CS202 – System Software

Dr. Manish Khare

Shift-Reduce Parser

- There are two main categories of shift-reduce parsers

1. Operator-Precedence Parser

- simple, but only a small class of grammars.

2. LR-Parsers

- covers wide range of grammars.
  - SLR – simple LR parser
  - Canonical LR – most general LR parser
  - LALR – intermediate LR parser (lookhead LR parser)
- SLR, Canonical LR and LALR work same, only their parsing tables are different.

Operator-Precedence Parser

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.

- Ex:

$E \rightarrow AB$	$E \rightarrow EOE$	$E \rightarrow E+E \mid$
$A \rightarrow a$	$E \rightarrow id$	$E*E \mid$
$B \rightarrow b$	$O \rightarrow +\mid *\mid /$	$E/E \mid id$
not operator grammar	not operator grammar	operator grammar

Operator Precedence Parser

- A parser that reads and understand an operator precedence grammar is called as **Operator Precedence Parser**.

- **Designing Operator Precedence Parser**

- In operator precedence parsing,
  - Firstly, we define precedence relations between every pair of terminal symbols.
  - Secondly, we construct an operator precedence table.

Defining Precedence Relations

- The precedence relations are defined using the following rules-

- **Rule-01:**
  - If precedence of b is higher than precedence of a, then we define a < b
  - If precedence of b is same as precedence of a, then we define a = b
  - If precedence of b is lower than precedence of a, then we define a > b

Rule-02:

- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence.

Rule-03:

- If two operators have the same precedence, then we go by checking their associativity.

Parsing A Given String

- The given input string is parsed using the following steps-

- **Step-01:**
- Insert the following-
  - $ symbol at the beginning and ending of the input string.
  - Precedence operator between every two symbols of the string by referring the operator precedence table.

- **Step-02:**
  - Start scanning the string from LHS in the forward direction until > symbol is encountered.
  - Keep a pointer on that location.

Step-03:

- Start scanning the string from RHS in the backward direction until < symbol is encountered.
- Keep a pointer on that location.

Step-04:

- Everything that lies in the middle of < and > forms the handle.
- Replace the handle with the head of the respective production.

Step-05:

- Keep repeating the cycle from Step-02 to Step-04 until the start symbol is reached.

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

$a \lessdot b$	b has higher precedence than a
$a \doteq b$	b has same precedence as a
$a \gtrdot b$	b has lower precedence than a

- These relations may appear similar to the ‘less than’, ‘equal to’ and ‘greater than’ operator.

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators.

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator O_1 has higher precedence than operator O_2,
   ➔ $O_1 \cdot> O_2$ and $O_2 <\cdot O_1$

2. If operator O_1 and operator O_2 have equal precedence,
   they are left-associative ➔ $O_1 \cdot> O_2$ and $O_2 \cdot> O_1$
   they are right-associative ➔ $O_1 <\cdot O_2$ and $O_2 <\cdot O_1$

3. For all operators O,
   $O <\cdot id$, $id \cdot> O$, $O <\cdot ($, $(<\cdot O$, $O \cdot>)$, $) \cdot> O$, $O \cdot> \$$, and $\$ <\cdot O$

4. Also, let

$(=\cdot)$	$\$ <\cdot ($	$id \cdot>)$	$) \cdot> \$$
$(<\cdot ($	$\$ <\cdot id$	$id \cdot> \$$	$) \cdot>)$
$(<\cdot id$			

	+	-	*	/	^	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
^	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=·	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Handling Unary Operators

- Operator-Precedence parsing cannot handle the unary minus when we also the binary minus in our grammar.
- The best approach to solve this problem, let the lexical analyzer handle this problem.
  - The lexical analyzer will return two different operators for the unary minus and the binary minus.
  - The lexical analyzer will need a lookahead to distinguish the binary minus from the unary minus.
- Then, we make

$O \lessdot$ unary-minus	for any operator
unary-minus $\gtrdot O$	if unary-minus has higher precedence than O
unary-minus $\lessdot O$	if unary-minus has lower (or equal) precedence than O

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b.

