CHAPTER 2 : SYNTAX AND SEMANTIC

CONTENTS

Topics:

Introduction

Syntax:

The form or structure of the expressions, statements, and program units

Semantics:

The meaning of the expressions, statements, and program units.

Syntax and semantics provide a language‘s definition

Users of a language definition

- Other language designers
- Implementers
- Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

- A sentence is a string of characters over some alphabet
- A language is a set of sentences

- A lexeme is the lowest level syntactic unit of a language (e.g., *, sum, begin)
- A token is a category of lexemes (e.g., identifier)

Formal Definition of Languages

Recognizers

- A recognition device reads input strings of the language and decides whether the input strings belong to the language

Generators

- A device that generates sentences of a language
- One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

Formal Methods of Describing Syntax

Backus-Naur Form and Context-Free Grammars

- Developed by Noam Chomsky in the mid-1950s.

A context Free Grammar (CFG) is a 4-tuple such that- **G = (V , T , P , S)**
where-
V = Finite non-empty set of variables / non-terminal symbols
T = Finite set of terminal symbols
P = Finite non-empty set of production rules of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$
S = Start symbol

•A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
•A set of tokens, known as **terminal symbols** (T). Terminals are the basic symbols from which strings are formed.
•A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.

Example Context free grammar

Consider a grammar G = (V , T , P , S) where-

- V = { S }
- T = { a , b }
- P = { S → aSbS , S → bSaS , S → ∈ }
- S = { S }

So, Language of this grammar is-

L(G) = { ∈ , ab , ba , aabb , bbaa , abab , baba , …… }

This grammar is an example of a context free grammar.

- It generates the strings having equal number of a's and b's.
- In the above grammar The Nonterminal is : S
- Terminal is : a,b, ∈
- Start Symbol is S
- And the Number of production is 3

Application of CFG (Context free grammar)

Context Free Grammar (CFG) is of great practical importance. It is used for following purposes-

- For defining programming languages
- For parsing the program by constructing syntax tree
- For translation of programming languages
- For describing arithmetic expressions
- For construction of compilers.

Backus-Naur Form (BNF)

BNF (Backus–Naur Form) is a context-free grammar commonly used by developers of programming languages to specify the syntax rules of a language.

John Backus was a program language designer who devised a notation to document IAL (an early implementation of Algol).

Peter Naur later worked on Backus’ findings, and the notation was jointly credited to both computer scientists.

BNF uses a range of symbols and expressions to create **production rules**.

A simple BNF production rule might look like this:

<digit> ::= 0|1|2|3|4|5|6|7|8|9

This would be interpreted as: *a digit can be defined as 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9*

The chevrons (< >) are used to denote a **non-terminal symbol**. If a non-terminal symbol appears on the right side of the production rules, it means that there will be another production rule (or set of rules) to define its replacement.

There is the production for any grammar as follows:

1. S → aSa
2. S → bSb
3. S → c

In BNF, we can represent above grammar as follows:

1. S → aSa| bSb| c

BNF Fundamentals

In the above grammar

Non-terminals: BNF abstractions = S

Terminals: lexemes and tokens = a,b,c

Grammar: a collection of rules

Examples of BNF rules:

<ident_list> → identifier | identifer, <ident_list>

<if_stmt> → if <logic_expr> then <stmt>

Example of Context free grammar

1. $S \rightarrow aSa$
2. $S \rightarrow bSb$
3. $S \rightarrow c$

Now check that abbcbba string can be derived from the given CFG.

1. $S \Rightarrow aSa$
2. $S \Rightarrow abSba$
3. $S \Rightarrow abbSbba$
4. $S \Rightarrow abbcbba$

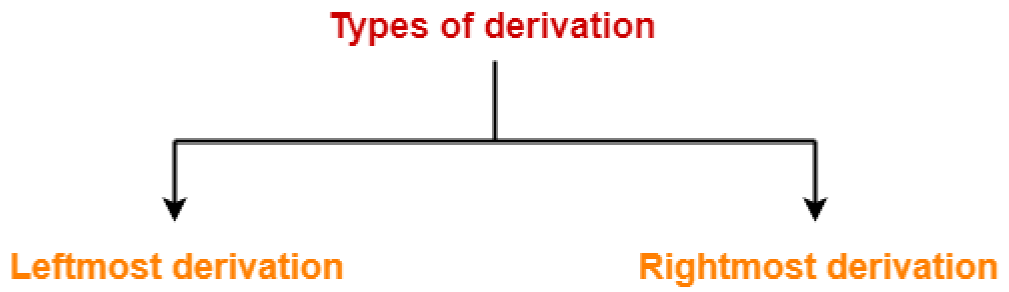
By applying the production $S \rightarrow aSa$, $S \rightarrow bSb$ recursively and finally applying the production $S \rightarrow c$, we get the string abbcbba.

