COMPILER CONSTRUCTION (LR(1) parser) **Fall 2019**

Lecture 20: Sept. 4-6,23 2019

Instructor: K.R. Chowdhary *: Professor of CS*

20.1 LR(1)-Parsing

In the LR(0) parser, the ACTION was either shift or reduce. In LR(1) there is additional function GOTO. The Fig. 20.1 shows the schematic of LR(1) parser. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION table and GOTO table). The driver program is same for all LR parsers, and only the parsing table changes from one parser to another parser. The parsing programs reads characters from input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR(0) and LR(1) parser shifts a state. Each state summarizes the information contained in the stack below it.

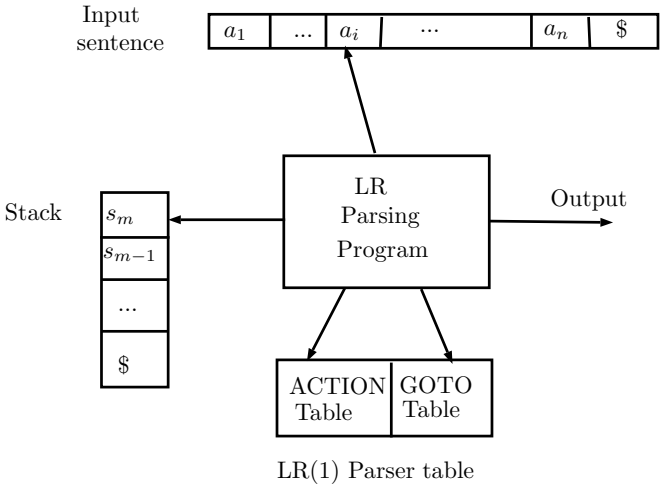

Figure 20.1: Model of an LR(1) Parser

The stack holds the states s_0, s_1, .., s_m, where s_m is at top. In the SLR method, the stack holds the states from the LR(0) automaton. The canonical LR (we will call LR(1) as LR only) and LALR (look-ahead LR) methods are similar. By construction, each state has a corresponding grammar symbol. Note that states corresponds to sets of items, and there is a transition from state i to state j if $GOTO(I_i, X) = I_j$. All transitions to state j must be for same grammar symbol X. Thus, each state, except the start state 0, has a unique grammar symbol associated with it. However, some times, there are more than one state for same symbol. For example, states I_5 and state I_9 are both associated with symbol F.

Similarly, for symbol T there are two states I_6 and I_8. These are the cases where there are two productions for one non-terminal symbol. There are also two states for symbol E (1 and 7).

Table 20.1: Expression Grammar

Rule No.	Rule
1.	$E \to E + T$
2.	$E \to T$
3.	$T \to T \times F$
4.	$T \to F$
5.	$F \to (E)$
6.	$F \to id$

