Compiler Design 19115045

Term Paper Kunal Sachdeva

5th Semester CSE

Topic

Design a system that takes two things as input a regular expression and a string. Returns whether or not the string can be parsed from the regular expression.

a. Computation of ε-NFA from regular expression.

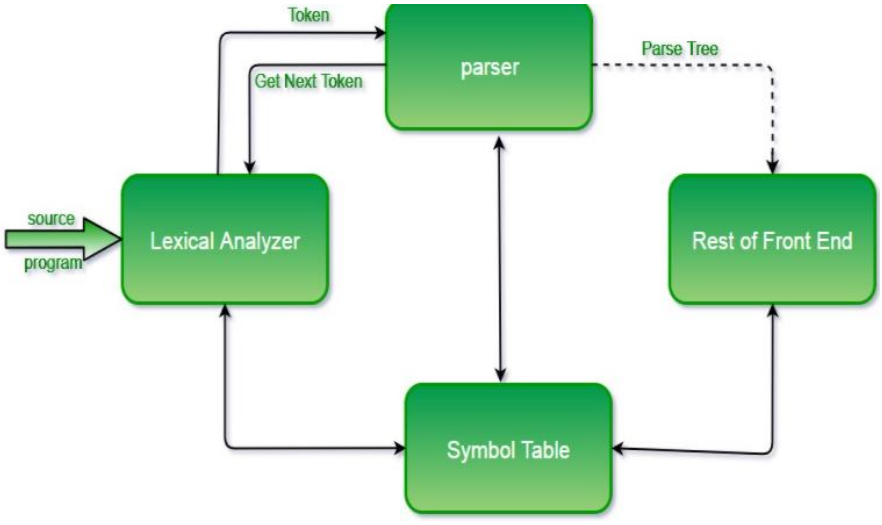
b. Simulation of ε-NFA on the given string.

Abstract

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase. A parser takes input in the form of sequence of tokens and produces output in the form of parse tree. Here I have discussed parsers and used implementation of LL(1) parser to check if a given string can be parsed by the regular expression.

Role of the parser

In the syntax analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. This is done by a parser. The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source



language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

Parser is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer.

Context Free Grammars: The syntax of a programming language is described by a context free grammar (CFG). CFG consists of set of terminals, set of non-terminals, a start symbol and set of productions.

Notation – ? ? ? where ? is a is a single variable [V]

? ? (V+T)*

Ambiguity: A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

Example- consider a grammar

S -> aS | Sa | a

Now for string aaa we will have 4 parse trees, hence ambiguous.

Removing Left Recursion : A grammar is left recursive if it has a non-terminal (variable) S such that there is a derivation

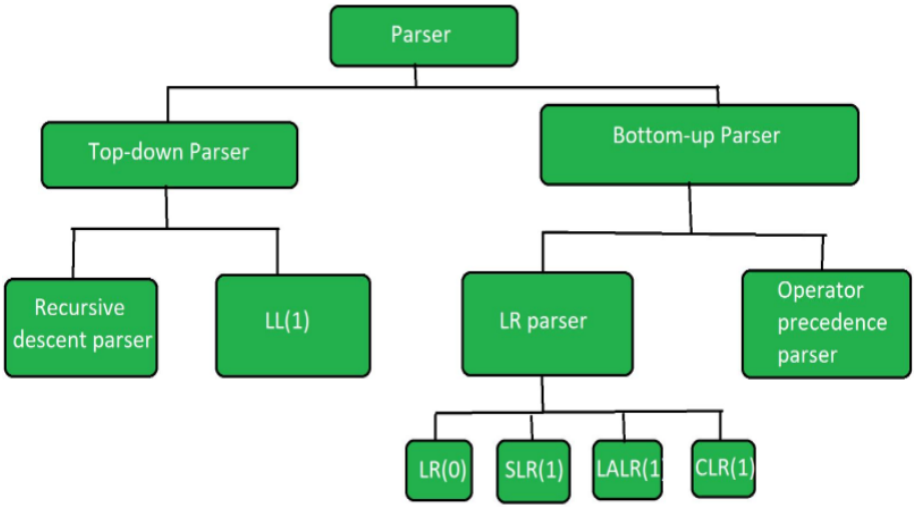
Removing Left Factoring : A grammar is said to be left factored when it is of the form – A -> αβ1 | αβ2 | αβ3 | …… | αβn | γ i.e. the productions start with the same terminal (or set of terminals). On seeing the input α we cannot immediately tell which production to choose to expand A.

