

SPL-1 Project Report, 2019

Implementation of Predictive Parser

Course: Software Project Lab I

Course No: SE 305

Submitted by

Ahmed Al Sabbir

BSSE Roll No. : 1016

BSSE Session: 2017-18

Supervised by

Mohd.Zulfiquar Hafiz

Designation: *Professor*

Institute of Information Technology



Institute of Information Technology

University of Dhaka

29-05-2019

Table of Contents

1.Introduction	3
1.1.Background study	3-4
1.2.Challenges	4
2.Project Overview	5-12
3.User Manual	11-12
4.Conclusion	13
5.References	13

1.Introduction

“Predictive parser” is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking. To perform its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some restrictions on the grammar and allows only a class of grammar known as LL(k) grammar.

1.1Background Study

Predictive Parser

It is top – down parsing. This an efficient non-backtracking form of top-down parser called a predictive parser. LL(1) grammars from which predictive parsers can be constructed automatically.

To construct a predictive parser we had to learn

- Input symbol a.
- Non terminal A to be expanded.
- Alternatives of production $A \rightarrow a_1 | a_2 | \dots | a_n$.
- That derives a string beginning with a.
- Proper alternative must be detectable by looking at only first symbol it derives.
- Flow of control constructs most programming language with their keywords that are detected.

First and Follow Set

FIRST(X) gives us the set of terminals that can begin the strings derived from X. For eg. if there is a derivations $S \rightarrow X \rightarrow aX$, then a is in First(S).

If your grammar is

$S \rightarrow X | Y$

$X \rightarrow aX | a$

$Y \rightarrow bB | b$

Now if we want to have a derivation for bbb starting from S, since we know that b is not in FIRST(X) but is in FIRST(Y), the first production to apply is $S \rightarrow Y$.

FOLLOW(X) gives us the set of terminals that can follow X in a string derived in this grammar.

For eg. if there is a derivation $S \rightarrow Y \rightarrow XaY$, then a is in FOLLOW(X)

An example of first and follow is given below-

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \varepsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \varepsilon \\ F &\rightarrow 0 \mid 1 \mid (E) \end{aligned}$$

FIRST(E)	= {0,1,(}
FIRST(E')	= {+, ε}
FIRST(T)	= {0,1,(}
FIRST(T')	= {*, ε}
FIRST(F)	= {0,1,(}

FOLLOW(E)	= {\$,)}
FOLLOW(E')	= {\$,)}
FOLLOW(T)	= {+, \$,)}
FOLLOW(T')	= {+, \$,)}
FOLLOW(F)	= {*, +, \$,)}

Context Free Grammar

A context free grammar is defined by 4 tuples

V - Set of variables

T - Set of Terminals

P - Set of productions

S - Start symbol

1.2 Challenges

Implementing a new project is always challenging . The procedures may be overwhelming, difficult. For the implementation of this project there are lots of challenges that I have faced. Some of them are:

- ☐ Handling large code for the first time
- ☐ Learning and understanding new topics like predictive parser
 - ☐ Having a brief overview over context free grammar
- ☐ Concept of first and follow set
- ☐ Learning parsing table

2. Project Overview

I have divided my whole project into three different parts. They are

- ☐ Context free grammar
- ☐ First and Follow set
- ☐ Algorithm of parsing table
- ☐ Syntax analysis

First and Follow set

First and Follow sets are needed so that the parser can properly apply the needed production rule at the correct position.

First Function

$\text{First}(\alpha)$ is a set of terminal symbols that begin in strings derived from α .

