(stateroot, s, midok);
            s := s^.next
          end;

        midstatenum := midstate^.snum;
          {Record midstate's state number}
        checkfsmtable(midstatenum, lrlok);
          {Display states generated}
        if vstate then writestates(stateroot);
          {Display states generated}
        if vstate then renumberfsmtable;
          {Renumber to fix up after deletions}
        build fsm keys;
          {Build keys to permit fast lookup}
        if vfsm then writefsm;
          {Display fsm}
if not idok then
    begin
        writeln;
        writeln("********** Midstate could not be computed; parsing aborted, **********");
        writeln
        end
else
    if not lrlok then
        begin
            writeln;
            writeln("********** Grammar is not LR(1); parsing aborted, **********");
        end
else
    begin
        readsymbols;
        {Read input text}
        initstack;
        {Initialize parse stacks}
        assignproc;
        {Assign text to processors}
        if vtrace or vreduce then
            begin
                page(output);
                writeln("Trace of Parse");
            writeln
            end;
            if vtrace then
                begin
                    writeln(" Curnt Left Right Curnt Input Symbol Curnt New ",
                    " Prod ");
                    writeln(" Proc Bound Bound Positn Symbol Id State Action State ",
                    " Num ");
                end;
            if vtrace or vreduce then
                begin
                    writeln("----------------------------------------------------------");
                    writeln("----------------------------------------------------------");
                writeln
                end;
parse(parseok);

if parseok
then
    begin
        writeln;
        writeln("********** Parse succeeded **********");
        writeln end
else
    begin
        writeln;
        writeln("********** Parse failed **********");
        writeln end
end end;
end (paralr).