KATHMANDU UNIVERSITY

SCHOOL OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REPORT ON
NON-RECURSIVE PREDICTIVE PARSER

Fourth Year First Semester Compiler Design Project Final Report submitted in partial fulfillment of the requirements for course COMP 409

Submitted By
Prajwol Shrestha [09]
Roshan Shrestha [22]
Shweta Shrestha [26]
Ajaya Thapa [28]

Submitted To
Mr. Sushil Nepal
Department of Computer Science and Engineering

Jan 26, 2014

TABLE OF CONTENTS

1. Introduction

The figure below is a compiler model that shows how a Parser is used to check tokens coming from the lexical analyzer, and verifies that the string can be generated by grammar from the source language. The parser is responsible to report any syntax error it encounters during parsing. It is also expected that the parser recovers from errors, those that occurs commonly so that it can continue processing the remainder of the input.

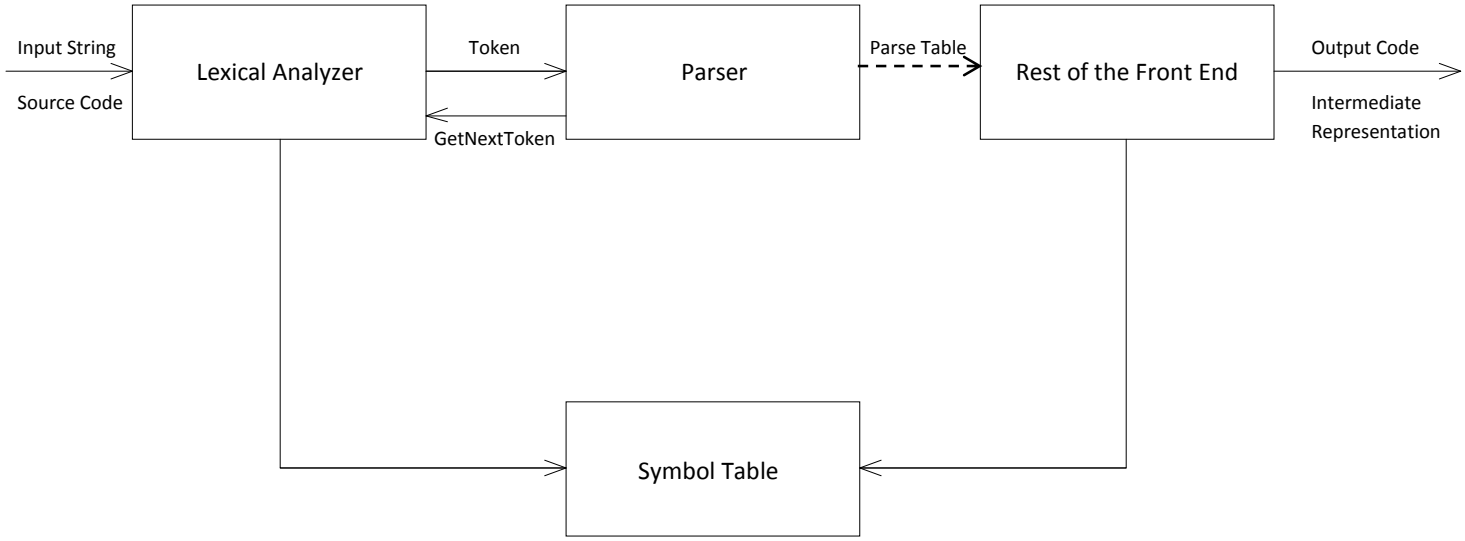

Fig. Parser Model

There are mainly three types of parser that can be discussed. They are:

1. Universal Parser: It can parse any kind of grammar. *Cocke-Younger-Kasami Algorithm* and Earley's Algorithm can be utilized to create universal parser.
2. Top-down Parser: As the name suggests, it builds parse tree from top (root) to the bottom (leaves). A parse tree is an ordered rooted tree that represents the syntactic structure of string according to some formal grammar. Examples of top-down parsers are: LL Grammar, Recursive Descent, **and Non-Recursive Predictive Parser**.
3. Bottom-up Parser: In this parser, the parse tree is built from the bottom (leaves) to the top (root). Examples of bottom-up parsers are: LR/SLR, CLR, LALR, Shift-Reduce, Operator Procedure Parser

In top-down and bottom-up parser, the parser scans the symbols from left to right, one symbol at a time.

