IMPLEMENTATION OF SLR(1) and LL(1) PARSER

Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri Campus, India
Kollam 690525
Kerala

Abhiram Prasad
AM.EN.U4AIE19001
abhiramprasad@am.students.amrita.edu

Nithin Sylesh
AM.EN.U4AIE19044
nithinsylesh@am.students.amrita.edu

Ritika R Prasad
AM.EN.U4AIE19053
ritikarprasad@am.students.amrita.edu

Vysakh S Nair
AM.EN.U4AIE19072
vysakhsnair@am.students.amrita.edu

Lakshmi G Pillai
AM.EN.U4AIE19074
lakshmigpillai@am.students.amrita.edu

Abstract*—The LL(1) ,SLR(1) parser is a useful formal syntax parser.The project includes the implementation of LL(1) SLR(1) and , two of the most essential principles in compiler design.This Paper demonstrates how one of the key abstract principles taught in a compiler development course may be applied.Compiler is the heart of the programming language so the therefore if we enhance the compilers; we make the execution more efficient.In this paper we present the SLR and LL(1) Parser model which is the major stage of the compiler phases as it is responsible for grammatical inspection of program statements and takes longer than the other stages

***Index Terms*— SLR(1), LL(1)**

I. INTRODUCTION

A compiler is a software that translates source code into machine code for a specific target. There are six steps in the compiler. These phases are lexical analysis,syntax analysis, semantic analysis, intermediate code generation, code optimization, and final code generation, and they all take place in the same order.At first the source program will be read character by character and tokens will be created.At second phase the tokens generated by the lexical analysis are utilised to check for syntax errors , and a syntax tree is built.In the next steps the source program is analyzed for conceptual problems, and comparable code is created and optimised so that the execution code can execute more quickly at run-time.

This paper focuses on the SLR(1) parser, which is a type of LR(0) that helps with syntax parsing.It is the smallest grammatical phase, with the fewest number of states. The main difference between SLR and LR parser is in the parsing table,there is a possibility of 'shift reduce' conflict because we are entering 'reduce', which corresponds to all terminal states.

We may fix this difficulty by entering 'reduce' in the ending state, which corresponds to the follow of the Left hand side of production. This is known as SLR(1) piece collection.A SLR(1) parser is made up of three important components: an SLR(1) parsing list, a stack, and an SLR(1) parsing program.

LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL(1) parsing are input string, a stack, parsing table for given grammar, and parser. Here, we discuss a parser that determines that given string can be generated from a given grammar(or

parsing table) or not.LL(*) and LL(*) are two types of nomenclature outlier parsers (finite). If a parser uses the LL(*)/LL(finite) parsing strategy, it is called LL(*)/LL(finite). The LL(*) and LL(finite) parsers are more functionally similar to PEG parsers. An LL(finite) parser can parse any LL(k) grammar with the fewest look-ahead and look-ahead comparisons possible. Because of the use of syntactic and semantic predicates, the class of grammars parsable by the LL(*) strategy includes some context-sensitive languages that have yet to be identified. LL(*) parsers, contrary to popular belief, are not LLR in general, and are built to perform worse on average (super-linear against linear time) and far worse in the worst-case scenario.

II. THEORY

LR(0): LR parser are non-recursive ,shift reduce,bottom up parser.Its also known as LR(K) parser .The full form of LR is as follows

L—-Left to right scanning of input stream

R—Construction of rightmost derivation in reverse

The simplest approach in the LR family is LR(0).The zero, which means no look ahead tokens, is the basic constraint of LR(0) are employed. It is a suffocating limitation to have to make assumptions based only on what has already occurred being read without even looking at what follows in the input If we could get a look at the next token and incorporate it into our decision-making, we will discover that it provides for a significantly bigger class of grammars to be parsed.