$f(a) < g(b)$ whenever $a \lessdot b$

$f(a) = g(b)$ whenever $a \doteq b$

$f(a) > g(b)$ whenever $a \gtrdot b$

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,
  - $\lessdot$ with marking the left end,
  - $\doteq$ appearing in the interior of the handle, and
  - $\gtrdot$ marking the right hand.

- To be more precise, suppose we have a right-sentential form of an operator grammar.
- The fact no adjacent non-terminals appear on the right sides of productions implies that no right-sentential form will have two adjacent non-terminals either.
- Thus, we may write the right-sentential form as $\beta_0 a_1 \beta_1 a_2 \ldots a_n \beta_1$, where β_i each is either ε (the empty string) or a single non-terminal, and each a_i is a single terminal.

Using Operator-Precedence Relations

- Suppose that between a_i and a_{i+1} exactly one of the relations $\lessdot$, $\doteq$, and $\gtrdot$ holds. Further let us use \$ to mark each end of the string, and define \$ $\lessdot$ B and b $\gtrdot$ \$.
- Now suppose we remove the non-terminals from the string and place the correct relation $\lessdot$, $\doteq$, and $\gtrdot$, between each pair of terminals and between endmost terminals and the \$'s marking the ends of the strings

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\hat{}E \mid (E) \mid -E \mid id$

The partial operator-precedence table for this grammar

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Then the input string 'id+id*id' with the precedence relations inserted will be:

\$ $\lessdot$ id $\gtrdot$ + $\lessdot$ id $\gtrdot$ * $\lessdot$ id $\gtrdot$ \$

To Find The Handles

1. Scan the string from left end until the first $\cdot>$ is encountered.
2. Then scan backwards (to the left) over any $=\cdot$ until a $<\cdot$ is encountered.
3. The handle contains everything to left of the first $\cdot>$ and to the right of the $<\cdot$ is encountered.

$ <· id ·> + <· id ·> * <· id ·> $	E → id	$ id + id * id $
$ <· + <· id ·> * <· id ·> $	E → id	$ E + id * id $
$ <· + <· * <· id ·> $	E → id	$ E + E * id $
$ <· + <· * ·> $	E → E*E	$ E + E * E $
$ <· + ·> $	E → E+E	$ E + E $
$ $		$ E $