DERIVATION

What is parse tree?

The process of deriving a string is called as derivation.

The geometrical representation of a derivation is called as a parse tree or derivation tree.



Leftmost Derivation-

•The process of deriving a string by expanding the leftmost non-terminal at each step is called as **leftmost derivation**.

Rightmost Derivation-

•The process of deriving a string by expanding the rightmost non-terminal at each step is called as **rightmost derivation**.

In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

Consider the grammar-

S → A1B

A → 0A / ∈

B → 0B / 1B / ∈

For the string w = 00101, find-

1. Leftmost derivation
2. Rightmost derivation
3. Parse Tree

Solution :

Leftmost Derivation-

S → **A**1B
→ 0**A**1B (Using A → 0A)
→ 00**A**1B (Using A → 0A)
→ 001**B** (Using A → ∈)
→ 0010**B** (Using B → 0B)
→ 00101**B** (Using B → 1B)
→ 00101 (Using B → ∈)

Rightmost Derivation-

S → A1B

→ A10B (Using B → 0B)

→ A101B (Using B → 1B)

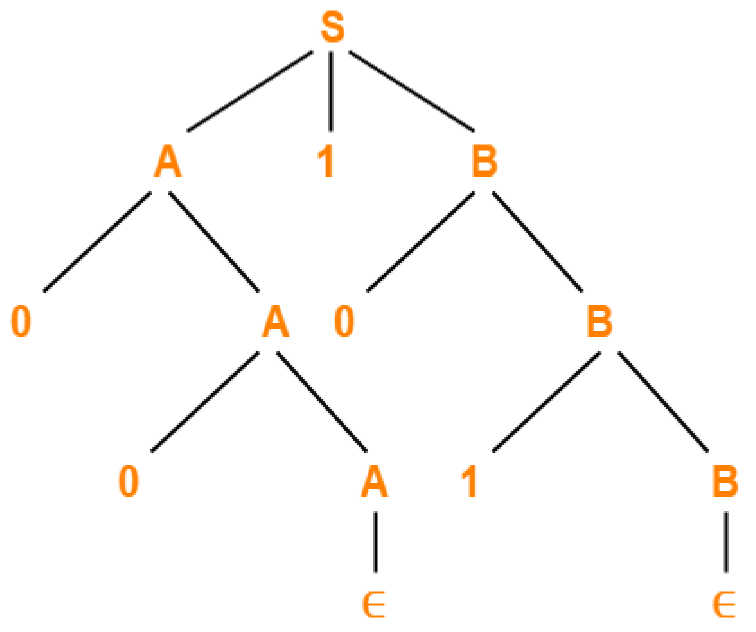
→ A101 (Using B → ∈)

→ 0A101 (Using A → 0A)

→ 00A101 (Using A → 0A)

→ 00101 (Using A → ∈)

Parse Tree



Parse Tree

Ambiguity in Grammars

A grammar is ambiguous if it generates a sentential form that has two or more distinct parse trees.

Semantics

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs.

The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking
- Array-bound checking

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Example:

```
E → E + T { E.value = E.value + T.value }
```

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

The attributes can be a number, type, memory location, return type etc….

Types of attributes are:

1. Synthesized attribute
2. Inherited attribute

Synthesized attributes

- Value of synthesized attribute at a node can be computed from the value of attributes at the children of that node in the parse tree.
- A syntax directed definition that uses synthesized attribute exclusively is said to be S-attribute definition.

