Module 2

Syntax Analysis: Review of Context-Free Grammars – Derivation and Parse trees, Ambiguity

Top Down Parsing: recursive Descent Parsing, Predictive Parsing, LL(1) Grammars

Context Free Grammar (CFG)

A context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

1. **Terminals** are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name.

2. **Nonterminals** are syntactic variables that denote sets of strings.

3. In a grammar, one nonterminal is distinguished as the **start symbol**, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.

4. The **productions** of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of:

    (a) A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.

    (b) The symbol →. Sometimes : : = has been used in place of the arrow.

    (c) A body or right side consisting of zero or more terminals and nonterminals.

Example: The grammar with the following productions defines simple arithmetic expression:

$$
\begin{aligned}
\textit{expression} &\rightarrow \textit{expression} + \textit{term} \\
\textit{expression} &\rightarrow \textit{expression} - \textit{term} \\
\textit{expression} &\rightarrow \textit{term} \\
\textit{term} &\rightarrow \textit{term} * \textit{factor} \\
\textit{term} &\rightarrow \textit{term} \, / \, \textit{factor} \\
\textit{term} &\rightarrow \textit{factor} \\
\textit{factor} &\rightarrow (\, \textit{expression} \,) \\
\textit{factor} &\rightarrow \mathbf{id}
\end{aligned}
$$

In this grammar, the terminal symbols are: **id + - * / ( )**

The nonterminal symbols are: **expression, term, factor**

Start symbol : **expression**

Notational Convent ions

To avoid always having to state that "these are the terminals," "these are the nonterminals," and so on, the following notational conventions for grammars will be used.

1. These symbols are terminals:

(a) Lowercase letters early in the alphabet, such as a, b, c.

(b) Operator symbols such as +, *, and so on.

(c) Punctuation symbols such as parentheses, comma, and so on.

(d) The digits 0, 1, . . . , 9.

(e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. These symbols are nonterminals:

(a) Uppercase letters early in the alphabet, such as A, B, C.

(b) The letter S, which, when it appears, is usually the start symbol.

(c) Lowercase, italic names such as *expr* or *stmt*.

(d)When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminalsfor expressions, terms, and factors are often represented by E, T, and F, respectively.

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet , chiefly u, v, ... ,z, represent (possibly empty) strings of terminals.
5. Lowercase Greek letters α, β, γ for example, represent (possibly empty) strings of grammar symbols.
6. A set of productions A → α1 , A → α2 , ... , A → αk with a common head A (call them A-productions) , may be written A → α1| α2|.....| αk · We call α1, α2,.., αn the alternatives for A.
7. Unless stated otherwise, the head of the first production is the start symbol.

Using these conventions, the grammar for arithmetic expression can be rewritten as:

E→ E + T | E - T | T

T→ T * F | T / F | F

F→ (E) | **id**

Derivations

The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.

For example , consider the following grammar , with a single nonterminal E:

E → E + E | E * E | - E | (E) | **id**

The production E → - E signifies that if E denotes an expression, then – E must also denote an expression. The replacement of a single E by - E will be described by writing

E => -E

which is read, "E derives - E." The production E -+ (E) can be applied to replace any instance of E in any string of grammar symbols by (E) ,

e.g., E * E => (E) * E or E * E => E * (E)

We can take a single E and repeatedly apply productiqns in any order to get a sequence of replacements. For example,

E => - E => - (E) => - (id)

We call such a sequence of replacements a **derivation** of - (id) from E. This derivation provides a proof that the string - (id) is one particular instance of an expression.

Derivation Order

$$1.\ S \to AB \qquad 2.\ A \to aaA \qquad 4.\ B \to Bb$$

$$3.\ A \to \lambda \qquad 5.\ B \to \lambda$$

Leftmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

Rightmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{5}{\Rightarrow} Ab \overset{2}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab$$

Basic Parsing Approaches

Parsing is the process of determining if a string of token can be generated by a grammar.

Mainly 2 parsing approaches:

1. Top Down Parsing
2. Bottom Up Parsing

In top down parsing, parse tree is constructed from top (root) to the bottom (leaves).

In bottom up parsing, parse tree is constructed from bottom (leaves)) to the top (root).

Top Down Parsing

It can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of parse tree in preorder.