SLR(1):parser has the same table structure and parser operation as LR(0) parser, as well as the same LR(0) configuring sets. The distinction occurs when it comes to allocating table actions, when we will use one token of look ahead to assist arbitrate among the problems.The reduction activities that generate the anguish were the kind of conflicts observed in LR(0) parsing. In an LR(0) parser, a state can only have one reduction action and cannot have both shift and reduce instructions. Because a decrease is signalled for every completed thing, each completed item must be in a condition by itself. SLR(1) improves on the basic LR(0) parser by reducing only if the next input word is a member of the following set of the non-terminal being reduced. We don’t assume a reduce on all inputs while filling up the table, as we did in LR(0); instead, we choose the reduction only when the next input symbol is a member of the following set.The SLR(1) parser allows for both shift and reduce items in the same state, as well as multiple reduce items. As long as the following sets are disjoint, the SLR(1) parser will be able to identify which action to take.

The simplest form of context-free parsing is LL(1) parsing. It top-down parses a document and generates a left-derivation tree from the output. It's almost identical to recursive-descent parsing in terms of functionality.

The main advantage it has over recursive-descent parsing is that generating an LL(1) table is slightly easier for a machine than generating recursive functions. Tables are generally faster than a slew of recursive methods in practice. Hand-written parsers benefit from recursive-descent. The performance of each varies due to different advantages and disadvantages (they're more efficient in some areas than others), but in practice, the differences in relative performance are minor.

III. METHODOLOGY

A. Preliminary Concepts

A parse program in the LR parser examines the input text in the form of tokens and produces the output using a stack and parse table, updating the stack. To build a parse table, we should first construct a transition diagram. The parse table is divided into two parts: the action table and the goto table. Some directives in cells of the action table assist the parsing procedure. It is not required to describe the LR parser algorithm for our purposes. It is necessary to state that the first requirement for creating SLR(1) parser output is that the input sequence must have the format of SLR(1) grammar and the parser must identify this characteristic. To

accomplish this goal in the traditional manner, a transition diagram and a parse table must be created.If there is no conflict after the preceding stages, we can examine the grammar in SLR(1) parsing; else, it is not permitted.

Rules for LR and SLR Parsing

- As the initial closed set, the first item from the specified grammatical rules is added.
- If an entity is contained in the closure of the type E-> a.b.c and the symbol following it is non-terminal, add the symbol's production rules where the dot before the first item.
- Continue with this step until no more items can be added to the closure (I).

B. Construct a Transition Diagram Of SLR(1)and LR(0)

To create a transition diagram, we must first proceed as follows:

- To the grammar, add a rule in the form of S'->S.
- Begin constructing the diagram with the S0 state and the S'->yS item, then add the closure of this item to the S0 state.
- If you have elements in the form of A'->y.xB, establish a new state named Sj, join Si to Sj using a connection labelled x, and place these elements in the new state.Then, to this form, apply the closure of Sj elements.If Sj is comparable to a present situation, such as Sk, Sj does not need to be created, but Si must link to Sk via a connection labelled x. Continue this procedure until there are no more states that can be added to the diagram.

C. Constructing Parse Table

Following the completion of a grammar's SLR(1) diagram, we finish its parse table in the following manner.To complete the action table, you must complete the following steps:

- Place the shift j command in action[i,a] if there is a

transition from state I to j with input a. If an item in the form of A->y exists in state I place a reduce n command in action[i,b] cells for each b that belongs to follow(A), where n is the number of A->a.rules.
- Whether there is an S'->Sy item in state I enter the accept command into the action field.
- To indicate an error, use a symbol in the empty cells of the action table.

If there is a transition from state I to state j with non-terminal A, enter j in goto[i,A] cell.

IV. CONSTRUCTION OF SLR(1) PARSER AND LL(1) PARSER

A SLR(1) parser is made up of three important components: an SLR(1) parsing list, a stack, and an SLR(1) parsing program.As an example, the syntax of arithmetic expression will be chosen.The SLR(1) parser will be built for the example arithmetic expression grammar .G[E] is the abbreviated form the syntax of arithmetic expression is listed below.