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen to make the right choice.

The process of deriving the string from the given grammar is known as derivation (parsing).

Types of Parsers

Parser is mainly classified into 2 categories: Top-down Parser, and Bottom-up Parser. These are explained as following below.



1. Top-down Parser:

   Top-down parser is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e., it starts from the start symbol and ends on the terminals. It uses left most derivation.

   Further Top-down parser is classified into 2 types: Recursive descent parser, and non-recursive descent parser.

   a. Recursive descent parser: It is also known as Brute force parser or the with backtracking parser. It basically generates the parse tree by using brute force and backtracking.

   b. Non-recursive descent parser: It is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses parsing table to generate the parse tree instead of backtracking.

2. Bottom-up Parser:

   Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e., it starts from non-terminals and ends on the start symbol. It uses reverse of the right most derivation.

   Further Bottom-up parser is classified into 2 types: LR parser, and Operator precedence parser.

   a. LR parser: LR parser is the bottom-up parser which generates the parse tree for the given string by using unambiguous grammar. It follows reverse of right most derivation.

      LR parser is of 4 types:

      i. LR(0)
      ii. SLR(1)
      iii. LALR(1)
      iv. CLR(1)

   b. Operator precedence parser: It generates the parse tree form given grammar and string, but the only condition is two consecutive non-terminals and epsilon never appear in the right-hand side of any production.

LL(1) Parsing

Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Algorithm to construct LL(1) Parsing Table:

Step 1: First check for left recursion in the grammar, if there is left

recursion in the grammar remove that and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

First(): If there is a variable, and from that variable, if we try to drive all

the strings then the beginning Terminal Symbol is called the First.

Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

Find First(α) and for each terminal in First(α), make entry A –> α in the table.

If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.

If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.

To construct the parsing table, we have two functions:

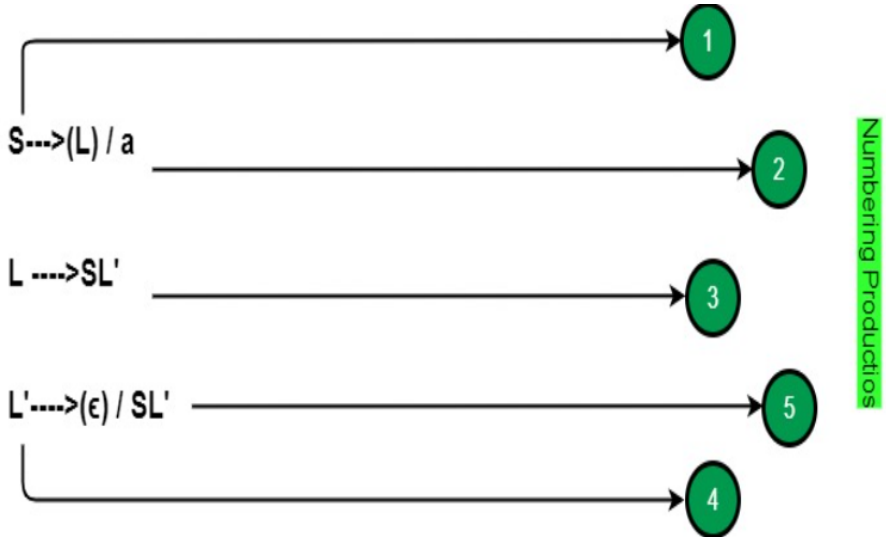
In the table, rows will contain the non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Example – consider the grammar

S -> (L) | a

L -> SL’

L’ -> ε | SL’