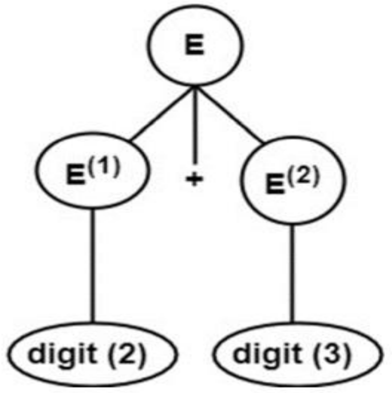
For Example, In Productions.

$E \rightarrow E^{(1)} + E^{(2)}$

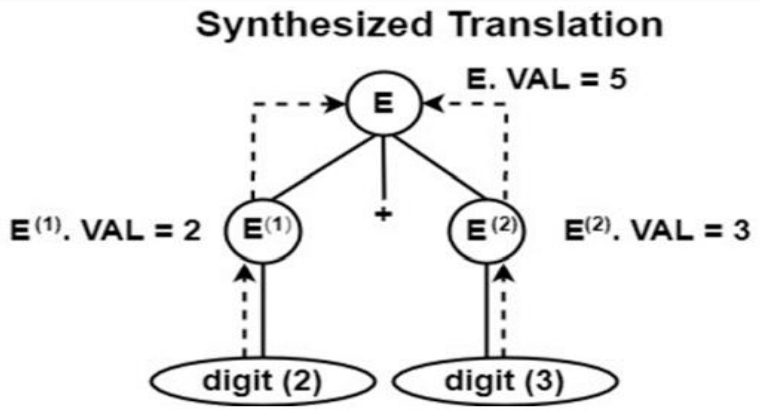
$E \rightarrow digit$

where digit = 0 | 1 | 2 |… . . | 9

The Parse Tree for string 2 + 3 will be



The complete parse tree, i.e., Parse Tree with translation will consist of a tree with each node labeled with attribute VAL.



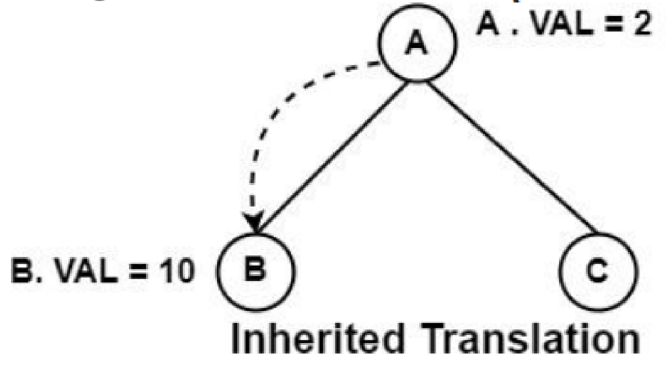
Here, E. VAL represents the value of non-terminal E, computed from the values at the children of that node in Parse Tree.

Inherited attribute

- An inherited value at a node in a parse tree is computed from the value of attributes at the parent and/or siblings of the node.

For example 1, a complete parse tree will be

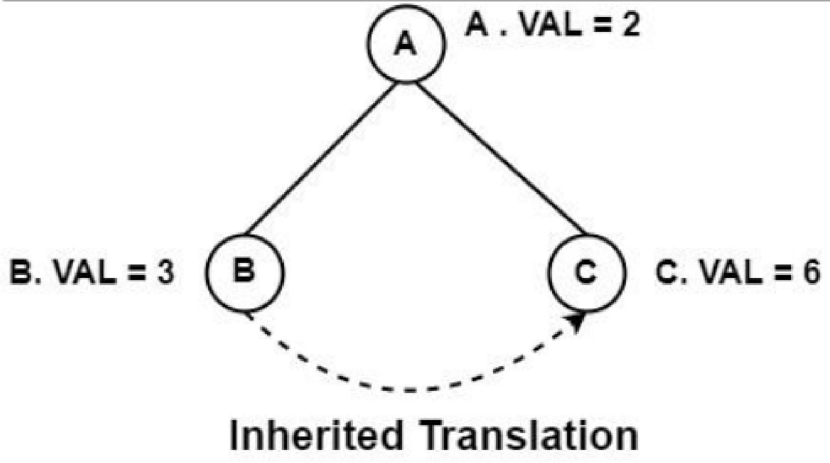
$A \longrightarrow BC$ $\{B.\ VAL = 5 * A.\ VAL\}$



If A.VAL = 2, so, B.VAL = 5 * 2 = 10

So, the value of B is computed from the parent of that node.