Operator-Precedence Parsing Algorithm

- The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
   if ( $ is on top of the stack and p points to $ ) then return
   else {
      let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
      if ( a <· b or a =· b ) then {        /* SHIFT */
         push b onto the stack;
         advance p to the next input symbol;
      }
      else if ( a ·> b ) then               /* REDUCE */
         repeat pop stack
         until ( the top of stack terminal is related by <· to the terminal most recently popped );
      else error();
   }
```

Operator-Precedence Parsing Algorithm - Example

$E \rightarrow E+E \mid E*E \mid id$

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

stack	***input***	***action***		
$	id+id*id$	$ $<\cdot$ id	shift	
$id	+id*id$	id $\cdot>$ +	reduce	$E \rightarrow id$
$	+id*id$	$ $<\cdot$ +	shift	
$+	id*id$	$ $<\cdot$ id	shift	
$+id	*id$	id $\cdot>$ *	reduce	$E \rightarrow id$
$+	*id$	+ $<\cdot$ *	shift	
$+*	id$	* $<\cdot$ id	shift	
$+*id	$	id $\cdot>$ $	reduce	$E \rightarrow id$
$+*	$	* $\cdot>$ $	reduce	$E \rightarrow E*E$
$+	$	+ $\cdot>$ $	reduce	$E \rightarrow E+E$
$	$	accept		

Exercise

- Consider the following grammar-
  - $S \rightarrow (L) \mid a$
  - $L \rightarrow L , S \mid S$
- Construct the operator precedence parser and parse the string (a , (a , a)).

- The terminal symbols in the grammar are { (,) , a , , }
- We construct the operator precedence table as-

	a	(	)	,	$
a		>	>	>	>
(	<	>	>	>	>
)	<	>	>	>	>
,	<	<	>	>	>
$	<	<	<	<	

Operator Precedence Table

Parsing Given String-

- Given string to be parsed is **(a , (a , a))**.
- We follow the following steps to parse the given string-

Step-01:

- We insert $ symbol at both ends of the string as-

$ (a , (a , a)) $

- We insert precedence operators between the string symbols as-

$ < (< a > , < (< a > , < a >) >) > $

Step-02:

We scan and parse the string as-

$ < (**< a >** , < (< a > , < a >) >) > $

$ < (S , < (**< a >** , < a >) >) > $

$ < (S , < (S , **< a >**) >) > $

$ < (S , **< (S , S) >**) > $

$ < (S , **< (L , S) >**) > $

$ < (S , **< (L) >**) > $

$ **< (S , S) >** $

$ **< (L , S) >** $

$ **< (L) >** $

$ **< S >** $

$ $

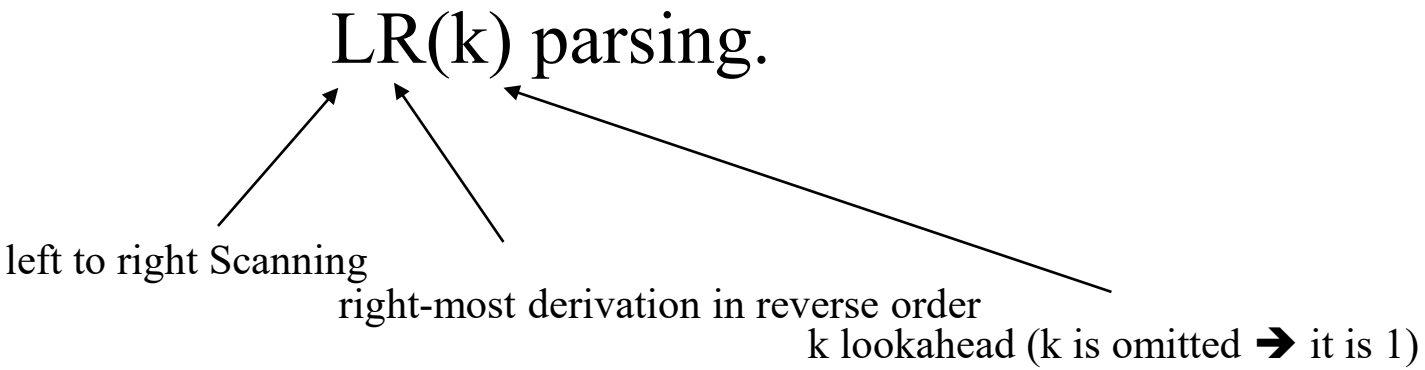
Disadvantages of Operator Precedence Parsing

- **Disadvantages**:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages

LR Parser

- The most powerful shift-reduce parsing (yet efficient) is:



- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    $$\text{LL(1)-Grammars} \subset \text{LR(1)-Grammars}$$

  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parser

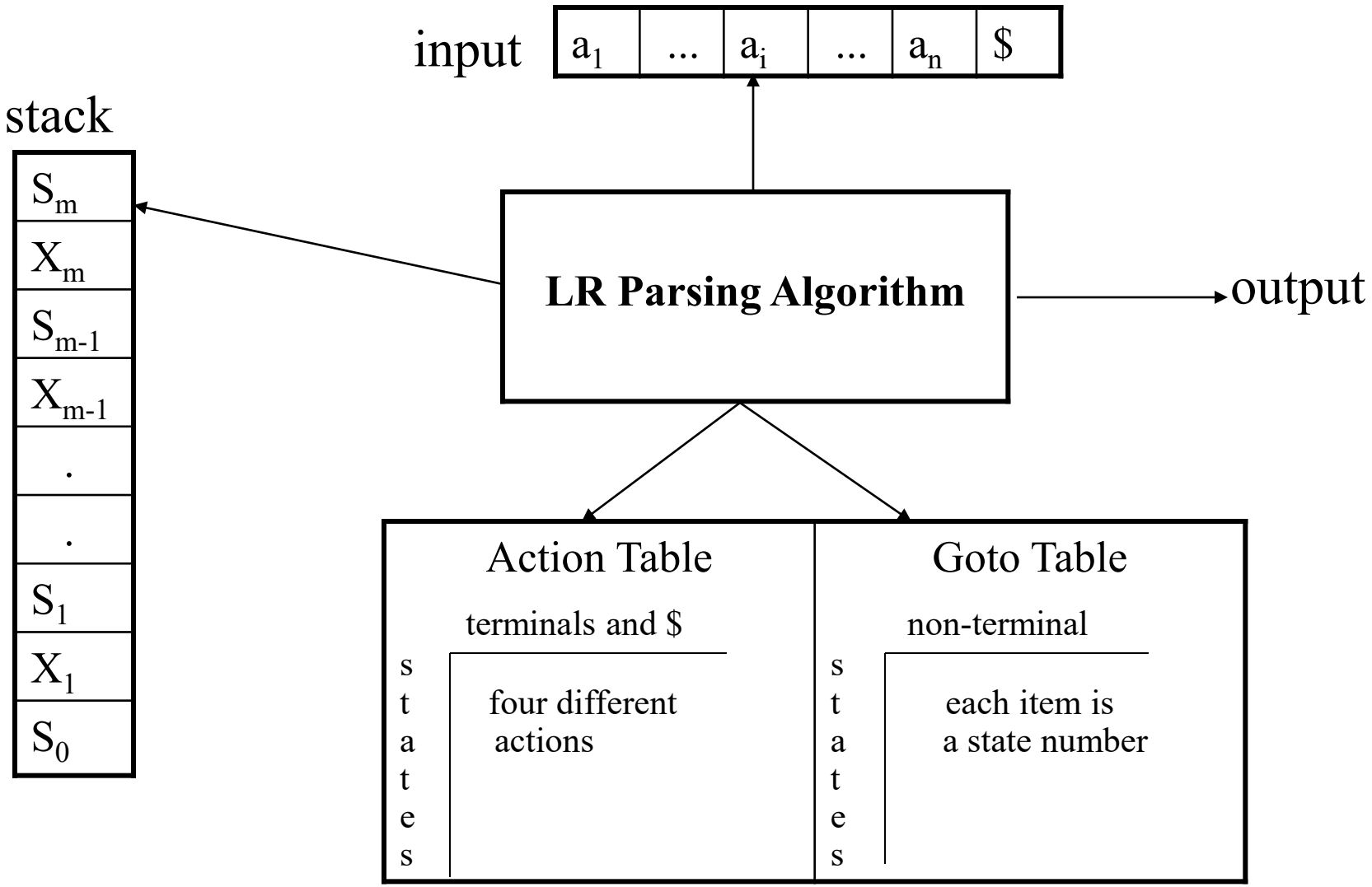
- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

- Disadvantage
  - This technique have too much work is needed to construct an LR parser by hand for a typical programming language.

Techniques for LR Parser

- Three techniques for LR Parser
  - The first method, called Simple LR (SLR), is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammar on which the other method succeed.