1.1. Non-recursive predictive parsing

Non-recursive predictive parsing can be build by maintaining a stack explicitly. The key problem encountered during predictive parsing is to determine the production to be applied for a non terminal. The non-recursive parser looks up the production to be applied in a parsing table, as in the figure below.

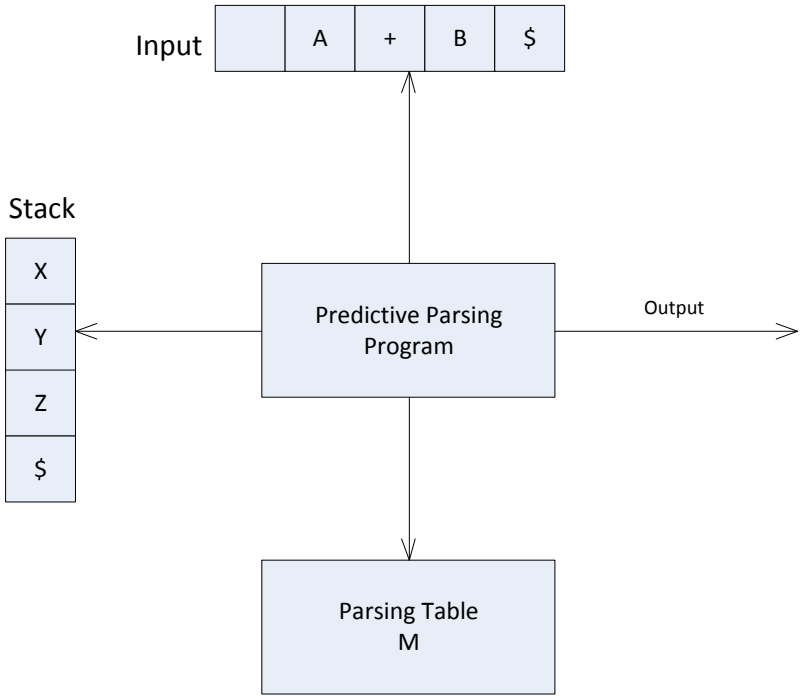

Fig. Non Recursive Predictive Parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output. The input buffer contains the string to be parsed, followed by $, which indicates the end of input string. The stack contains a sequence of grammar symbols with $ on the top. Initially, the stack contains the start symbol of the grammar on the top of $. The parsing table is a two-dimensional array m[A,a], where A is a non terminal and a is terminal or symbol $.

The parser program behaves as follows. The program considers X, the symbol in top of stack, and a, the current input symbol. These two symbols determine the action of the parser. There are following three possibilities:

1. If X=a=$, then parser halts and announces successful completion of parsing.
2. If X=a≠$, the parser pops X off the stack and advances the input pointer to next input symbol.

3. If X is a non-terminal the program consults entry M[X,a] of the parsing table M. this entry will be either an x-production of grammar or an error entry.

1.1.1 FIRST

The *first set* of a sequence of symbols u, written as FIRST(u) is the set of terminals which start the sequences of symbols derivable from u.

