International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878, Volume-8 Issue-3, September 2019

Bottom-Up Parser: Look-Ahead LR Parser

Pooja Rani

Abstract: ***Compiler is used for the purpose of converting high level code to machine code. For doing this procedure we have six steps. On these steps the syntax analyses is the second step of compiler. The lexical analyzer produce token in the output. The tokens are used as input to syntax analyzer. Syntax analyzer performs parsing operation. The parsing can be used for deriving the string from the given grammar called as derivation. It depend upon how derivation will be performed either top down or bottom up. The bottom up parsers LR (Left-to-right), SLR (simple LR) has some conflicts. To remove these conflicts we use LALR (Look ahead LR parser). The conflicts are available if the state contains minimum two or more productions. If there is one shift operation in state and other one is reduce operation it means that shift-reduce operation at the same time. Then this state is called as inadequate state. This Inadequate state problem is solved in LALR parser. Other problem with other parsers is that they have more states as compared to LALR parser. So cost will be high. But in LALR parser minimum states used and cost will automatically be reduced. LALR is also called as Minimization algorithm of CLR (Canonical LR parser).***

Keyword: lexical analyzer, syntax analyzer, shift, reduce, look ahead parser.

I. INTRODUCTION

The parsing can be defined as the process of deriving the string from the given grammar is known as parsing. The parsing is of two types:
Top-Down
Bottom-Up
With the top down parsing if the production contain more than one possibility selecting the correct one is always the difficult in top down parser. So, the bottom up parser we used. Bottom up parser starts from string and proceeds to start symbol. It follows the reverse of right most derivation. Identifying the correct handle sometimes difficult in bottom up parser. The bottom up parsing is as follows:

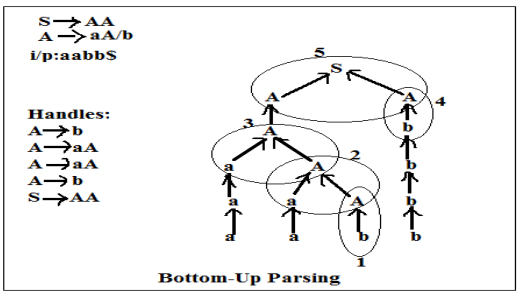


Fig.1. Bottom-Up Parser

Manuscript published on 30 September 2019
* Correspondence Author
Pooja Rani*, Department of Computer Science at HP University, Shimla, India.

In compiler the bottom up parser are two types:
1. LR Parser.
2. Operator Precedence Parser.
The LR parser is unambiguous in nature. Here we don't need to bother about the left recursion, right recursion and left factoring. The LR parser again split into two types:
1. LR (0)-items
2. LR (1)-items
These are defined in following figure:

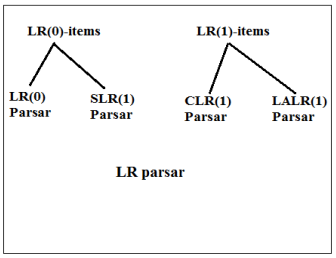


Fig.2. LR Parser Types

The CLR (1) is the best bottom-up parser. The LALR (1) is the minimization algorithm of CLR (1). Bottom-up parsers table size is approximate 2 times more than the top down parser table size. Bottom up grammar provide more grammar. Bottom up parsers are more powerful because it accepted more grammer.In bottom up parser the goto part of the table not change. The reduce entries change so the blank spaces may increased more. The conflicts present in the LR (0) and SLR (1) are the shift-reduce and the reduce-reduce conflicts. The shift –reduce conflicts are the conflicts where in state the shift and reduce operation performed simultaneously. The shift can be defined as where the closure is move formed and the reduce can be defined as where the closure reached at the end of production and we call it as reduce. When these two operation comes simultaneously we call it shift-reduce conflicts. For example: We have the states as:

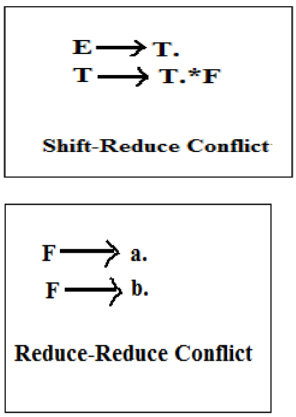


Fig.3. Types of conflicts

The reduce-reduce conflicts can be defined as where the two productions closure reached at the end in the same state so we call it as reduce-reduce conflicts.

Retrieval Number C4666098319/2019©BEIESP
DOI: 10.35940/ijrte.C4666.098319
Journal Website: www.ijrte.org

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

This conflict is shown in above images. These conflicts are solved in CLR (1) and LALR (1) parsers.

