CLR PARSING

CLR Parsing

After discussing the SLR parsing and the problems associated with the SLR parsers, this module will discuss the powerful LR parser – Canonical LR parser. In this module, we will learn to construct LR(1) items which is necessary for constructing the CALR parsing table and using this table parse a given string using the CALR parser.

1. Need for CALR parsers

In the SLR parser, there is a problem of shift / reduce conflict even if the grammar is unambiguous. This is due to the fact that the SLR parsers uses the FOLLOW() information to perform a reduce action by matching the stack information with input symbol. However the FOLLOW() information alone is not sufficient to decide when to reduce. Hence, powerful parser is required.

The issues associated with considering the FOLLOW() information is discussed as follows:

- In SLR, if there is a production of the form $A \rightarrow \alpha\blacksquare$, then a reduce action takes place based on FOLLOW(A).
- However, there would be situations, where, when state 'I' appears on the top of the stack, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by terminal 'a' in a right sentential form. Hence, the reduction $A \rightarrow \alpha$ would be invalid on input 'a'

This results in the shift/reduce conflict. To resolve this conflict, we will consider and check whether it is possible to perform more in the states that allow us to rule out some of the invalid reduction. This is done by introducing more set of items thus resulting in more states in the CALR parsing table. Thus we would be introducing exactly which input symbols to follow a particular non-terminal.

2. CALR Parser

The steps involved in the CALR parser are as follows:

- Construct LR(1) items – This is in contrast with the LR(0) items that is constructed for the SLR parser. This also uses Closure() and goto(), but the algorithm for these two functions are different.
- LR(1) items are used to construct the CALR parsing table involving action, goto.- The parsing table resembles SLR parsing table but has more states and there is little variation in the construction procedure.
- Use this table, along with input string and a stack is used to parse the string – The parsing action is same as the SLR parser's algorithm.

The CALR parser uses the LR(1) items. The LR(1) items are constructed and this results in increased number of states. The states are increased by accommodating an extra symbol in the items to include a terminal symbol as a second component. Thus A →[α .β, a] will be the item in the LR(1) items collection, if A →αβ is a production and 'a' is a terminal or the right end marker. If there is no terminal available then the right end marker is $.

1. LR(1) item construction

LR(1) items are constructed that has a right end marker in addition to the format of the LR(0) items. The '1' refers to the length of the second component which is the look-ahead of the item. This look-ahead has no effect in A →[α .β , a] where β is not ε, but will ensure that a conflict does not arise if A →[α . , a] calls for a reduction A → α if the next input symbol is 'a'. This terminal 'a' will be subset of FOLLOW(A). A → [α .β , a] is a valid item for a viable prefix γ if there is a derivation S => δAw => δαβw where γ = δα and either 'a' is the first symbol of 'w' or 'w' is ε and 'a' is $.

LR(1) items construction requires computation of closure() and goto(). The computation of Closure(I) is given in algorithm 17.1. This algorithm is similar to the LR(0)'s closure in considering the augmented grammar to start this, but will accommodate the look-ahead component.

Algorithm 17.1

Closure(I, Augment grammar G')

{
repeat
a. for each item [A → α▪Bβ, a] in I,
each production B → γ in G'and each terminal b in FIRST(βa)
such that [B → .γ , b] is not in I do
1. add [B → .γ , b]
until no more items can be added to I
}

Step 'a' of the algorithm 17.1 initially starts by adding the initial production of the augmented grammar as an item with the look-ahead as $. After that it considers the non-terminal that appears after the dot. This item is added and its look-ahead is computed by computing the FIRST() of the remaining symbols after this non-terminal including the current look-ahead. So, if β is ε, even than as 'a' is '$' to start with, the look-ahead will be FIRST($). We keep adding till no more items can be added. Thus the difference between LR(1) and LR(0) is that in considering FIRST(βa) and adding 'a' as a look-ahead. It is interesting to observe that a single item-set may contain the same items with different look-aheads.

The next algorithm is to compute the goto(I, X) where X is a grammar symbol. This is given in Algorithm 17.2. This algorithm is the same as LR(0)'s goto() but this incorporates the look- ahead symbol.

