**Experiment No - 06**

**Aim**: Implement a program for constructing

1. Recursive Decent Parser (RDP)
2. LALR Parser

# Date:

## Competency and Practical Skills:

* Understanding of RDP and bottom up parsers and its role in compiler construction
* Ability to write acceptance of string through RDP and parsing of string using LALR parsers for a given grammar
* Ability to develop RDP and LALR parser using bottom up approach

## Relevant CO: CO2

## Objectives:

By the end of this experiment, the students should be able to:

* Understand the RDP ,broad classification of bottom up parsers and its significance in compiler construction
* Verifying whether the string is accepted for RDP, a given grammar is parsed using LR parsers.
* Implement a RDP and LALR parser

## Software/Equipment: C compiler Theory:

## ❖ Recursive Descent Parser:

Recursive Descent Parser uses the technique of Top-Down Parsing without backtracking. It can be defined as a Parser that uses the various recursive procedure to process the input string with no backtracking. It can be simply performed using a Recursive language. The first symbol of the string of R.H.S of production will uniquely determine the correct alternative to choose.

The major approach of recursive-descent parsing is to relate each non-terminal with a procedure. The objective of each procedure is to read a sequence of input characters that can be produced by the corresponding non-terminal, and return a pointer to the root of the parse tree for the nonterminal. The structure of the procedure is prescribed by the productions for the equivalent nonterminal.

The recursive procedures can be simply to write and adequately effective if written in a language that executes the procedure call effectively. There is a procedure for each non-terminal in the grammar. It can consider a global variable lookahead, holding the current input token and a procedure match (Expected Token) is the action of recognizing the next token in the parsing process and advancing the input stream pointer, such that lookahead points to the next token to be parsed.

Match () is effectively a call to the lexical analyzer to get the next token.

For example, input stream is a + b$.

lookahead == a

match() lookahead == + match () lookahead == b

……………………….

……………………….

In this manner, parsing can be done.

## ❖ LALR (1) Parsing:

LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table. Example S → AA

A → aA

A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

S` → •S, $

S → •AA, $

A → •aA, a/b A → •b, a/b I0 State:

Add Augment production to the I0 State and Compute the ClosureL

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the nonterminal. So, the I0 State becomes

I0 = S` → •S, $

S → •AA, $

Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $

S → •AA, $

A → •aA, a/b

A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $

I2= Go to (I0, A) = closure ( S → A•A, $ )

Add all productions starting with A in I2 State because "•" is followed by the nonterminal. So, the I2 State becomes

I2= S → A•A, $ A → •aA, $

A → •b, $

I3= Go to (I0, a) = Closure ( A → a•A, a/b )

Add all productions starting with A in I3 State because "•" is followed by the nonterminal. So, the I3 State becomes

I3= A → a•A, a/b

A → •aA, a/b

A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)

Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure ( A → b•, a/b) = A → b•, a/b

I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "•" is followed by the nonterminal. So, the I6 State becomes

I6 = A → a•A, $

A → •aA, $

A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)