II. LOOK AHEAD PARSER

Look-Ahead is the bottom up parser. This is used for deriving the string from the grammar. It starts parsing from the bottom of the parsing tree. There are some conflicts occur in the LR (0) and SLR (1) parser. So to solve those conflicts we use CLR (1) parser but with CLR (1) states will be more. That cause the high cost. So, what is use of parser if the cost will be high? So, we use the LALR (1) parser which combines the same states. The LALR (1) parser is the subset of LR (1) and superset of SLR (1).The states in the LALR (1) parser is same as the SLR (1) parser and LR (0) parser. If the given grammar is LR(0) grammar then that grammar automatically will be the SLR(1),LALR(1) and CLR(1).If the grammar not the LR(0) means that there are conflicts present in the LR(0) then that conflicts definitely present in the LAR(1) parser. The grammar that is SLR (1) that grammar definitely LALR (1).The grammar that is not SLR (1) may or may not be LALR (1) depending upon the wheather the look ahead resolve the SLR (1) conflicts.
LALR (1) parser proved to the most variant of the LR family. The LR (0) and SLR (1) parser built to be means of handling a small set of grammars. But LALR (1) handles all set of grammars. The LALR (1) is the intractable approach. the expansive memory in LR(1) caused it languish for several years as compared to theoretically.LALR(1) make it possible that there will be good balance between the table size and specific lookaheads.The popular tools used here:yacc and bison. These tools generate LALR (1) parser. The most of the programming languages constructs can be described with the LALR (1) grammar. LALR (1) parser is under the type of LR parser. So, every LR grammar is unambiguous but every need not to be LR.For all LR parsers, parsing algorithm will be same but parsing table will be different. Here the parsing is depends upon the look-ahead symbols. The look-ahead means that the terminals in the grammar.
The LR (1)-items in LR parser are as follows:
LR (0)-items + Look Ahead symbols=LR (1)-items

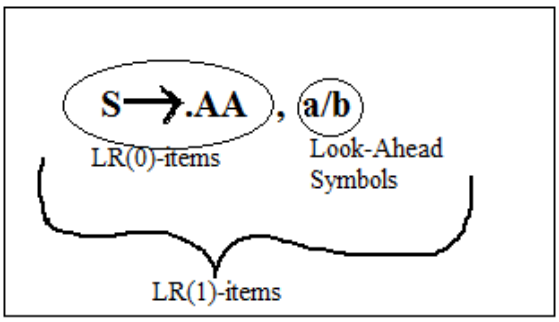


Fig.4. Example of LR (1) items

A. *Algorithm of Look-Ahead LR Parser*

Let x be the state number on the top of stack and a will be the look-Ahead symbol.
1. If action[x,a]=si(shift)
then shift a&&i and increment input pointer.
2. If action[x,a]=rj(reduce)and jth production is g=>h
then pop2*|h| symbols and replace by g.If Sm-1 is the state below g then push go—to[Sm-1,g].
3. If action[x,a]=blank then parsing error.
4. If action[x,a]=accept then successfully completion of parsing.

B. Definition

Closure (I):
1. Add I
2. If A→B.CDE,a/b is I and C→ECG is in G then add C→.ECG, Fi(DE, a/b) to closure(I)
3. Repeat second step for every newly added LR(1)-item.

Example:
S→AA
A→aA/b

G'=S'→S
S→AA
A→aA/b

Closure(S'→.S, $) =S'→.S, $
S→.AA,$
A→.aA,a/b
.b,a/b
Closure(S→A.A,a/b)=S→A.A,a/b
A→.aA,a/b
.b,a/b

GOTO:
GOTO (I,x)
1. Add (I) by moving dot after x.
2. Find closure (1st step) previous definition.

Example:
Goto(S'→.S, a/b ,S)=S'→S. ,a/b
Goto(S→.AA, c/d , A)=S→A.A ,c/d
A→.aA, c/d
.b, c/d
Goto(S→.aA, a/b, a)=A→a.A ,a/b
A→.aA , a/b
.b, a/b

Let take an example where we can apply the LALR (1) parser. For parsing our grammar we have to create the DFA of states. The given grammar as follows:

S→AA
A→aA
A→b
The augmented grammar as follows
S'→S
S→AA
A→Aa/b
Closure(S'→.S,$)
Here start the DFA for LALR(1) parser. Start with state I0.below shows the state I0:
S'→.S, $
S→.AA, $
A→.aA,a/b
A→.b,a/b

This is state I0.Now we see in state I0 closure present at the starting of all the productions.So, start the shift operation on this state and accordingly apply the reduce operation w.r.t to the look ahead symbols. Start the shift operation of S variable. After done the shift operation on S we has following state namely I1 that goes from state I0.