Algorithm 17.2

goto(I,X)

{
    a. Let J be the set of items [A→αX.β , a] such that
        [A→α.Xβ , a] is in I;
    Return closure(J)
}

Step 'a' of algorithm 17.2, shifts the dot by one position to the right of an item, retaining the look-ahead of the original item. Then after shifting, we compute closure of the shifted item and add that to the set of items. So, if β is a non-terminal, we add more items to the current item set with different / same look-ahead and if it is a terminal we do not have any more new items to the current item set.

After computing the closure() and goto(), we use these two functions to compute LR(1) items and is given in algorithm 17.3

Algorithm 17.3

ITEMS(Grammar G')

{   C:= closure ({S' → .S, $});
    repeat
    for each set of items I in C and each grammar symbol X such that goto(I,X) is not empty and not in C
        add goto(I, X) to C
    until no more set of items can be added to C
}

This algorithm is same as the LR(0) items construction algorithm, but it considers the items with look-ahead.

Example 17.1 Consider the following grammar and construct the LR(1) items

S → CC
C → cC
C → d

We form the Augmented grammar by introducing the new start symbol S' and form the set of items and is given in table 17.1

S' → S
S → CC
C → cC
C → d

Table 17.1 Set of LR(1) items

Item	Set of Items	Goto(I, X)	Comments
I_0	S' → .S, $ S → .CC, $ C → .cC, c/d C → .d, c/d		This is the initial item. We have a non-terminal S after the dot. So we add the productions of S, with look-ahead as FIRST($) since β is ε. Now again we have non-terminal C after the dot and here β is 'C" and 'a' is $. So, we add the productions of C with lookahead as FIRST(C$). FIRST(C) = {c, d} from the two productions of C. Thus we add two items for each of the productions of C one with 'c' and other with 'd' as look-ahead. However, we could represent it in a combined fashion as given in this items set.
I_1	S' → S., $	(I_0, S)	Shifting the dot results in a kernel item, the look-ahead remains the same.
I_2	S → C.C, $ C → .cC, $ C → .d, $	(I_0, C)	The dot is shifted by one position to the right. Now we have C after the dot. β is ε and we add the items of C with FIRST($) as look-ahead.
I_3	C → c.C, c/d C → .cC, c/d C → .d, c/d	(I_0, c), (I_3, c),	Shifting the dot by one position and keeping the initial look ahead as it is results in the first item. Now we have a C after the dot. β is ε and we add the items of C with FIRST(c/d) as look-ahead.
I_4	C → d., c/d	(I_0, d), (I_3, d)	Kernel item with the look-ahead being the same
I_5	S → CC., $	(I_2, C)	Kernel item
I_6	C → c.C, $ C → .cC, $ C → .d, $	(I_2, c) (I_6, c)	The dot is shifted by one position to the right. Now we have C after the dot. β is ε and we add the items of C with FIRST($) as look-ahead.
I_7	C → d., $	(I_2, d) (I_6, d)	Kernel item
I_8	C → cC., c/d	(I_3, C)	Kernel item
I_9	C → cC., $	(I_6, C)	Kernel item and no more new items are necessary to be added.

The LR(1) items can also be represented as a DFA similar to the LR(0) items where the states correspond to the nodes and edges correspond to the grammar symbols.

2. CALR Parsing Table

If we could recollect the SLR parsing table requires the knowledge of the FOLLOW() of the non-terminals. This FOLLOW() set is used to populate the SLR parsing table for the reduce action. This is however not required here as the look-ahead which is conveyed by the FOLLOW() in the SLR parsing table, is available along with the item itself in the LR(1) items. The CALR parsing table also has two divisions: action and goto. The action() fields are constituted by the terminals and it has the shift, reduce, accept and error actions. The goto() fields are constituted by the non-terminals and it contains the state numbers which is the result of the goto(). The procedure for the CALR parsing table construction is given in Algorithm 17.4.