Go to (I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $

I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b

I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead. I3 = { A → a•A, a/b

A → •aA, a/b

A → •b, a/b

}

I6= { A → a•A, $

A → •aA, $

A → •b, $

}

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { A → a•A, a/b/$

A → •aA, a/b/$

A → •b, a/b/$

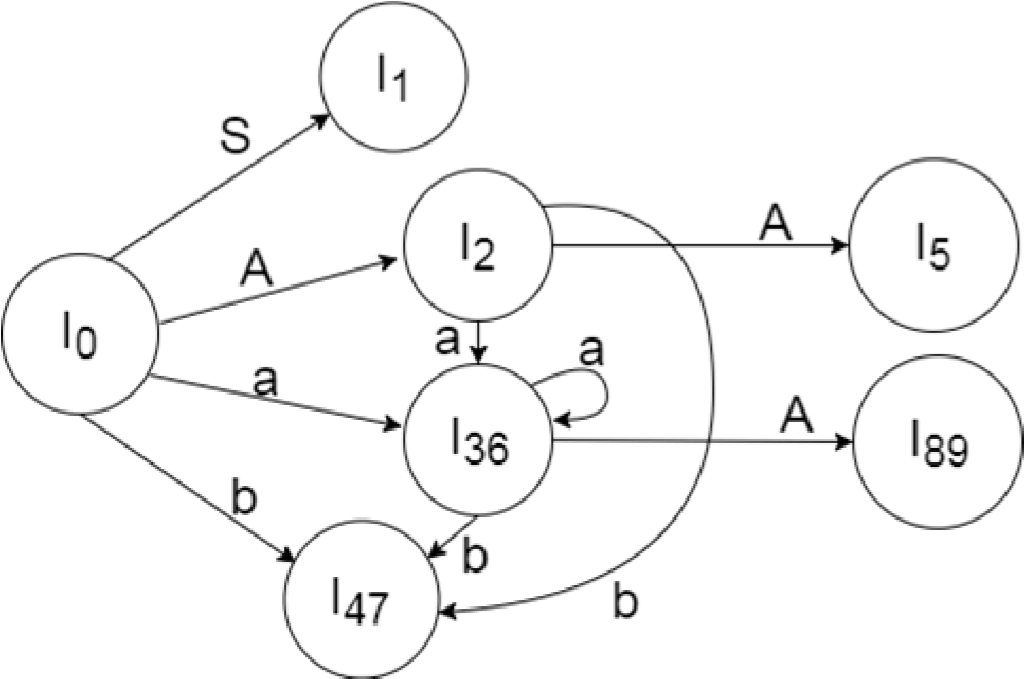
}

The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

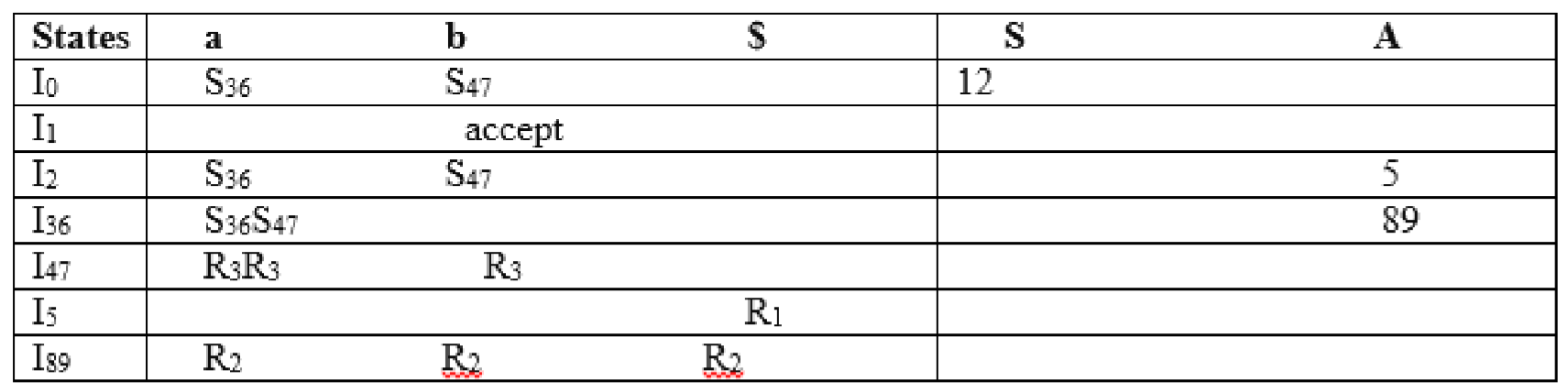
I47 = {A → b•, a/b/$}

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = {A → aA•, a/b/$} Drawing DFA:



LALR (1) Parsing table:



## Program-1:

#include<stdio.h>

#include<string.h>

#include<ctype.h>

char input[10]; int i,error; void E(); void T(); void Eprime(); void Tprime(); void F(); main()

{

i=0; error=0;

printf("Enter an arithmetic expression : "); // Eg: a+a\*a gets(input); E();

if(strlen(input)==i&&error==0) printf("\nAccepted..!!!\n");

else printf("\nRejected..!!!\n");

}

void E() {

T();

Eprime();

}

void Eprime()

{

if(input[i]=='+')

{ i++;

T();

Eprime();

}

}

void T()

{

F();

Tprime();

}

void Tprime()

{

if(input[i]=='\*')

{ i++;

F();

Tprime();

} }

void F()

{

if(isalnum(input[i]))i++;

else if(input[i]=='(')

{ i++; E();

if(input[i]==')')

i++;

else error=1;

}

else error=1; }

## Observations and Conclusion:

***Program -1:***

a+(a\*a), a+a\*a, (a), a , a+a+a\*a+a.... etc are accepted ++a, a\*\*\*a, +a, a\*, ((a . . . etc are rejected.

In the above output, as pe the grammar provided and as per calling procedure , the tree is parsed and thereby the inputted strings are mapped w.r.t calling procedure ; and the string/s which are successfully parsed are accepted and others rejected.

## Quiz:

1. What do you mean by shift reduce parsing?
2. Provide broad classification of LR parsers.
3. Differentiate RDP and LALR parser.
4. How to do merging of itemsets?

## Suggested Reference:

1. Introduction to Automata Theory, Languages and Computation by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman.
2. Geeks for geeks: <https://www.geeksforgeeks.org/recursive-descent-parser/>
3. <https://www.youtube.com/watch?v=odoHgcoombw>
4. https://www.geeksforgeeks.org/lalr-parser-with-examples/

## References used by the students:

## Rubric wise marks obtained:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Rubrics** | **Knowledge of Parsing**  **techniques**  **(2)** | | **Problem implementati**  **on (2)** | | **Code**  **Quality (2)** | | **Completeness and accuracy**  **(2)** | | **Presentation**  **(2)** | | **Total** |
| **Good**  **(2)** | **Avg.**  **(1)** | **Good**  **(2)** | **Avg.**  **(1)** | **Good**  **(2)** | **Avg.**  **(1)** | **Good**  **(2)** | **Avg.**  **(1)** | **Good**  **(2)** | **Avg.**  **(1)** |
| **Marks** |  |  |  |  |  |  |  |  |  |  |  |