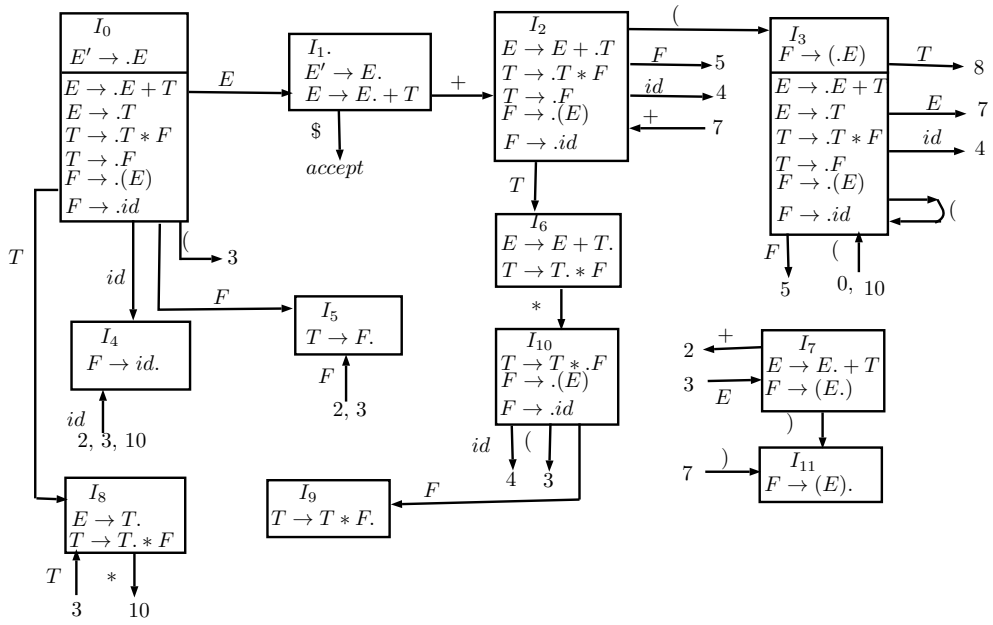

Figure 20.2: LR(0) automaton for expression grammar given in Table 20.1

Structure of LR(1)-Parsing table We will first consider SLR(1) where the S stands for simple. SLR(1) parsers use the same LR(0) configuration sets and have the same table structure and parser operation, so everything you have already learned about LR(0) applies to LR(1) parser also. The difference comes in assigning table actions, where we are going to use one token of lookahead to help arbitrate among the conflicts. If we think back to the kind of conflicts we encountered in LR(0) parsing, a state in an LR(0) parser can have at most one reduce action and cannot have both shift and reduce instructions.

Since a reduce is indicated for any completed item, this dictates that each completed item must be in a state by itself. But let's revisit the assumption that if the item is complete, the parser must choose to reduce. Is that always appropriate? If we peeked at the next upcoming token, it may tell us something that invalidates that reduction. If the sequence on top of the stack could be reduced to the nonterminal A , what tokens do we expect to find as the next input? What tokens would tell us that the reduction is not appropriate?

Perhaps $Follow(A)$ could be useful here!

The parsing table of LR(1) parser consists of two parts: a parsing-action function ACTION, and a goto function GOTO.

1. The ACTION function takes arguments a state s_i and a terminal a (or \$, the end marker). The value of $ACTION[s_i, a]$ can have one of four forms:
   (a) *Shift* s_j, where s_j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state s_j to represent a.
   (b) *Reduce* $A \to \beta$. The action of the parser effectively reduces β on the top of the stack to head A. The states which corresponds to Reduce ACTION have no further transitions, e.g., states 4, 5, 9, 11.
   (c) *Accept.* The parser accepts the input and finishes parsing.
   (d) *Error.* The parser discovers an error in the input, and takes some corrective action.
2. We extend the GOTO function, defined on sets of items, to states: if $GOTO[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

LR(1)-Parser configurations The parser configurations represents the the complete state of the parser. A state is: its stack and the remaining input. A *configuration* of an LR parser is a pair:

$$(s_0 s_1 ... s_m, a_i a_{i+1} ... a_n \$)$$

where first component is the stack content (top on the right), and the second component is the remaining input $a_i ... a_n$. This configuration represents the right sentential form:

$$X_1 X_2 ... X_m a_i a_{i+1} .. a_n$$

in the same way as shift-reduce parser; the only difference is that instead of grammar symbols, the stack holds states from which grammar symbols can be recovered. That is, X_i is the grammar symbol represented by state s_i. Note that s_0, the start state of the parser, does not represent a grammar symbol, and servers as a bottom-of-stack marker, as well as playing an important role in the parser.

Behaviour of LR(1) parser The next move of the parser from the configuration above is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and the consulting entry after each of the four steps of move are as follows:

1. If $ACTION[s_m, a_i]$ = shift s, the parser executes a shift move; it shifts the next state s onto the stack, and enters s into the configuration,

$$(s_0 s_1 ... s_m s, a_{i+1} ... a_n \$)$$

The symbol a_i need not to be held on the stack, since it can be recovered from s, if needed. The current input symbol is now a_{i+1}.

2. If $ACTION[s_m, a_i]$ = reduce $A \to \beta$, then the parser executes a reduce move, entering the configuration:

$$(s_0 s_1 ... s_{m-r} s, a_i a_{i+1} ... a_n \$)$$

where r is the length of β, and $s = GOTO[s_{m-r}, A]$. Here the parser first popped r state symbols off the stack, exposing state s_{m-r}. The parser then pushed s, the entry for $GOTO[s_{m-r}, A]$, on to the stack. The current input symbol is not changed in a reduce move. For the LR parsers we will construct $X_{m-r+1} ... X_m$, the sequence of grammar symbols corresponding to the states popped off the stack, will always match β, the right side of a reducing production. Note that, X_i stands for terminals/non-terminals.