  - The second method, called Canonical LR (CLR), is most powerful and work on very large class of grammars. It is very expensive and difficult to implement.
  - The third method, called Look Ahead LR (LALR), is intermediate in power between SLR and CLR. It works on most of the class of grammar and with come efforts it can be implemented easily.

Structure of LR Parser



Structure of LR Parser

- LR Parser consists
  - An input
  - An output
  - A stack
  - A driving program
  - A parsing table
    - Action
    - Goto

Structure of LR Parser

- A stack is used by a LR Parser to store a string of the form $S_0X_1S_1X_2\ldots\ldots S_mX_m$, where X_i is the grammar symbol, S_i is a state symbol and S_m is on TOP.
- Parsing table consists of two parts
  - Action: is a table of size *n x m* where *n* is the total number of states of the parser and *m* is the number of terminal symbol in *T(G)* including $.
  - Goto: is a table of size *n x p* where *p* is number of non-terminal symbols in N(G). Function Goto takes a state and grammar symbol as arguments and produces a state.

Structure of LR Parser

- The program driving the LR parser behaves as follows:
- It determines S_m, the state currently on the top of the stack, and a_i the current input symbol. It then consults ACTION $[S_m, a_i]$ the parsing action table entry for state Sm and input ai.
- The entry ACTION $[S_m, a_i]$ can have one of the four possible entries:
  - Si, which means shift to state i
  - Rj, which means reduce by using the jth rule.
  - acc, which means accept
  - e, which means error

Configuration of LR Parser

- A configuration of a LR parsing is:

$$(S_o X_1 S_1 \ldots X_m S_m, \; a_i a_{i+1} \ldots a_n \$)$$

Stack — Rest of Input

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)
- A configuration of a LR parsing represents the right sentential form:

$$X_1 \ldots X_m a_i a_{i+1} \ldots a_n \$$$

Configuration of LR Parser

1. IF ACTION $[S_m, a_i]$ = Shift s

the parser executes a shift move, entering the configuration

$$(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \rightarrow (S_o X_1 S_1 \ldots X_m S_m \mathbf{a_i s}, a_{i+1} \ldots a_n \$)$$

Here the parser has shift both the current input symbol a_i and the next state S, which is given in ACTION $[S_m, a_i]$ onto stack, and a_{i+1} becomes the current input symbol.

2. IF ACTION $[S_m, a_i]$ = reduce $A \rightarrow \beta$

then the parser executes a reduce move, entering the configuration

$$(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \rightarrow (S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A s, a_i \ldots a_n \$)$$

Where **s=goto[s_{m-r}, A] and r is the length of** β, the right side of the production

Configuration of LR Parser

3. IF ACTION $[S_m, a_i]$ = **Accept**

Parsing successfully completed

4. IF ACTION $[S_m, a_i]$ = **Error**

The parser has discovered an error and calls error revcovery routine.

Construction of SLR or LR(0) Parser

- Step 1:
  - Constitute the augmented grammar G' by adding one production $S' \rightarrow S$, where S is the start symbol of grammar. This is done in order to stop parsing and announce acceptance.