Example-

Consider the production rule-

$A \rightarrow abc / def / ghi$

Then, we have-

$\text{First}(A) = \{ a, d, g \}$

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \epsilon$,

$$\text{First}(X) = \{ \epsilon \}$$

Rule-02:

For any terminal symbol 'a',

$$\text{First}(a) = \{ a \}$$

Rule-03:

For a production rule $X \rightarrow Y_1 Y_2 Y_3$,

Calculating First(X)

- If $\epsilon \notin \text{First}(Y_1)$, then $\text{First}(X) = \text{First}(Y_1)$
- If $\epsilon \in \text{First}(Y_1)$, then $\text{First}(X) = \{ \text{First}(Y_1) - \epsilon \} \cup \text{First}(Y_2 Y_3)$

Calculating First($Y_2 Y_3$)

- If $\epsilon \notin \text{First}(Y_2)$, then $\text{First}(Y_2 Y_3) = \text{First}(Y_2)$
- If $\epsilon \in \text{First}(Y_2)$, then $\text{First}(Y_2 Y_3) = \{ \text{First}(Y_2) - \epsilon \} \cup \text{First}(Y_3)$

Similarly, we can make expansion for any production rule $X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$.

Follow Function-

$\text{Follow}(\alpha)$ is a set of terminal symbols that appear immediately to the right of α .

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place \$ in Follow(S).

Rule-02:

For any production rule $A \rightarrow \alpha B$,

$$\text{Follow}(B) = \text{Follow}(A)$$

Rule-03:

For any production rule $A \rightarrow \alpha B \beta$,

- If $\epsilon \notin \text{First}(\beta)$, then $\text{Follow}(B) = \text{First}(\beta)$
- If $\epsilon \in \text{First}(\beta)$, then $\text{Follow}(B) = \{ \text{First}(\beta) - \epsilon \} \cup \text{Follow}(A)$

Important Notes-

Note-01:

- ϵ may appear in the first function of a non-terminal.
- ϵ will never appear in the follow function of a non-terminal.

Note-02:

- Before calculating the first and follow functions, eliminate left recursion from the grammar, if present
- We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

Example

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid id$

	First	Follow
F	{(,id}	{+, *,), \$}
T	{(,id}	{+,), \$}
E	{(,id}	{), \$}
E'	{+,ε}	{), \$}
T'	{*,ε}	{+,), \$}

Non - terminal	Input Symbol					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Sample Output of first and follow

```
First(E) = { (, i, }
```

```
First(R) = { +, #, }
```

```
First(T) = { (, i, }
```

```
First(Y) = { *, #, }
```

```
First(F) = { (, i, }
```

```
Follow(E) = { $, ), }
```

```
Follow(R) = { $, ), }
```

```
Follow(T) = { +, $, ), }
```

```
Follow(Y) = { +, $, ), }
```

```
Follow(F) = { *, +, $, ), }
```

```
Process returned -1073741819 (0xC0000005)   execution time : 30.030 s
Press any key to continue.
```


Context-Free Grammars

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

1. a set of terminal symbols, which are the characters of the alphabet that appear in the strings generated by the grammar.
2. a set of nonterminal symbols, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
3. a set of productions, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
4. a start symbol, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

To generate a string of terminal symbols from a CFG, we:

Begin with a string consisting of the start symbol;

Apply one of the productions with the start symbol on the left hand side, replacing the start symbol with the right hand side of the production;

Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols.

A CFG for Arithmetic Expressions

An example grammar that generates strings representing arithmetic expressions with the four operators +, -, *, /, and numbers as operands is:

$\langle \text{expression} \rangle \rightarrow \text{number}$

$\langle \text{expression} \rangle \rightarrow (\langle \text{expression} \rangle)$

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle + \langle \text{expression} \rangle$

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle - \langle \text{expression} \rangle$

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle * \langle \text{expression} \rangle$

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle / \langle \text{expression} \rangle$

Algorithm of Parsing table and Stack implementation

Construction of LL(1) Parsing Table:

To construct the Parsing table, we have two functions:

- 1: 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.
- 2: Follow(): What is the Terminal Symbol which follow a variable in the process of derivation.

Now, after computing the First and Follow set for each Non-Terminal symbol we have to construct the Parsing table. 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 First set.

Now, let's understand with an example.