Output of the LR parser is generated after a reduce move by executing the semantic action associated with the reducing production.

3. If $ACTION[s_m, a_i]$ = accept, parsing is complete.

4. If $ACTION[s_m, a_i]$ = error, it calls for error routine.

The LR parser algorithm is given below as Algorithm- 1. All the LR parser behave in the similar way, and the only difference between them is the variation in functionalities of the ACTION and GOTO functions. Initially, the parser has s_0 on its stack, where s_0 is initial state, and w \$ in the input buffer.

Algorithm 1 LR(1)-parsing Algorithm (Input: String w and LR-parsing table with functions ACTION and GOTO for a grammar G, Output: IF $w \in L(G)$, output is reduction steps of a bottom-up parse for w, otherwise outputs error)

```
1: Initial state: a is first input symbol of w $
2: push s_0 on stack
3: while (True) do
4:    Let s state is on top of stack
5:    if (ACTION[s, a] = shift s_i) then
6:       push state s_i onto stack
7:       let a be the next input symbol;
8:    else
9:       if (ACTION[s, a] = reduce A → β) then
10:         pop |β| symbols off the stack
11:         let state s now be on the top of the stack
12:         output the production A → β
13:      else
14:         if (ACTION[s, a]= accept) then
15:            accept & break loop
16:         else
17:            call error-recovery routine
18:         end if
19:      end if
20:   end if
21: end while
```

Entries in the ACTION table are encoded using the letters s for shift and r for reduce. Thus, the entry s 4 indicates the action "shift and go to state s 4," while r 2 indicates "reduce by production 2." The GOTO table encodes the transition that must be taken after a reduce action. Its sparsity reflects the fact that relatively few states represent reductions.

Example 20.1 *Construct the LR-parsing table for the augmented expression grammar table:*

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to id$

The codes for action are as follows:

1. si means shift and stack state i
2. rj means reduce by the production number numbered j
3. acc means accept
4. blank means error

Note that the value of $GOTO[s, a]$ for terminal a is found in the ACTION field connected with the shift action on input a for state s. The GOTO field gives $GOTO[s, A]$ for nonterminal A.

20.2 Construction of SLR-Parsing Tables

The SLR method begins with LR(0) item and LR(0) automata. That is, given a grammar G, we augment G to produce G', with a new symbol $'$. From G' we construct C, the canonical collection of sets of items for G' together with GOTO function.

The ACTION and GOTO entries in the parsing table are then constructed using the Algorithm 2. For this we must know FOLLOW(A) for each nonterminal A of a grammar.

The parsing table determined by above algorithm is called $SLR(1)$ table for G. The parser is called $SLR(1)$ parser. However, 1 is omitted, and the parser is called SLR. This is because, more than one lookahead is not common.

Example 20.2 *Construct SLR table for augmented expression grammar.*

We consider the set of items of LR(0) given in Fig. 20.2 as follows:

Algorithm 2 Constructing-SLR-parsing-table(Input: Augmented grammar G', Output: SLR-parsing table functions ACTION and GOTO for G')

Construct $C = \{I_0, I_1, ..., I_n\}$, the collection sets of LR(0) item sets for G'
State i is constructed from I_i. The parsing actions for i are determined as follows:
if $[A \to \alpha.a\beta] \in I_i$ and $GOTO(I_i, a) \in I_j$ **then**
    set $ACTION[i, a]$ to "shift j." ; a is terminal
end if
if $[A \to \alpha.] \in I_i$ **then**
    set $ACTION[i, a]$ to "reduce $A \to \alpha$" (for all $a \in FOLLOW(A)$, and $A \neq S'$
end if
if $[S' \to S.] \in I_i$ **then**
    set $ACTION[i, \$]$ to "accept."
end if
;GOTO for state i are constructed for all nonterminals A using the rule:
if $GOTO(I_i, A) = I_j$ **then**
    $GOTO[i, A] = j$
end if
Mark all empty entries as error
Initial state of the parser is constructed from set of items containing $[S \to S']$.