- Step 2:
  - Construct canonical collection of LR(0) items where LR(0) items of the given grammar G is the production of G having ('.') at some position on RHS of production. LR(0) items are also called **canonical collection of items**.
  - Initially LR(0) set of items are found as I_0=closure ($E' \rightarrow .E$), where closure (I) will be defined by algorithm and I_0 is start state of LR(0) parser.

Construction of SLR or LR(0) Parser

- Step 3:
  - Find the state transition using GOTO, where GOTO will be defined by algorithm
  - I_1=GOTO (I_0,.X), where X is some terminal or non-terminal.

- Step 4:
  - Using Closure (I) and GOTO (I,X) construct new states and hence new LR(0) set of items for each state.

- Step 5:
  - Continue Step 4 till all the state transition have been calculated and there is no more transition on input.

Construction of SLR or LR(0) Parser

- Step 6:
  - Construct parse table by applying SLR Table construction algorithm.

- Step 7:
  - Apply shift reduce action for verifying the acceptance of input string or error.

Closure (I) Operation

- If I is the set of items of the grammar G, then the items closure (I) is constructed from I, by using following rules.
  1. Every item in I is added in closure(I).
  2. If there is production of the form $A \rightarrow \alpha.B\beta$, where B is a non-terminal such that if there is also a production of the form $B \rightarrow .\gamma$ in G, if it is not there in I.
  3. Again evaluate closure (I) to added items.

GOTO Operation

- GOTO(I,X), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items ($A \rightarrow \alpha x.\beta$) such that $A \rightarrow \alpha.x\beta$ is in I.

- If I is the set of items that are valid for some viable prefix γ, then GOTO(I,X) is the set of items that are valid for the viable prefix of X.

Canonical Collection of Set of LR(0)

- We are now ready to give the algorithm to construct C, the canonical collection of set of LR(0) items for an augmented grammar G'.

Procedure items(G')

begin

C={closure({$S' \rightarrow .S$})}

repeat

for each set of items I in C an each grammar symbol X such that GOTO(I,X) is not empty and not in C do

add GOTO(I,X) in C

until no more set of items can be added to C

end

Algorithm for Construction of SLR Parsing Table

- Now we shall discuss, how to construct the SLR parsing action and GOTO function.
- For a given grammar G, we augmented G to produce G/ and from G/, we construct C (the canonical collection of set of items for G/).
- We construct ACTION (the parsing action function), and GOTO (the GOTO function) from C.
- It requires us to know FOLLOW(A), for each non-terminal A of the grammar.

1. Construct C={I_0,I_1,…..I_n}, the collection of set of LR(0) items for G/.
2. In SLR table states in C constitutes the row, just name them from 0 to n, divide the columns into ACTION and GOTO, such that ACTION contains all terminal symbols of G and $, and GOTO contains all the non-terminals of G.
3. The parsing action for state j, are determined as follows

ACTION Section Parsing rule