Rule 1: If X is a terminal then , FIRST(x)={x}

Rule 2: If X→aα, where a is a terminal and α can be either a terminal or non-terminal or both
Then, add a to FIRST(a) ie FIRST(X)={a}
And if, X->ε is a a production then
Add ε to FIRST(x) ie FIRST(X)={ ε }

Rule 3: If $X \rightarrow Y_1Y_2 \ldots Y_k$ is a production where all Y_i's are non-terminals
Then, $FIRST(X)=\{FIRST(Y_1)- \varepsilon \}$
If $FIRST(Y_1)$contains ε symbol then,
$FIRST(X)= \{FIRST(Y_1)- \varepsilon \}U\{FIRST(Y_2)- \varepsilon \}$
Again if $FIRST(Y_2)$contains ε symbol then,
$FIRST(x)= \{FIRST(Y_1)- \varepsilon \}U\{FIRST(Y_2)- \varepsilon \}U\{FIRST(Y_3)- \varepsilon \}$
…
This process continues to include non ε symbol of $FIRST(Y_i)$ to FIRST(X) till $FIRST(Y_i)$ doesn't contain ε

1.1.2 FOLLOW

The *follow set* of a nonterminal A is the set of terminal symbols that can appear immediately to the right of A in a valid sentence

---

1. Place EOF in Follow(S) where S is the start symbol and EOF is the input's right endmarker. The endmarker might be end of file, newline, or a special symbol, whatever is the expected end of input indication for this grammar. We will typically use $ as the endmarker.
2. For every production A→ αBβ where α and β are any string of grammar symbols (ie β≠ ε) and B is a nonterminal, everything in FIRST(β) except **ε** is placed in Follow(B) ie FOLLOW(B)={FIRST(β) - ε }
3. For every production A→αB, or a production A→αBβ where β=> ε (FIRST(β) = ε), then everything in FOLLOW(A) is added to FOLLOW(B) ie FOLLOW(B)={FOLLOW (A)}.

1.1.3 Algorithm to fill the parsing table

Input: Set of FIRST and FOLLOW elements along with Grammar G

Output: Parsing Table M

Method: We check for every production of the form A->alpha, the FIRST (α). If it contains ε then add the production to M[A,a]. Else add each element of FOLLOW(A) in M[A, b].

Algorithm:

```
Compute FIRST and FOLLOW for every non-terminals of the grammar
For every production A → α do
{
        For every non - ε member a in FIRST(α) do
                Table M[A, a] = A→ α
        If FIRST(α) contain ε then
                For every b in FOLLOW(A) do
                        Table M[A,b] = A→ α
}
```

1.1.4 Algorithm of non-recursive predictive parsing

Input: A string w and a parsing table M for grammar G.

Output: If w is in L(G), a leftmost derivation of w; otherwise and error indication.

Method: initially, the parser is in a configuration in which it has $S on the stack with S, the start symbol of G on top; and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown as:

```
Push start symbol into stack
Repeat
Begin
        Make X to be the top stack symbol and 'a'
        The next input symbol
        If  X is a terminal or $ then,
                If X=a then
                        POP X from stack and remove 'a' from input
                Else
                        Error()
        Else
                /*X is non terminal */
                If M[X,a] = X->Y1,Y2,Y3,…,Yk then
                Begin
```

```
                    POP X from stack
                    PUSH Yk,Yk-1…Y2,Y1 onto  stack so y is on top
            End
            Else
                    Error()
End until X=$/*until stack is empty */
```

The construction of a predictive parser is aided by two functions associated with a grammar G. these functions, FIRST and FOLLOW allows us to fill in the entries of a predictive parsing table for G.

2. Motivation:

The language chosen for the implementation of this parser is Java. Java is open sourced object oriented platform language where libraries of different classes are available easily. One of such classes includes StringBuffer, which can be effectively used to manipulate strings of variable length anytime. This is one of the important features that drag us to using Java, where variable length string can be programmed easily as is required for the parser. This motivated us to use JAVA as our programming language.

2.1 Tools required

The following methodology and tools have been applied for the Project Completion:
For developing the parser we used the following tools:

- **Eclipse**

  In this project Eclipse acts as an Integrated Development Environment (IDE). For the development of the project, whole programming part is done in this IDE. In this project, different libraries of Java are required along with the java compiler and a console window to run the program, so to do this in one go Eclipse IDE has been used. In this way Eclipse IDE is used.