Preorder traversal means: 1. Visit the root 2. Traverse left subtree 3. Traverse right subtree

Top down parsing can be viewed as an attempt to find a leftmost derivation for an input string (that is expanding the leftmost terminal at every step).

Recursive Descent Parsing

It is the most general form of top-down parsing. It may involve backtracking, that is making repeated scans of input, to obtain the correct expansion of the leftmost non-terminal. Unless the grammar is ambiguous or left-recursive, it finds a suitable parse tree

<u>Example:</u>

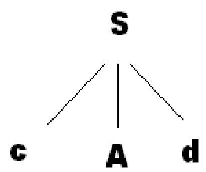
Consider the grammar:

S -> cAd | bd

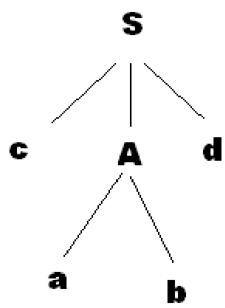
A -> ab | a

and the input string **w = cad.**

To construct a parse tree for this string top down, we initially create a tree consisting of a single node labelled **S**. An input pointer points to c, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree as:

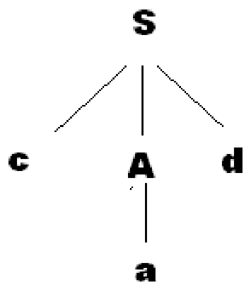


The leftmost leaf, labeled **c**, matches the first symbol of input w, so we advance the input pointer to **a**, the second symbol of **w**, and consider the next leaf, labeled **A**. Now, we expand **A** using the first alternative **A → ab** to obtain the tree as:



We have a match for the second input symbol, **a**, so we advance the input pointer to **d**, the third input symbol, and compare d against the next leaf, labeled **b**. Since **b** does not match **d**, we report failure and go back to **A** to see whether there is another alternative for **A** that has not been tried, but that might produce a match.

In going back to **A**, we must reset the input pointer to position 2 , the position it had when we first came to **A**, which means that the procedure for **A** must store the input pointer in a local variable.The second alternative for **A** produces the tree as:



The leaf **a** matches the second symbol of **w** and the leaf **d** matches the third symbol. Since we have produced a parse tree for w, we halt and announce successful completion of parsing. (that is the string parsed completely and the parser stops).

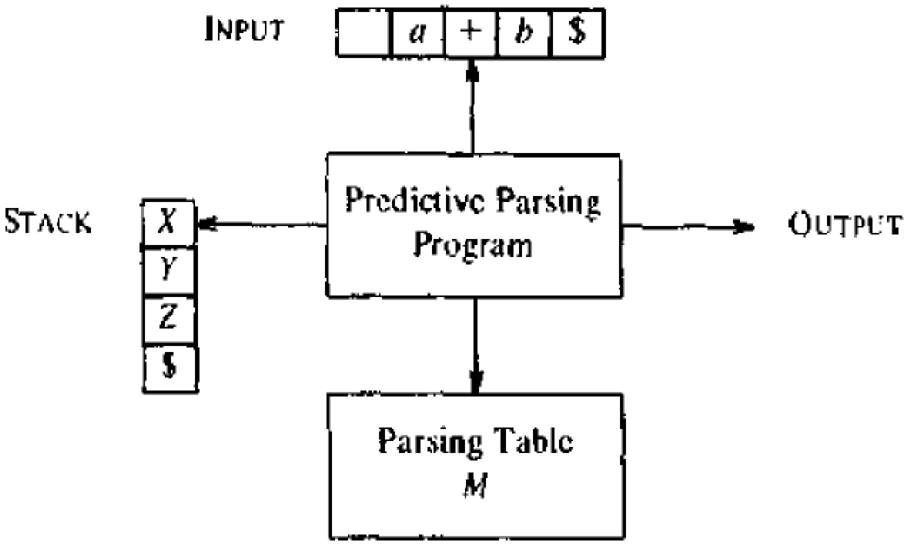
A left-recursive grammar can cause a recursive-descent parser, to go into an infinite loop. That is when we try to expand A, we may find ourselves again trying to expanding A, without having consumed any input.

Recursive-descent parsers are not very common as programming language constructs can be parsed without resorting to backtracking.