M	a	,	(	)	$
S	2		1		
L	3		3		
L’		5		4	

No multiple entries this is LL(1) Grammar

For any grammar if M have multiple entries than it is not LL(1) grammar

Example –

S -> iEtSS’/a

S’ ->eS/ε

E -> b

M	a	b	E	I	t	$
S	2			1		
S’			¾			4
E		5				

Multiple entries so it is not LL(1) Grammar

C program for constructing of LL (1) parsing

LOGIC: Read the input string. Using predictive parsing table parse the given input using stack . If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to $. C implementation of above logic is as follows:

Code:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

char s[20], stack[20];

void main()

{

char m[5][6][3]={"tb", " ", " ", "tb", " ", " ", " ", "+tb", " ", " ", "n", "n", "fc", " ", " ", "fc", " ", " ", " ", "n", "*fc", " a ", "n", "n", "i", " ", " ", "(e)", " ", " "};

int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
```

```
int i,j,k,n,str1,str2;
clrscr();
printf("\n Enter the input string: ");
scanf("%s",s);
strcat(s,"$");
n=strlen(s);
stack[0]='$';
stack[1]='e';
i=1;
j=0;
printf("\nStack     Input\n");
printf("_____\n");
while((stack[i]!='$')&&(s[j]!='$'))
{
if(stack[i]==s[j])
{
i--;
j++;
}
switch(stack[i])
{
case 'e': str1=0;
break;
case 'b': str1=1;
break;
case 't': str1=2;
break;
```