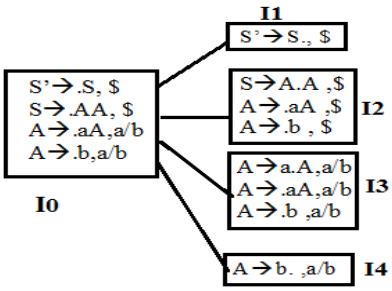
S'→S., $

This state automatically reduced. Here no need of shift operation. Next start to shift variable A from state I0 to I2.The state I2 is as follows:

S→A.A ,$
A→.aA ,$
A→.b , $

Here we see in state I2 we derive two production below the first production because in first production closure is present before the variable A.We write all the production of A.Now take the terminal a from I0 and apply shift operation.This state is I3:
A→a.A,a/b
A→.aA,a/b
A→.b ,a/b
Now take the terminal b from I0 state and apply the shift operation.We call this state as I4.The state I4 is as follows:

A→b. ,a/b
After processing all the variables and terminals in the state I0 the state I0 to I4 is shown below.



Now start processing the state I1 but state I1 is already completed or we can say reduced. So, no need of performing shifts operation on it. Take the state I2 and start processing it. From state I2 take the production 1 where we have to shift the A. after shifting we have following production. We name this state as I5.

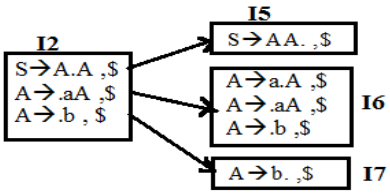
S→AA. ,$

Now, take the second production of stateI2 and perform the shift operation. This state names it I6.the following are the productions in state I6:

A→a.A ,$
A→.aA ,$
A→.b ,$

Now take the third production of I2 and apply the shift operation on it.After shift operation the state we called as I7 which is defined as below:
A→b, $

After processing state I2 we have following states which are defined as shown below.

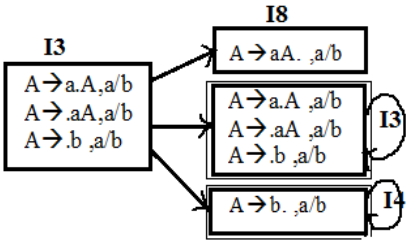


After processing state I2.Now take the state I3 for processing.Take the 1st production of state I3.Process/shifting perform on it. By performing shift opearion the production is reduced.This state named as I8.after shifting the production is as follows:

A→aA. ,a/b
Now take the second proction of I3 and perfom shift operation.Below shows the production as:

A→a.A ,a/b
A→.aA ,a/b
A→.b ,a/b

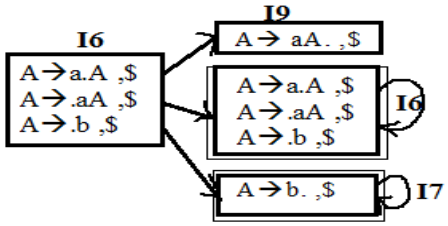
This state is repeated so we named it as name of state whose content is repeated in this state. i.e state I3.
Now take the last production of state I3 and perform the shift operation. This state again repeated so we name this state as I4.The production is as follows:
A→b. ,a/b
After performing this operation all the production of state I3 is completed.The following figure shows the LALR parsing of state I3.



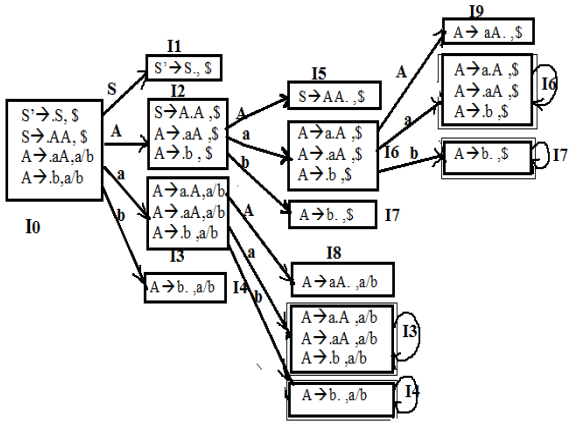
Now we see that the state I4 is already reduced. The state I5 is also reduced. So, we take state I6.We take the production 1 of state I6.After shifting we have the production as:
A→ aA. ,$
Now take the second production of state I6 and performing the shift operation on it.We has the following production as follows:
A→a.A ,$
A→.aA ,$
A→.b ,$
Now take the third production of I6 and performing the shift operation on it. By doing so the production automatically is reduce. We named as state I7 because it is similar to the state I7.

A→b. ,$

The LALR parsing of state I6 is shown below.



Hence all the productions are completed. All the productions of given grammar is reduced. The all parsing of states is shown below.