$E' \to .E$
$E \to .E + T$
$E \to .T$
$T \to .T \times F$
$T \to .F$
$F \to .(E)$
$F \to .id$

Table 20.2: Parsing table for expression Grammar

STATE	ACTION table						GOTO-table		
	id	+	*	(	)	$	E	T	F
0	s4			s3			1	8	5
1		s2				acc			
2	s4			s3				6	5
3	s4			s3			7	8	5
4		r6	r6		r6	r6			
5		r4	r4		r4	r4			
6		r1	s10		r1	r1			
7		s2			s11				
8		r2	s10		r2	r2			
9		r3	r3		r3	r3			
10	s4			s3					9
11	r5	r5			r5	r5			

The LR(1) parser table comprising ACTION table and GOTO table for expression grammar and item sets of Fig. fig-automgrm, is shown as Table 20.2. The entries in this table are self explanatory, for example, when state is 0, and input is *id*, we note from Fig. 20.2 that we shift state of "$F \to id.$", i.e., state 4 (or I_4) to stack. Wen input is '+', at state 1, there is

a transition to state 2.

When stack on top of stack is 4, and input is '+', there is no transition, but there us reduction due to "$F \rightarrow id$." rule.

The item $F \rightarrow .(E)$ gives rise to the entry $ACTION[0, (] = shift\ 3$, and item $F \rightarrow .id$ to entry $ACTION[0, id] = shift\ 4$. Other items in I_0 result to no actions.

Now consider I_1: $E' \rightarrow E.$, and $E \rightarrow E.*T$: Since $FOLLOW(E) = \{\$, +,)\}$, the first item yields $ACTION[1, \$] = accept$, and the second yields $ACTION[1, +] = shift\ 2$.

□

20.3 Review Questions

1. In LR(0) parser, when top of stack is β, such that there is a production $A \rightarrow \beta$, is it always advisable to reduce it always, and not to consider any scope to shift the next input terminal to shift it to stack? Justify your answer.
2. What ia main difference between LR(0) and LR(1) parser?
3. What is common part between LR(0) and LR(1) parser?
4. Where is no reduce state after I_6 due to "$E \rightarrow E + T.$" in Fig. 20.2?
5. What are all the four forms of ACTION functions in a LR(1) parser?

20.4 Exercises

1. For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$, indicate the handle in each of the right-sentential forms:
   (a) 000111
   (b) 00S11
2. For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$, give the bottom-up parses for the following input strings 000111.
3. For the grammar $S \rightarrow SS+ \mid SS* \mid a$, indicate the handle in each of the following following right-sentential forms:
   (a) $SSS + a * +$
   (b) $SS + a * a+$
   (c) $aaa * a + +$
4. For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$, give the bottom-up parsing for the input string 000111.
5. For the grammar $S \rightarrow SS+ \mid SS* \mid a$, give the bottom-up parsing for the input string $aaa * a + +$.
6. Use the grammar: $S \rightarrow aSb \mid bSa \mid c$, to parse the following expressions using shift-reduce parsing:

(a) acb

(b) bca

(c) a^2cb^2

(d) $a^2b^2ca^2b^2$

7. If there are more than one handle available at the same time in a sentential, does it imply that corresponding grammar ambiguous? Justify your answer for Yes/No, and given examples in support or in counter.

8. Use shift-reduce parser to parse the following expressions, using expression grammar:

   (a) $id * id + id$

   (b) $id * id/id$

   (c) $id * id + id * id$

9. In the Fig. 20.2, how many states are reduce states? They corresponds to what property of the grammar?

10. Why there is no *accept* state after state I_6 due to "$E \to E + T.$" in the Fig. 20.2?

11. Explain the configurations of the LR(1) parser.

12. Write down the LR(1) algorithm in the form of description in your own language.

13. For the expression grammar and the expression $id + id$, show all the steps, progressive stack contents, ACTION and GOTO computations for the LR(1) parser.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, et al., Sep 10, 2006.

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990.

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

[4] Tools for Large-scale Parser Development, Proceedings of the COLING-2000 Workshop on Efficiency In Large-Scale Parsing Systems, 2000, pp. 54-54, `http://dl.acm.org/citation.cfm?id=2387596.2387604`.