E-> E +T ‖ *T*

T-> T*F ‖ *F*

F-> (E) ‖ *id*

FIRST / FOLLOW table		
Nonterminal	FIRST	FOLLOW
E'	{(,id}	{$}
E	{(,id}	{$,+,)}
T	{(,id}	{$,+,*,)}
F	{(,id}	{$,+,*,)}

Fig. 1. Example of a figure caption.

A. Construction of SLR(1) parsing list

First add augmented production for the given grammar

E'->E

E-> E +T

E->T

T->T*F

T->F

F->(E)

F->id

B. Create Canonical collection of LR (0) items

E'->.E

E-> .E +T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id
E'->.E,E–>.E+T,E ->.T,T ->.T*F ,T->.F,F->.(E),
F->.id
E'->.E,E'->E.+T I2:
E'->T. , T'->T.*F I3:
T->F.
F->(.E),E->.E+T ,E->.T ,T ->.T*F ,T->.F,F ->.(E),
F->.id
F->id
E->E+.T, T->.T*F, T->.F,F->.(E),F->.id
T->T*.F ,F->.(E) ,F->.id
F->(E.),E -> E.+T
I9: E->E+T.,T->T.*F
I10: T->T*F.
F->(E).

C. Parsing Table

LR table										
State	ACTION						GOTO			
	+	*	(	)	id	$	E'	E	T	F
0			s4		s5			1	2	3
1	s6					acc				
2	r_2	s7		r_2		r_2				
3	r_4	r_4		r_4		r_4				
4			s4		s5			8	2	3
5	r_6	r_6		r_6		r_6				
6			s4		s5				9	3
7			s4		s5					10
8	s6			s11						
9	r_1	s7		r_1		r_1				
10	r_3	r_3		r_3		r_3				
11	r_5	r_5		r_5		r_5				

Fig. 2. Parsing Table

V. LL(1)

A. Concepts Of LL(1)

An LL Parser is a program that accepts LL grammar. LL grammar is a subset of context-free grammar with some limitations in order to achieve a simplified version that is easy to implement. Both recursive-descent and table-driven algorithms can be used to implement LL grammar.

The LL parser is abbreviated as LL (k). The first L in LL(k) represents left-to-right parsing, the second L represents left-most derivation, and k represents the number of look aheads. Because k = 1 in most cases, LL(k) can also be written as LL (1)

Because the size of the table grows exponentially with the value of k, we can stick to deterministic LL(1) for a parser explanation.

B. Construction of LL(1) parser

Consider the following grammar:

E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> int

The grammar is also left-recursive, which as mentioned earlier, will not do. We're going to manually refactor it and eliminate the left recursion.

E -> T E'
E' -> + T E'
E' ->
T -> F T'
T' -> * F T'
T' ->

F -> (E)
F -> int

Above is the modified grammar we'll be using and referring to so keep that in mind.

Now we can move on to computing the FIRSTS sets

FIRSTS(E) = { (, int }

FIRSTS(E') = { +, nil }
FIRSTS(T) = { (, int }
FIRSTS(T') = { *, nil }

FOLLOWS() lets us know the terminals that can come after a non-terminal. This is not the final terminal in a non-terminal. It's the ones that can follow it.
and finally, the one we already did:

FIRSTS(F) = { (, int }
FOLLOWS(E) = { #EOS,) }
FOLLOWS(E') = { #EOS,) }
FOLLOWS(T) = { +, #EOS,) }
FOLLOWS(T') = { +, #EOS,) }
FOLLOWS(F) = { *, +, #EOS,) }

We're done with that step.
Now finally, something easy. Let's construct our parse table:Now we can move on to computing the FIRSTS sets (or more generally, the PREDICT sets, but we'll be looking at FIRSTS in particular)As before, FIRSTS are the first terminals that can appear for any non-terminal, while a PREDICT is the same but with the rule that each of the FIRSTS came from.
For each row, we get the PREDICT for that row, which is the set of terminals and the rule associated with the row's non-terminal. Let's do E:
remember FIRSTS(E) = { (, int }