Example

Non - terminal	Input Symbol					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

Stack	Input	Action
E\$)id*+id\$	Error, Skip)
E\$	id*+id\$	id is in First(E)
TE'\$	id*+id\$	
FT'E'\$	id*+id\$	
idT'E'\$	id*+id\$	
T'E'\$	*+id\$	
*FT'E'\$	*+id\$	
FT'E'\$	+id\$	Error, M[F,+]=synch
T'E'\$	+id\$	F has been popped

3.User Manual

Here, on the input side if we give 2 or more finite number of valid productions of a context free grammar, then we will get a parse table for those productions. The parse table will be created following first and follow algorithm. Then, we will take a string as an input in order to check whether that string is accepted by that grammar or not.

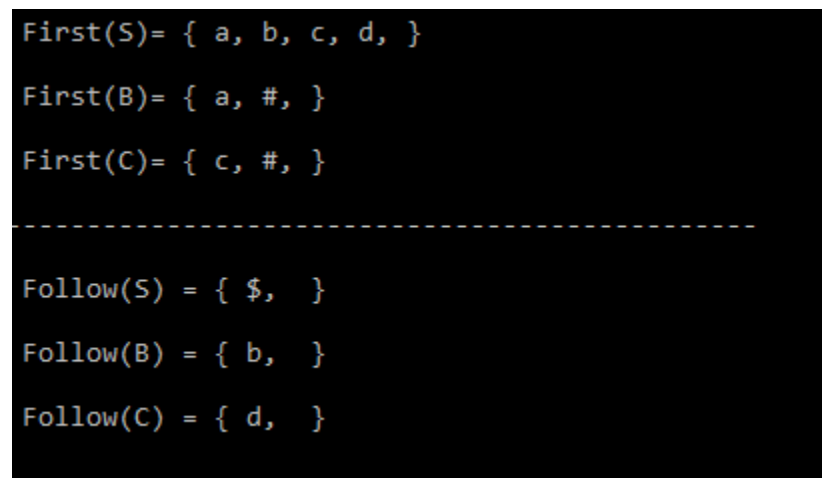
Sample Input

At first, the user will be considering any finite number of productions of a grammar:



```
C:\Users\sabbir\Desktop\Project\predictiveparser.exe
how many productions do you want to input?
6
Enter 6 productionss:
S=Bb
S=Cd
B=aB
B=#
C=cC
C=#
```

For this production as input, it'll show the first and follow set of the grammar followed by a parse table



```
First(S)= { a, b, c, d, }
First(B)= { a, #, }
First(C)= { c, #, }

-----

Follow(S) = { $, }
Follow(B) = { b, }
Follow(C) = { d, }
```

Here is our predictive parse table

Predictive Parsing Table for the above grammer :-						
		b	d	a	c	\$
S		S=Bb	S=Cd	S=Bb	S=Cd	
B		B=#		B=aB		
C			C=#		C=cC	

Now, the user will take a sample string to check to the grammar is accepting that string is accepted or not

Enter a string: aaaaab\$

Stack	Input	Action
\$S	aaaaab\$	S=Bb
\$bB	aaaaab\$	B=aB
\$bBa	aaaaab\$	pop action
\$bB	aaaaab\$	B=aB
\$bBa	aaaab\$	pop action
\$bB	aaab\$	B=aB
\$bBa	aaab\$	pop action
\$bB	aab\$	B=aB
\$bBa	aab\$	pop action
\$bB	ab\$	B=aB
\$bBa	ab\$	pop action
\$bB	b\$	B=#
\$b	b\$	pop action
\$	\$	pop action

the string is accepted

Conclusion

It helped me a lot to improve my coding skill and I have known how to handle large code for the first time. I hope it will guide me to deal with bigger problems in the coming days.

References

1. <https://bit.ly/2YMZ3On>, *Compilers: Principles, Techniques, and Tools by Ullman*, 26-05, 2019
2. <https://bit.ly/2JFumaf>, Compiler Design | Construction of Parsing Table, 27-05-2019
3. <https://bit.ly/1eYNIEj>, Context free grammar, Monday, 27-05-2019
4. <https://bit.ly/2X6Ij4g>, Top down parsing, 28-05-2019