- **JAVA**

  Java is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. The latest library made available in Eclipse that was used was JRE System Library 7.

3. Implementation:

The grammar given to design the parser was:

$$S \rightarrow ABC$$
$$A \rightarrow abA \mid ab$$
$$B \rightarrow b \mid BC$$
$$C \rightarrow c \mid cC$$

In order to generate a non-recursive parser, we had to remove any left recursion if was available using the following rule.

$$A \rightarrow A\alpha|\beta$$

Removing Left Recursion:

$$A \rightarrow \beta A'$$

$$A \rightarrow \alpha A'|\Sigma$$

Hence the production rule that was used in the program was modified to:

$$S \rightarrow ABC$$
$$A \rightarrow abA \mid ab$$
$$B \rightarrow bK$$
$$K \rightarrow KC|E$$
$$C \rightarrow c \mid cC$$

where E represented epsilon and K represented B'.

A single class NPParser is created which is used to define the production rule, calculate the FIRST and FOLLOW, determine and remove the ambiguity of the grammar, create the parse table and analyze the given string using the parser. Several modules have been created within the class that was utilized to create the parser. The following image shows lists of variables and methods used during implementation of the NPParser class.

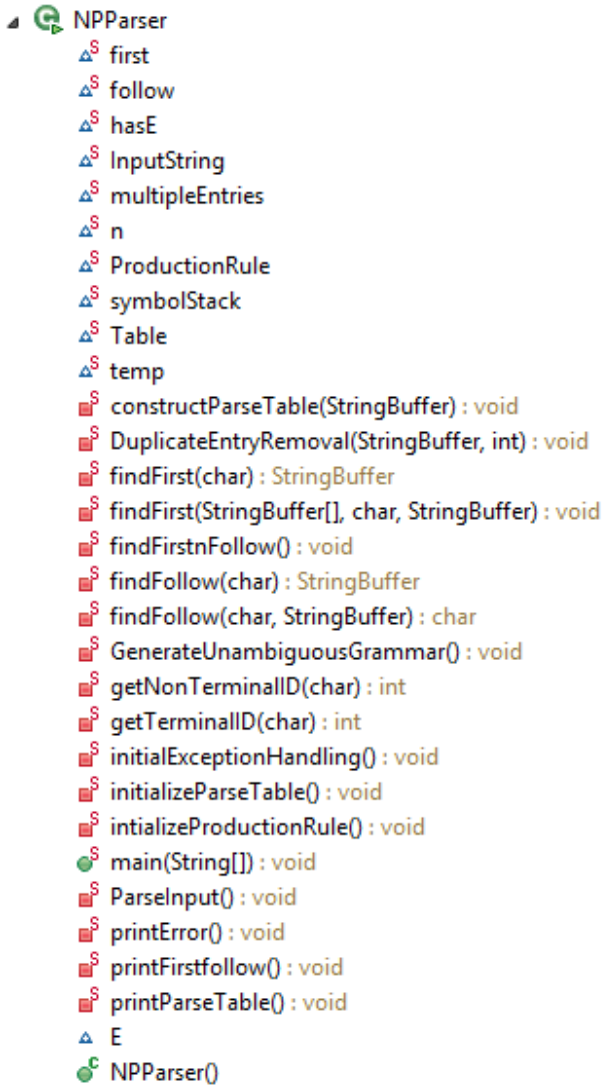

Fig. The NPParser class

Among the several modules of the NPParser class, the most important modules have been described below:

- findFirstnFollow(): The function is used to find the FIRST and FOLLOW. The function has been built based upon the algorithm to find FIRST and FOLLOW of each of the terminals and non-terminals. However, the first and follow of only the non-terminals have been stored in the variables first[5] and follow[5] of the five non-terminals S, A, B, K and C. Any duplicate entries in the FIRST and FOLLOW of the terminals have been removed using DuplicateEntryRemoval(StringBuffer, int) module.
- constructParseTable(StringBuffer): This module is used to construct the parse table using the algorithm after successfully finding the FIRST and FOLLOW of all the

non-terminals. This function also checks on the multiple entries if occurred on the table. When multiple entries occurred, a new unambiguous grammar was introduced. For the given grammar the following rule removed the ambiguity, which was done using the LL(1) grammar. The construction was done manually using GenerateUnambiguousGrammar().