C. Parsing Table

	+	*	(	)	int	#EOS
E			E-> T E'		E-> T E'	
E'	E' -> + T E'			E' ->		E' ->
T			T-> F T'		T-> F T'	
T'	T'->	T'-> * F T'		T'->		T'->
F			F-> (E)		F-> int	

VI. RESULT

SLR(1)

```
Enter Terminals (|) : +|(|)|*|#
No. of Non - Terminals  : 3
NonTerminals : E
Productions (|) : E+T|T
NonTerminals : T
Productions (|) : T*F|F
NonTerminals : F
Productions (|) : (E)|#
Start Symbol :  E
Productions :
E --> E+T | T |
T --> T*F | F |
F --> (E) | # |
```

```
Grammar Rule                First                       Follow
E                           ['+', '#', '(', '*']              ['+', '$', ')']
T                           ['*', '(', '#']           ['+', '$', ')', '*']
F                           ['(', '#']                ['+', '$', ')', '*']
Parse Table
            +       (       )       *       #       $       E       T       F
0           -       s1      -       -       s2      -       g3      g4      g5
1           -       s1      -       -       s2      -       g6      g4      g5
2           r5      -       r5      r5      -       r5      -       -       -
3           s7      -       -       -       -       a       -       -       -
4           r1      -       r1      s8      -       r1      -       -       -
5           r3      -       r3      r3      -       r3      -       -       -
6           s7      -       s9      -       -       -       -       -       -
7           -       s1      -       -       s2      -       -       g10     g5
8           -       s1      -       -       s2      -       -       -       g11
9           r4      -       r4      r4      -       r4      -       -       -
10          r0      -       r0      s8      -       r0      -       -       -
11          r2      -       r2      r2      -       r2      -       -       -
String : #+#*#
stack [0]               #+#*#$          0
stack [2, #, 0]         #+#*#$          1
stack [5, F, 0]         #+#*#$          1
stack [4, T, 0]         #+#*#$          1
stack [3, E, 0]         #+#*#$          1
stack [7, +, 3, E, 0]           #+#*#$          2
stack [2, #, 7, +, 3, E, 0]             #+#*#$          3
stack [5, F, 7, +, 3, E, 0]             #+#*#$          3
stack [10, T, 7, +, 3, E, 0]            #+#*#$          3
stack [8, *, 10, T, 7, +, 3, E, 0]                      #+#*#$          4
stack [2, #, 8, *, 10, T, 7, +, 3, E, 0]                        #+#*#$          5
stack [11, F, 8, *, 10, T, 7, +, 3, E, 0]                       #+#*#$          5
stack [10, T, 7, +, 3, E, 0]            #+#*#$          5
stack [3, E, 0]         #+#*#$          5
accepted
```

LL(1)

```
Enter Terminals (,) : +,-,*,/,(,),#,
No. of Non - Terminals  : 5
NonTerminals : E
Productions (,) : TP
NonTerminals : P
Productions (,) : +TP,-TP,@
NonTerminals : T
Productions (,) : FQ
NonTerminals : Q
Productions (,) : *FQ,/FQ,@
NonTerminals : F
Productions (,) : (E),#
Start Symbol :  E
Productions :
E --> TP |
P --> +TP | -TP | @ |
T --> FQ |
Q --> *FQ | /FQ | @ |
F --> (E) | # |

After Left Recurions Productions :
E --> TP |
P --> +TP | -TP | @ |
T --> FQ |
Q --> *FQ | /FQ | @ |
F --> (E) | # |
```

Grammar Rule	First	Follow
E	['#', '(']	['$', ')']
P	['+', '-', '@']	['$', ')']
T	['(', '#']	['$', ')', '+', '-']
Q	['@', '/', '*']	['$', ')', '+', '-']
F	['(', '#']	['*', '-', '$', ')', '/', '+

```
{'+': 1, '-': 2, '*': 3, '/': 4, '(': 5, ')': 6, '#': 7, '@': 8, '$': 9}
 {'E': 1, 'P': 2, 'T': 3, 'Q': 4, 'F': 5}
```