We see in the above DFA the states are same only difference in between the look ahead symbols. this is disadvantages in CLR(1) parser that if two states are differ by only one look ahead symbol we are considering it as two states. If Number of states are more the cost will be also high. So, minimization of states are necessary. This is done in LALR (1) parser. So, LALR(1) parser is called as minimization algorithm of CLR(1).In LALR(1) parser if two states differ by only one look ahead symbol we are considering it as single by combing states.

III. PARSING TABLE CONSTRUCTION

A. Brute Force Approach

There are two approaches to construct the parsing table but we study here the only one approach i.e. brute force approach. In this approach first LR (1) parsing table is constructed and the similar set is merged together. This is called as brute force approach of making parsing table. The compression of LR table into LALR (1) version is straightforward.

Steps:

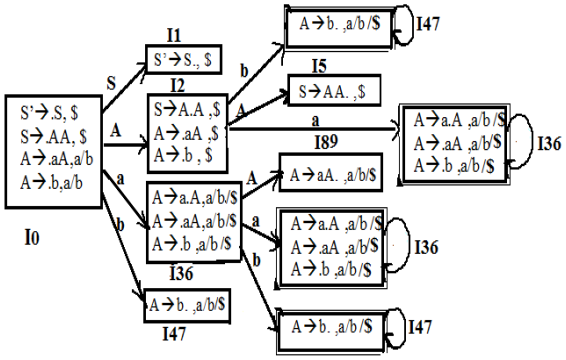
1. Construct all canonical LR (1) states.
2. Combine the states which are similar in the productions as well as lookaheads.If the states are differ only in lokaahead then also we can merge those states in LALR(1) parsing table.
3. The successor function for new LALR (1) state is the union of the successors of the merged states.
4. The action and goto entries in LALR (1) is same as entries in LR (1) parser.

Table-I: Parsing table without minimization

States	Action			Go To	
	a	b	$	S	A
0	S3	S4		1	2
1			accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

B. Minimization of DFA

We see in the above DFA the states three and six are same in production with closure but different in look ahead symbols then we take them as one state named as state number 36.The states four and seven are same in production with closure but different in look ahead symbols so we considered them as single state and named as state number 47.The states eight and nine are same in production with closure but different in look ahead symbols so we considered them as single state and named as state number 89.The above DFA is minimized in LALR (1) as shown below.



Minimized Form of CLR(1)=LALR(1)

C. LALR (1) Parsing Table Construction

A LALR(1) parsing table is built from states of DFA in the same way as canonical LR(1).The look ahead symbols tells us where we have to place the reduce actions. If in the DFA of LALR(1) all the states are different then that DFA is similar of CLR(1) DFA and the parsing table of LALR(1) also similar to the parsing table of CLR(1) and we gain nothing. So, we can say that in LALR(1) some states will be merged and has fewer rows than the CLR(1) and LR(0).There are so many rows available in LR table for programming language i.e. thousand rows which will be merged in LALR(1) into some hundred of rows. Due to the merging in LALR (1), the table looks similar to LR (0) and SLR (1), we can say all three has same no. of states but in LALR (1) the reduce actions has fewer.

Retrieval Number C4666098319/2019©BEIESP
DOI: 10.35940/ijrte.C4666.098319
Journal Website: www.ijrte.org

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

Table-II: Look ahead Parsing Table

States	Action			GoTo	
	a	b	$	S	A
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

IV. CONCLUSION

In this study, the bottom-up parser look ahead LR parser is introduced with example. I started first with introduction of parsing. After that parsing types are explained with conflicts. Hence I conclude that LALR parser is minimized form of CLR parser. LALR parser has less state as compared to CLR parser. So the cost will be low.

REFERENCES

1. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf
2. http://www.fit.vutbr.cz/~izemek/grants.php.cs?file=%2Fproj%2F589%2FPresentations%2FPB06-Lookahead-In-Translators.pdf&id=589
3. https://parasol.tamu.edu/~rwerger/Courses/434/lec10.pdf
4. http://user.it.uu.se/~kostis/Teaching/KT1-12/Slides/handout08.pdf
5. http://www.cs.ecu.edu/karl/5220/spr16/Notes/Bottom-up/lalr.html
6. http://ecomputernotes.com/compiler-design/lr-parsers
7. https://web.cs.dal.ca/~sjackson/lalr1.html
8. https://www.javatpoint.com/lalr-1-parsing

AUTHOR PROFILE

Pooja Rani is Guest Faculty in Department of Computer Science at HP University, Shimla, India. She has been completed her M.Tech from Central University of Punjab, Bathinda.She has been completed her B.Tech from HPTU Hamirpur. She has been awarded with Gold Medal in B.Tech. She is NET Qualified. Her research interests are Machine Learning, Text Analytics and Learning Technologies.