Algorithm 17.4

CALR_ParsingTable (Augmented Grammar G')

{

1. Construct C = {I_0 ,I_1 ,I_2 … I_n } the collection of LR(1) items for G'
2. State 'i' of the parser is from I_i
   i. if [A → α.aβ, b] is in I_i and goto(I_i, a) = I_j set action [i, a] = shift j, where a is a terminal
   ii. if [A → α . , a] is in I_i and A ≠ S', then set action[i, a] = reduce by A → α
      a. //a conflict here implies the grammar is not CALR grammar
   iii. if [S' → .S, $] implies an accept action at action[i,$] = accept
   iv. all other entries are error

3. If goto(I_i , A) = I_j then goto (i, A) = j
4. All other entries are error

}

Step 1 of the algorithm 17.4 calls for the construction of the LR(1) items. Step 2 has four actions which are used to construct the action field of the CALR parsing table. The first one is a shift action which is the same as the SLR table's shift action. If goto(I_i, a) = I_j then at the intersection of [i, a] we set the action as "sj" to indicate "shift j". Step 2 (ii) of the algorithm is for reduce action where the kernel items are considered. At the table entry of kernel item number and the look-ahead symbol we add the action reduce by the production indicated by the kernel item. The item number that has the initial kernel item is used to indicate the accept action as in the SLR parsing table. All other entries are considered as error in the action field of the CALR parsing table. The goto() field is the same as the SLR table's goto field. If goto(I_i , A) = I_j then goto (i, A) = j is added to the CALR parsing table.

Example 17.2

For the grammar discussed in example 17.1, let us construct the CALR parsing table based on the set of items discussed in Table 17.1. The CALR parsing table is given in Table 17.2

Table 17.2 CALR parsing table

State	Action			Goto		Comments
	c	d	$	S	C	
0	s3	s4		1	2	Goto(I_0 ,c) = I_3 , => [0,c] = s3 Goto(I_0 ,d) = I_4=> [0,d] = s4 Goto(I_0 ,S) = I_1 => [0,S] = 1 Goto(I_0 ,C) = I_2 => [0,C] = 2
1			accept			I_1 has [S' → S., $] so at [1, $] we have accept action
2	s6	s7			5	Goto(I_2 ,c) = I_6 , => [2,c] = s6 Goto(I_2 ,d) = I_7=> [2,d] = s7 Goto(I_2 ,C) = I_5 => [2,C] = 5
3	s3	s4			8	Goto(I_3 ,c) = I_3 , => [3,c] = s3 Goto(I_3 ,d) = I_4=> [3,d] = s4 Goto(I_3 ,C) = I_8 => [3,C] = 8
4	r3	r3				C → d., c/d, so at the intersection of [4, c] and [4,d] we set reduce by C → d
5			r1			S → CC., $, at the intersection of [5, $] we set reduce by S → CC
6	s6	s7			9	Goto(I_6 ,c) = I_6 , => [6,c] = s6 Goto(I_6 ,d) = I_7=> [6,d] = s7 Goto(I_6 ,C) = I_9 => [6,C] = 9
7			r3			C → d., $, we set at the intersection of [7, $] we set reduce by C → d
8	r2	r2				C → cC., c/d at the intersection of [8,c] and [8,d] we set reduce by C → cC
9			r2			C → cC., $ at the intersection of [9,$] we set reduce by C → cC

17.2.3 CALR parsing

CALR parsing action is exactly same as the SLR parsing action but this is done with the help of CALR parsing table. The stack is initialized with the state 0. The stack contains alternately state number and the grammar symbol with the state number on the stack. The input is appended with $. The stack symbol and the input are compared in the table and the stack is manipulated accordingly. The CALR parsing table is given in Algorithm 17.5

Algortihm 17.5

CALR Parsing Table (Table T, Input w$)

{

- Set input to point to the first symbol of w$
- Repeat forever
  - Let s be the state on the top of the stack
  - Let a be the symbol pointed to by ip
  - If action [s, a] = shift s' then
    - Push a then s' on top of the stack
    - Move input to the next input symbol
  - Else if action [s, a] = reduce A → β then
    - Pop 2 * | β | symbols off the stack
  - Let s' be the state now on the top of the stack
    - Push A then goto [s', A] on top of the stack
    - Output the production A → β
  - Else if action[s, a] = accept then return;
  - Else error()

}

Example 17.3
Table 17.2 could be used to parse the string "ccdd" and is explained in table 17.3

Table 17.3 CALR parser's action on the input "ccdd$"