1. If GOTO (I_i,a) = I_j, then set ACTION [i,a] = $shift_j$ (S_j).
2. If [A→α.] is in I_i, then set ACTION [i,a] =$reduce_k$ (r_k), where r means reduce and k is the production number of production A→α in G/ starting from zero in augmented grammar. This shows the reduce action. This entry ACTION [i,a] =r_k, is made for all a in FOLLOW(A) for production A→α, here A may not be S/.
3. If [S/ → S.] in Ii, then set ACTION [i,$] to accept.

GOTO Section Parsing rule

The GOTO transition for state i are determined as follows

1. If GOTO (I_i,A) = I_j, then set GOTO [i,A] = j.

   which means, corresponding to state I and non-terminal A, entry j, i.e. state number of the state to which transition is made into the GOTO part of LR(1) parse table.

2. All entries not defined by above rules are made "error"

3. The initial state of the parser is to one constructed from the set of items containing [$S^{/} \rightarrow .S,\$$]

Exercise

- Consider the following grammar
  - E→E+T | T
  - T →T*F | F
  - F →(E) | id

Construct the SLR Parsing table.

CLR/LR(1)

- We seen that the SLR parser failed for some unambiguous grammar.
- So we define CLR/LR(1) parser with some extra information in the form of terminal symbol, as a second component in each item of state.
- The general form of item becomes [A→α.β,a], where A→αβ is a production and ‘a’ is a terminal symbol, or right end marker ‘$’.
- This parser also called LR(1), where 1 refers to the length od second component, called the lookahead of the item.

- The lookahead does not effect the item of the form [A→α.β,a], where β is not Ɛ, but an item of the form [A→α.,a] calls for reduction by A→α only on those input symbols a for which [A→α.,a] is an LR(1) item in the state on the top of stack.

- Now we redefine Closure (I) and GOTO (I,X) for finding LR(1) items, as below:

Computation of Closure (I)

Begin

repeat

for each item [A→α.Bβ,a] in I and each production B→ γ in G/ and each terminal b in FIRST(βa) if [B→ .γ,b] is not in I, then

add [B→ .γ,b] to I

until no more items can be added to I;

return I

end

Computation of GOTO (I,X)

Begin

Let J be the set of items [A→αX.β,a] such that [A→α.Xβ,a] is in I

return Closure (J)

end

Canonical Collection of Set of LR(1)

- We are now ready to give the algorithm to construct C, the canonical collection of set of LR(0) items for an augmented grammar G'.

Procedure items(G')

begin

C={closure({$S' \rightarrow .S,\$$})}

repeat

for each set of items I in C an each grammar symbol X such that GOTO(I,X) is not empty and not in C do

add GOTO(I,X) in C

until no more set of items can be added to C

end

Algorithm for Construction of Canonical LR Parsing Table

1. Construct C={I_0,I_1,…..I_n}, the collection of set of LR(1) items for G'.
2. In LR(1) table states in C constitutes the row, just name them from 0 to n, divide the columns into ACTION and GOTO, such that ACTION contains all terminal symbols of G and $, and GOTO contains all the non-terminals of G.
3. The parsing action for state j, are determined as follows

ACTION Section Parsing rule

1. If [A→α.aβ,b] is in I, and GOTO (I_i,a) = I_j then set ACTION [i,a]=Shift j (S_j)

