1 LR(0) Definitions

rule A production from the grammar. Example:

\[
stmt \rightarrow \text{if expr then stmt}
\]

item A rule with a ‘dot’ somewhere in it. Example:

\[
stmt \rightarrow \text{if . expr then stmt}
\]

complete item An item with a dot at the end of the rule. Example:

\[
stmt \rightarrow \text{if expr then stmt .}
\]

augmentation The process of adding an item to a set of items in which a dot occurs before an occurrence of a nonterminal. Items are added for all rules for that nonterminal with the dot at the start. Example: the set

\[
stmt \rightarrow \text{if . expr then stmt}
\]

\[
stmt \rightarrow \text{if . expr then stmt else stmt}
\]

includes dots before expr and is augmented by adding the following items:

\[
expr \rightarrow \text{. expr + expr}
\]

\[
expr \rightarrow \text{. expr * expr}
\]

\[
expr \rightarrow \text{. ( expr )}
\]

\[
expr \rightarrow \text{. ID}
\]

\[
expr \rightarrow \text{. INT_CONST}
\]

state A closed set of items, that is, a set that cannot be further augmented. Example:

\[
program \rightarrow \text{. stmt_list}
\]

\[
stmt_list \rightarrow \text{. stmt_list stmt}
\]

\[
stmt_list \rightarrow \text{. stmt}
\]

\[
stmt \rightarrow \text{. ID = expr ;}
\]

\[
stmt \rightarrow \text{. if expr then stmt}
\]

\[
stmt \rightarrow \text{. if expr then stmt else stmt}
\]

partial tree The result of reducing a portion of the yield to a single symbol. The symbol may be a terminal, in which case, the partial tree represents just the terminal’s value, or it may be a nonterminal, in which case the partial tree represents one or more reductions. The parser never examines the contents of the partial tree. For PA3, the partial tree will either be ignored (such as for operator symbols) or will be an AST (abstract syntax tree). Example:

\[
stmt
\]

\[
| ID = expr
\]

\[
| INT_CONST
\]

parse stack A stack of alternating states and partial trees. The deepest element will be a state.
2 LR Parsing

We add a special production to the grammar:

\[ SS \to \text{start\_symbol} \quad $$ \]

where $$ represents the end-of-file condition.

The following algorithm assumes we are constructing states on the fly. If the states are created ahead of time, some of the steps can be simplified as described in Section 3.

1. Construct the state that includes the item

\[ SS \to \quad . \quad \text{start\_symbol} \quad $$ \]

by augmenting the set until closed. Push this state on the stack. Continue with step 2.

2. Non-deterministically choose an action:
   
   - Take next terminal from input and place on stack (shift). Go to step 3.
   - If the top state on the stack contains a complete item, (reduce) go to step 4.

3. Look at all items in the state immediately beneath the top partial tree. Construct the set of all items that can be constructed by moving the dot over exactly the symbol of the partial tree. If the set is empty, stop the parse with REJECT. Close the set and push the closed set (a state) on the stack. Go to step 2.

4. If rule for complete item is the specially added rule, stop the parse with ACCEPT. Otherwise, let \( n \) be the number of symbols on the right-hand side of the rule from the stack and pop \( 2n \) elements from the stack: \( n \) states and \( n \) partial trees. Discard the popped states, but use the popped partial trees to construct a new partial tree with symbol from the left-hand-side of the rule. Push new partial tree on stack. Go to step 3.

The non-determinism of step 2 can lead to parsing conflicts: shift/reduce conflicts when there is a dot before a terminal in the current state as well as a complete item, and reduce/reduce conflicts when there are more than one complete item. LR(0) has no way to avoid such conflicts.

The “perfect” LR parser nondeterministically chooses the action that enables an ACCEPT. Realistic parsers such as SLR(1), LALR(1) and LR(1) use one token of lookahead to determine for each complete item whether the reduction will lead to a parse state that can shift the given token.

3 Parse Tables

If the states are constructed ahead of time in parse tables, then step 1 consists of pushing state 0 on the stack and step 3 consists of using the current state and symbol to look up the new state from the table. A parse table will also determine the action for step 2 by using lookahead or arbitrarily choosing one action over others. For example, bison-generated parsers choose shift actions over reduce actions and an earlier rule over a later rule. This breaking of conflicts is undesirable.