S→ ABC
A→ ab
B→ bC
K→KC|E
C→cC

- ParseInput(): The given input buffer to be parsed was **abbcc.** Applying the algorithm, using the stack and the given input, it was checked if the grammar expressed in terms of the parsing table could parse the input. The input was valid for the given grammar.

4. Output:

Using the user defined modules; we can obtain the following result for the non-recursive predictive parser.

- **The Grammar**

```
Given Grammar Production Rule
S->ABC
A->abA|ab
B->bK
K->CK|E
C->c|cC
```

- **The table: A→α, FIRST(A), FOLLOW(A)**

```
The FIRST(A) and FOLLOW(A) of nonterminal A
+-----------------------------------+
A->X|Y      First(A)        Follow(A)
S->ABC        {a}              {$}
A->abA|ab     {a}              {b}
B->bK         {b}              {c}
K->CK|E       {c,E}             {c}
C->c|cC       {c}              {$,c, }
+-----------------------------------+
```

- **The parse table (With Ambiguity)**

```
The obtained Parse Table
+-------------------------------------------+
     a             b             c          $
S   S->ABC         -             S->E       -
A   A->abA,A->ab   -             -          -
B   -              B->bK         -          -
K   -              -             -          -
C   -              -             C->c,C->cC -
+-------------------------------------------+
```

- **The Unambiguous Grammar Production Rule (NEW)**

```
Given Grammar Production Rule
S->ABC
A->ab
B->bK
K->CK|E
C->c
```

- **The table: A→α, FIRST(A), FOLLOW(A) for NEW grammar**

```
The FIRST(A) and FOLLOW(A) of nonterminal A
+-----------------------------------+
A->X|Y      First(A)        Follow(A)
S->ABC        {a}              {$}
A->ab         {a}              {b}
B->bK         {b}              {c}
K->CK|E       {c,E}             {c}
C->c          {c}              {$,c, }
+-----------------------------------+
```

- **The parse table (NEW, Unambiguous)**

```
The obtained Parse Table
+-------------------------------------------+
     a            b            c            $
S   S->ABC        -            S->E         -
A   A->ab         -            -            -
B   -             B->bK        -            -
K   -             -            -            -
C   -             -            C->c         -
+-------------------------------------------+
```

- **The Input and Parsing of the Input**

```
Input String:abbcc$

+-------------------------------------------------------------+
Stack(X)                Input(a)                Process
[$, S]                  abbcc$          M[0,0] : rule = S->ABC
[$, C, B, A]            abbcc$          M[1,0] : rule = A->ab
[$, C, B, b, a]         abbcc$          a=a
[$, C, B, b]            bbcc$           b=b
[$, C, B]               bcc$            M[2,1] : rule = B->bC
[$, C, C, b]            bcc$            b=b
[$, C, C]               cc$             M[4,2] : rule = C->c
[$, C, c]               cc$             c=c
[$, C]                  c$              M[4,2] : rule = C->c
[$, c]                  c$              c=c
[$]                      $
+-------------------------------------------------------------+
```

The presence of input '$' at the end signifies that the given input can be parsed by the given algorithm.

5. Conclusion:

The program implements a simple non Recursive Predictive Parser. The program uses a static production rule (both unambiguous/non-recursive grammar) and hence the FIRST and FOLLOW is limited to the given symbols only. The program can parse the table limited to these symbols but can be done effectively for others too. The input symbol can be changed statically or changed to ask for dynamic inputs programmatically for the parsing process.