   Here a is required to be a terminal
2. If there is some production of the form [A→α.,a] in I_i and A≠S', then set ACTION [i,a]=r_j, for all a in FOLLOW(A); where r is for reduce action with production number j in G'.
3. If [S' → S., $] is in I_i then set ACTION [i, $] = 'Acc' means ACCEPT.

If any conflicts results from the above rules, then grammar is not LR(1).

GOTO Section Parsing rule

1. If GOTO (I_i,A) = I_j, then set GOTO [i,A] = j.
2. All entries not defined by above rules are made "error"
3. The initial state of the parser is to one constructed from the set of items containing [$S^{/} \rightarrow .S, \$$]

The parse table formed using above algorithm is called Canonical LR (LR(1)) parse table, and the LR parser using this table is called a canonical LR parser.

If parsing table has no multiple entries in same row and column, then the grammar is called LR(1) grammar.

Example

Construct the CLR parsing table for the following grammar.

$S \rightarrow CC$

$C \rightarrow aC$

$C \rightarrow d$

LALR Parser

- LALR parser is lookahead LR parser.
- In this parser we construct LALR items in the similar way, as we did in Canonical LR(1) parser.
- Here we look for the items having same core, i.e. set of first components, and they are merged together to form a single set of items, that means if some state have matching items, except for the second components then they are combined to form a single item and state.

- Let S_1 and S_2 are two states in a canonical LR grammar.

$S_1 \rightarrow \{C \rightarrow c.C, c/d; C \rightarrow .cC, c/d; C \rightarrow .d, c/d\}$

$S_2 \rightarrow \{C \rightarrow c.C, \$; C \rightarrow .cC, \$; C \rightarrow .d, \$\}$

- Then in LALR these two states are merged together and form new state S_{12}

$S_{12} \rightarrow \{C \rightarrow c.C, c/d/\$; C \rightarrow .cC, c/d/\$; C \rightarrow .d, c/d/\$\}$

Construction of LALR parser

- Construct the set of LR(1) items.
- Merge the set with common core together as one set, if no conflict (shift-shift or shift-reduce).
- If a conflict arises, it implies that the grammar is not LALR.
- The parsing table is constructed from the collection of merged sets of item using the same algorithm for LR(1) parsing.

Algorithm for Construction of LALR Parsing Table

1. Construct $C=\{I_0,I_1,.....I_n\}$, the collection of set of LR(1) items for G'.

2. For each core present in among these sets, find all sets having the core, and replace these sets by their union.

3. Parsing table is constructed as for canonical LR.

4. The GOTO table is constructed by taking the union of all sets of items having the same core. If J is the union of one or more sets of LR(1) items, that is $J=I_1 \ U \ I_2 \ U \ I_3 \ ..... \ U \ I_K$, then the cores of $GOTO(I_1,X)$, $GOTO(I_2,X)$, …… $GOTO(I_K,X)$ are the same as all of them have same core. Let K be the union of all sets of items having same core as $GOTO(I_1,X)$.

   Then $GOTO(J,X) = K$

Example

Construct the LALR parsing table for the following grammar.

$S \rightarrow CC$

$C \rightarrow aC$

$C \rightarrow d$