LL Parser

- Top Down Parsing
- Predictive Parsing
  - To construct a predictive parser we must know, given the current input symbol a and the nonterminal A to be expanded, which one of the alternatives of production A→a1|a2|..|an is the unique alternative of strings that begin with a.
- LL means:
  - First L : means that scanning takes place from Left to right
  - Second L: means that the Left derivation is produced first
- LL(1): The “1" in parentheses implies that LL(1) parsing uses only one symbol of input to predict the next grammar rule that should be used.
- Non Recursive Predictive parser
- Requirements are:
  - 1.Stack
  - 2.Parsing Table
  - 3.Input Buffer

– 4.Parsing program



Model of a nonrecursive predictive parser.

- **Input buffer** - contains the string to be parsed, followed by $(used to indicate end of input string)
- **Stack** – initialized with $, to indicate bottom of stack.
- **Parsing table -** 2 D array M[A,a] where A is a nonterminal and a is terminal or the symbol $

Predictive Parsing Algorithm

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 below

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else    /* X is a nonterminal */
        if M[X, a] = X → Y1Y2 ··· Yk then begin
            pop X from the stack;
            push Yk, Yk-1, ..., Y1 onto the stack, with Y1 on top;
            output the production X → Y1Y2 ··· Yk
        end
        else error()
until X = $    /* stack is empty */
```

Example:

Grammar:

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \text{id}$			$F \to (E)$		

Parsing table M

Moves made by predictive parser for the input id+id*id

STACK	INPUT	OUTPUT
$\$E$	id + id * id$	
$\$E'T$	id + id * id$	$E \to TE'$
$\$E'T'F$	id + id * id$	$T \to FT'$
$\$E'T'\text{id}$	id + id * id$	$F \to \text{id}$
$\$E'T'$	+ id * id$	
$\$E'$	+ id * id$	$T' \to \epsilon$
$\$E'T+$	+ id * id$	$E' \to +TE'$
$\$E'T$	id * id$	
$\$E'T'F$	id * id$	$T \to FT'$
$\$E'T'\text{id}$	id * id$	$F \to \text{id}$
$\$E'T'$	* id$	
$\$E'T'F*$	* id$	$T' \to *FT'$
$\$E'T'F$	id$	
$\$E'T'\text{id}$	id$	$F \to \text{id}$
$\$E'T'$	$	
$\$E'$	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

Construction of predictive parsing table

- Uses 2 functions:
  - FIRST()
  - FOLLOW()
  - These functions allows us to fill the entries of predictive parsing table

FIRST

If 'α' is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the string derived from α . If α==*>ϵ then add ϵ to FIRST(α).

Rules To Compute First Set

1) If X is a terminal , then FIRST(X) is {X}

2) If X--> ϵ then add ϵ to FIRST(X)

3) If X is a non terminal and X-->Y1Y2Y3...Yn , then put 'a' in FIRST(X) if for some i, a is in FIRST(Yi) and ϵ is in all of FIRST(Y1),...FIRST(Yi-1).

Example:

Consider the grammar G:

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

	FIRST
E	*Id , (*
E'	+ , ε
T	*Id , (*
T'	* , ε
F	*Id , (*

FOLLOW

- FOLLOW is defined only for non terminals of the grammar G.
- It can be defined as the set of terminals of grammar G , which can immediately follow the non terminal in a production rule from start symbol.
- In other words, if A is a nonterminal, then FOLLOW(A) is the set of terminals 'a' that can appear immediately to the right of A in some sentential form

<u>Rules To Compute Follow Set</u>

1. If S is the start symbol, then add $ to the FOLLOW(S).

2. If there is a production rule A--> αBβ then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).

3. If there is a production A--> αB , or a production A--> αBβ where FIRST(β) contains ϵ then everything in FOLLOW(A) is in FOLLOW(B).

Example:

Consider the grammar G:

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

	FIRST	***FOLLOW***
E	*Id , (*	*$,)*
E'	+	*$,)*
T	*Id , (*	+ , *$,)*
T'	*	+ , *$,)*
F	*Id , (*	+ , * , *$,)*

Algorithm to construct predictive parsing table:

Input. Grammar G.

Output. Parsing table M.

Method.

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be error.

	FIRST	*FOLLOW*
E	*Id* , (	\$,)
E'	+	\$,)
T	*Id* , (	+ , \$,)
T'	*	+ , \$,)
F	*Id* , (	+ , * , \$,)

NONTER-MINAL	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	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Parsing table M