4 Example Output from Bison
State 19 conflicts: 1 shift/reduce

Grammar

0 $accept: program $end
1 program: stmt_list
2 stmt_list: stmt_list stmt
3 | stmt
4 stmt: ID '=' expr ';' | IF expr THEN stmt | IF expr THEN stmt ELSE stmt
5 | expr '+' expr
6 | expr '*' expr
7 | '(' expr ')' | ID
10 | INT_CONST

state 0

0 $accept: . program $end
IF shift, and go to state 1
ID shift, and go to state 2
program go to state 3
stmt_list go to state 4
stmt go to state 5

state 1

5 stmt: IF . expr THEN stmt
6 | IF . expr THEN stmt ELSE stmt
ID shift, and go to state 6
INT_CONST shift, and go to state 7
'(' shift, and go to state 8
expr go to state 9

state 2

4 stmt: ID . '=' expr ';';
'=' shift, and go to state 10

state 3

0 $accept: program . $end
$end shift, and go to state 11

state 4

1 program: stmt_list . [$end]
2 stmt_list: stmt_list . stmt
IF shift, and go to state 1
ID shift, and go to state 2

$default reduce using rule 1 (program)
stmt go to state 12

state 5
3 stmt_list: stmt .
$default reduce using rule 3 (stmt_list)

state 6
10 expr: ID .
$default reduce using rule 10 (expr)

state 7
11 expr: INT_CONST .
$default reduce using rule 11 (expr)

state 8
9 expr: '(' . expr ')'
ID shift, and go to state 6
INT_CONST shift, and go to state 7
'(' shift, and go to state 8
expr go to state 13

state 9
5 stmt: IF expr . THEN stmt
6 | IF expr . THEN stmt ELSE stmt
7 expr: expr . '+' expr
8 | expr . '*' expr
THEN shift, and go to state 14
'+' shift, and go to state 15
'*' shift, and go to state 16

state 10
4 stmt: ID '=' . expr ';';
ID shift, and go to state 6
INT_CONST shift, and go to state 7
'(' shift, and go to state 8
expr go to state 17

state 11
0 $accept: program $end .
$default accept
state 12
2 stmt_list: stmt_list stmt .
$default reduce using rule 2 (stmt_list)

state 13
7 expr: expr . '+' expr
8 | expr . '*' expr
9 | '(' expr . ')

'+' shift, and go to state 15
'*' shift, and go to state 16
')' shift, and go to state 18

state 14
5 stmt: IF expr THEN . stmt
6 | IF expr THEN . stmt ELSE stmt

IF shift, and go to state 1
ID shift, and go to state 2
stmt go to state 19

state 15
7 expr: expr .'+' expr
ID shift, and go to state 6
INT_CONST shift, and go to state 7
('(' shift, and go to state 8
expr go to state 20

state 16
8 expr: expr . '*' expr
ID shift, and go to state 6
INT_CONST shift, and go to state 7
('(' shift, and go to state 8
expr go to state 21

state 17
4 stmt: ID '=' expr . ';' .
$default reduce using rule 4 (stmt)

state 18
9 expr: '(' expr ')'.
$default reduce using rule 9 (expr)

state 19
5 stmt: IF expr THEN stmt . [END, IF, ELSE, ID]
6 | IF expr THEN stmt . ELSE stmt

ELSE shift, and go to state 23
ELSE [reduce using rule 5 (stmt)]
$default reduce using rule 5 (stmt)

state 20
7 expr: expr .'+' expr [THEN, '+' , ';' , ')']
8 | expr . '*' expr [THEN, '+' , ';' , ')']

'+' shift, and go to state 15
'*' shift, and go to state 16
expr go to state 21

state 21
7 expr: expr . ' + ' expr [THEN, ' + ' , '+' , ';' , ')']
8 | expr . ' + ' expr [THEN, ' + ' , '+' , ';' , ')']

$default reduce using rule 8 (expr)

state 22
4 stmt: ID '=' expr . ';' .
$default reduce using rule 4 (stmt)

state 23
6 stmt: IF expr THEN stmt ELSE . stmt

IF shift, and go to state 1
ID shift, and go to state 2
stmt go to state 24

state 24
6 stmt: IF expr THEN stmt ELSE stmt .
$default reduce using rule 6 (stmt)