Stack	Input	Comments
0	ccdd$	[0, c] – shift 3 , push c, 3 onto the stack
0 c 3	c d d $	[3, c] – shift 3 , push c, 3 onto the stack
0 c 3 c 3	d d $	[3, d] – shift 4 , push d , 4 onto the stack
0 c 3 c 3 d 4	d $	[4, d] – reduce 3, pop 2 symbols from stack, as it corresponds to the production C → d, push C and goto(3, C) = 8
0 c 3 c 3 C 8	d $	[8, d] – reduce 2, pop 4 symbols from the stack as it corresponds to the production C → cC, push C and goto(3, C) = 8
0 c 3 C 8	d $	[8, d] – reduce 2, pop 4 symbols from the stack as it corresponds to C → cC, push C and goto(0, C) = 2
0 C 2	d $	[2, d] – shift 7, push d, 7 onto the stack
0 C 2 d 7	$	[7, $] – reduce 3, pop 2 symbols from the stack for the production C →d, and push C, goto(2, C) = 5
0 C 2 C 5	$	[5, $] – reduce 1, pop 4 symbols off the stack corresponding to the production S → CC, and push S, goto(0, S) = 1
0 S 1	$	[1, $] – accept – successful parsing

Any other combination would result in the error combination and the CALR parser has to recover from errors to accommodate an incorrect input.

Example 17.3

We considered the pointer variable declaration grammar having the shift / reduce conflict in the SLR parsing table. Let us construct the CALR parsing table and verify whether it is a CALR grammar.

- *S'*→ *S*
- *S* → *L* = *R*
- *S*→ *R*
- *L* → * *R*
- L → id
- *R* → *L*

The LR(1) items is given in Table 17.4 and the parsing table is given in Table 17.5

Table 17.4 Set of Items for the pointer grammar

Item	**Set of Items**	**Goto(I, X)**	**Comments**
I_0	S' → .S, $		Initial item. Then all the items need to be added with '$' as look ahead for S, R. But for L we have two look-ahead '$' and '=' one from S→ .L=R and other from R → .L.
	S → •L=R, $		
	S → •R,$		
	L → •*R,=/$		
	L → •id,=/$		
	R → •L,$		
I_1	S' → •S,$	(I_0,S)	Kernel item to result in accept action
I_2	S → L•=R,$	(I_0,L)	After the dot we have a terminal and hence no additional items need to be added
	R → L•, $		Kernel item
I_3	S → R•, $	(I_0,R)	Kernel item
I_4	L → *•R,=/$	(I_0,*) (I_4,*)	Items of R to be added with the same look-ahead which results in addition of the items corresponding to R and in –turn L
	R → •L,=/$		
	L → •*R,=/$		
	L → •id, =/$		
I_5	L → id•,=/$	(I_0,id) (I_4,id)	Kernel item
I_6	S → L=•R,$	(I_2,=)	Items of R to be added with same look ahead and in-turn items of L are added.
	R → •L, $		
	L → •*R, $		
	L →•id, $		
I_7	L → *R•,=/$	(I_4,R)	Kernel item
I_8	R → L•,=/$	(I_4,L)	Kernel item
I_9	S → L=R•,$	(I_6,R)	Kernel item
I_{10}	R → L•,$]	(I_6,L), (I_{11},L)	Kernel item

I_{11}	L → *•R,$ R → •L,$ L → •*R,$ L → •id, $	(I_6,*) (I_{11},*)	This is a new item and is different from I_4 because they have a different look-ahead
I_{12}	L → id•,$]	(I_6,id) (I_{11},id)	Kernel item
I_{13}	L → *R•, $	(I_{11},id)	Kernel item

Table 17.5 CALR Parsing Table

State	Action				Goto		
	id	*****	**=**	**$**	**S**	**L**	**R**
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

As can be seen from the Table 17.5, the table does not have any conflict but the number of set of items is 14 as compared to 10 in the SLR parsing table.

Summary:

In this module we discussed the CALR parser which is the most powerful parser. This parser constructs the parsing table by constructing the LR(1) items. This parser has many items as compared to the LR(0) items. The increased number of items is compensated with reduction in shift/reduce conflicts.