```
case 'c': str1=3;
break;
case 'f': str1=4;
break;
}
witch(s[j])
{
case 'i': str2=0;
break;
case '+': str2=1;
break;
case '*': str2=2;
break;
case '(': str2=3;
break;
case ')': str2=4;
break;
case '$': str2=5;
break;
}
if(m[str1][str2][0]=='\0')
{
printf("\nERROR");
exit(0);
}
else if(m[str1][str2][0]=='n')
i--;
```

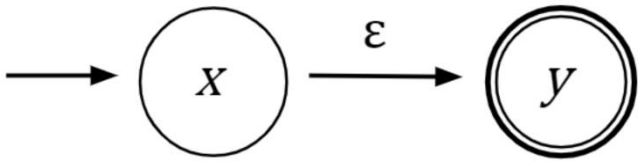
```
else if(m[str1][str2][0]=='i')
stack[i]='i';
else
{
for(k=size[str1][str2]-1;k>=0;k--)
{
stack[i]=m[str1][str2][k];
i++;
}
i--;
}
for(k=0;k<=i;k++)
printf(" %c",stack[k]);
printf("       ");
for(k=j;k<=n;k++)
printf("%c",s[k]);
printf(" \n ");
}
printf("\n SUCCESS");
getch();
}
```

Computation of ε-NFA from regular expression

To apply Thompson's Construction Algorithm to convert this expression into its respective NFA. Thompson's Construction Algorithm is a method for converting regular expressions to their respective NFA diagrams. We will use the rules which defined a regular expression as a basis for the construction:

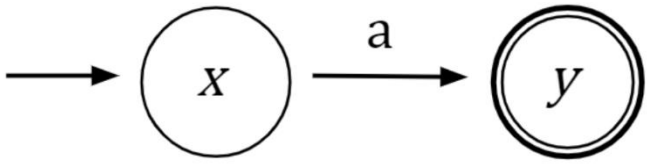
Rule #1: An Empty Expression

An empty expression, for example ‘’ or ε, will be converted to:



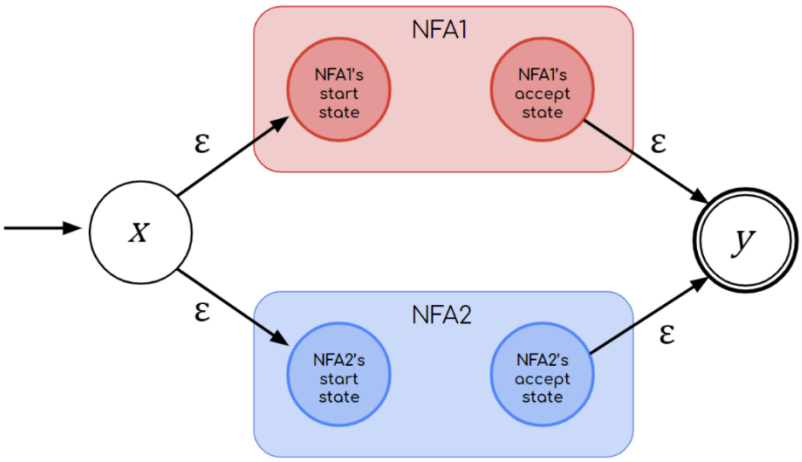
Rule #2: A symbol

A symbol, such as a or b, we will convert to:



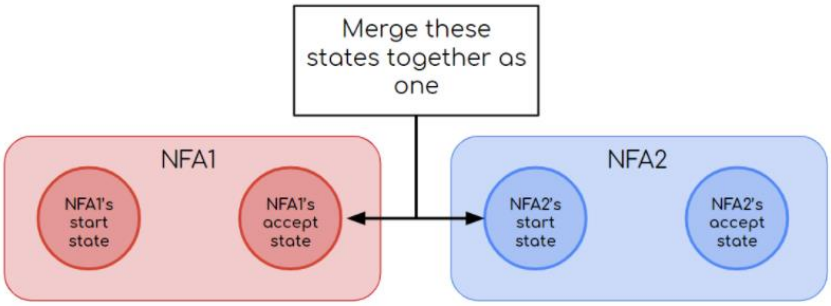
Rule #3: Union expression

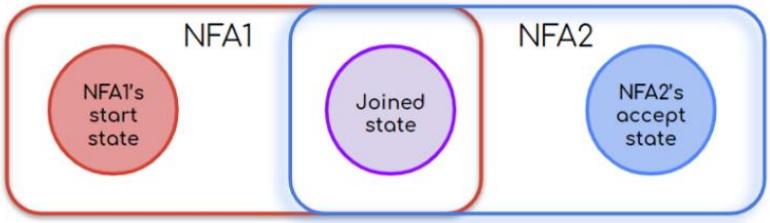
A union expression a + b will be converted to:



Rule #4: Concatenation expression

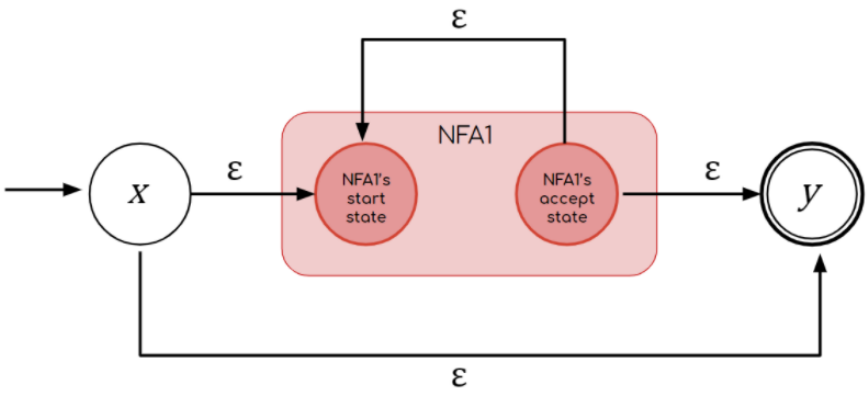
A concatenation expression ab or a?b will be converted to:





Rule #5: A closure/kleene star expression

A closure a* will be converted to:



Conclusion

Code is taking a user input string and using predictive parsing table parsing the given input using stack. Parsing table for LL(1) parser is being constructed and returns if that string can be parsed by the regular expression.

Bibliography

- https://www.geeksforgeeks.org/introduction-of-parsing-ambiguity-and-parsers-set-1/
- https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/#
- https://medium.com/swlh/visualizing-thompsons-construction-algorithm-for-nfas-step-by-step-f92ef378581b