	+	-	*	/	(	)	$
@							
E	0	0	0	0	ETP	0	0
P	P+TP	P-TP	0	0	0	P@	P@
T	0	0	0	0	TFQ	0	0
Q	Q@	Q@	Q*FQ	Q/FQ	0	Q@	Q@
F	0	0	0	0	F(E)	0	0

```
String to Parse :  #+#*#$
stack [E, $]  Exp :  #
stack [T, P, $]  Exp :  #
stack [F, Q, P, $]  Exp :  #
stack [#, Q, P, $]  Exp :  #
stack [Q, P, $]  Exp :  +
stack [@, P, $]  Exp :  +
stack [P, $]  Exp :  +
stack [+, T, P, $]  Exp :  +
stack [T, P, $]  Exp :  #
stack [F, Q, P, $]  Exp :  #
stack [#, Q, P, $]  Exp :  #
stack [Q, P, $]  Exp :  *
stack [*, F, Q, P, $]  Exp :  *
stack [F, Q, P, $]  Exp :  #
stack [#, Q, P, $]  Exp :  #
stack [Q, P, $]  Exp :  $
stack [@, P, $]  Exp :  $
stack [P, $]  Exp :  $
stack [@, $]  Exp :  $
stack [$]  Exp :  $
Successfully Parsed
```

VII. CONCLUSION

We have successfully implemented SLR(1) parser and LL(1) parser. We passed many different test cases through our models and have got the desired results . All the errors we found out have been successfully rectified. We have observed the fact that The LL (1) parsing table generates more states than the SLR (1) parsing table

ACKNOWLEDGMENT

First of all we would like to thank god almighty for showering his blessings upon us. Next we would like to thank our professor Nisha for her constant guidance and support.

REFERENCES

1) Understanding the bottom-up SLR parser Authors: Sami Khuri, Jason Williams Authors Info & Claims ACM SIGCSE BulletinVolume 26 Issue 1March 1994 pp 339–343https://doi.org/10.1145/191033.191163ences

2) David A. Rosenblueth, Julio C. Peralta, SLR inference: An inference system for fixed-mode logic programs, based on SLR parsing, The Journal of Logic Programming, Volume 34, Issue 3,1998,Pages 227-259, ISSN 0743-1066, https://doi.org/10.1016/S0743-1066(96)000

3) LL parsing, LR parsing, complexity, and automata Author: R. Gregory Taylor Authors Info & ACM SIGCSE BulletinVolume 34Issue 4December 2002pp 71–75https://doi.org/10.1145/820127.820170

4) Tremblay, J.P. and Sorenson, P.G., 1985. *Theory and Practice of Compiler Writing*. McGraw-Hill, Inc..

5) Aho, A.V., 1986. Compilers principles. *Techniques, and Tools*.

6) Nieraj Singh, Celina Gibbs, and Yvonne Coady. 2007. C-CLR: a tool for navigating highly configurable system software. In Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software (ACP4IS '07). Association for Computing Machinery, New York, NY, USA, 9–es. DOI:https://doi.org/10.1145/1233901.1233910

7) Xiaoqing Wu, Barrett R. Bryant, Jeff Gray, Marjan Mernik,Component-based LR parsing,Computer Languages, Systems & Structures,Volume 36, Issue 1,2010,Pages 16-33,ISSN 1477-8424, https://doi.org/10.1016/j.cl.2009.01.002. (https://www.sciencedirect.com/science/article/pii/S1477842409000037)

8) A New Probabilistic LR Parsing Virach Sornlertlamvanich, Kentaro Inui, Hozumi Tanaka and Takenobu Tokunaga Department of Computer Science, Tokyo Institute of Technology fvirach,inui,tanaka,takeg@cs.titech.ac.jp

9) @misc{wu2008conquer, title={Conquer Compiler Complexity}, author={Wu, Xiaoqing}, year={2008}, publisher={VDM Verlag}

10) https://www.geeksforgeeks.org/clr-parser-with-examples/