Example 2 of Inherited Translation– Consider the production
$A \longrightarrow BC$ $\{C.\ VAL = 2 * B.\ VAL\}$
Its complete parse tree will be



Here, value at node C on its sibling B. So, in inherited translation, the value of inherited attributes at a node depends upon its parent node or Sibling node.

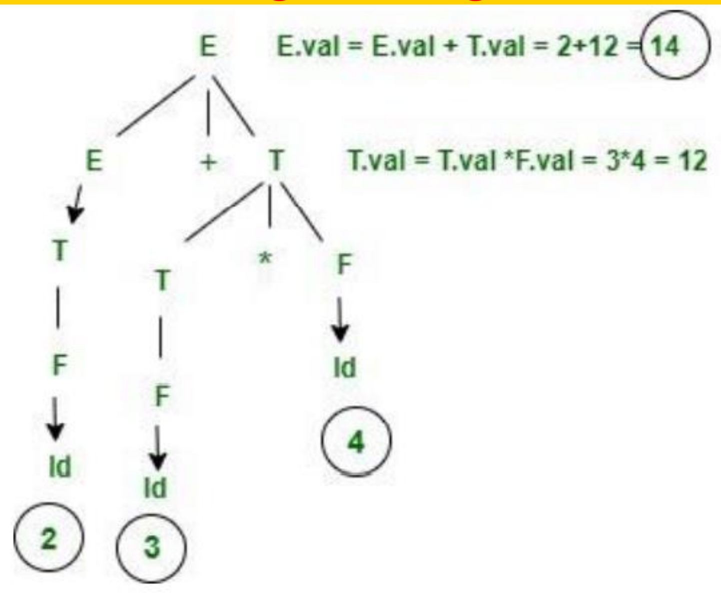
Example

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.
num.lexval is the attribute returned by the lexical analyzer.

Q: Draw the parse tree and translate the expression 2 + 3 * 2 using following semantic rule.

Production	Semantic Rules
E → E + T	{E.VAL := E.VAL + T.VAL }
E → T	{E.VAL := T.VAL }
T → T * F	{T.VAL := T.VAL * F. VAL }
T→ F	{T.VAL := F.VAL }
F → Id	{F.VAL := Id }

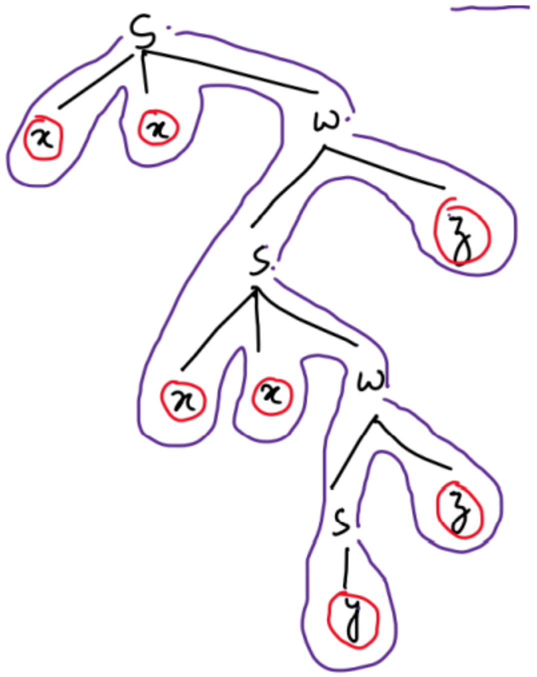

E
E.val = E.val + T.val = 2+12 = 14
E
+
T
T.val = T.val *F.val = 3*4 = 12
T
T
*
F
F
F
Id
Id
Id
2
3
4

Write the value of xxxxyzz from the following translation scheme.

Production	Semantic Rules (Action)
S→ xxw	{print(1)}
S → y	{print(2)}
W→Sz	{print(3)}

Solution:

- Draw the Parse tree according to grammar then traverse through depth first traversal.
- Wherever you find the reduction, see the production and take the action.



Output is 23131

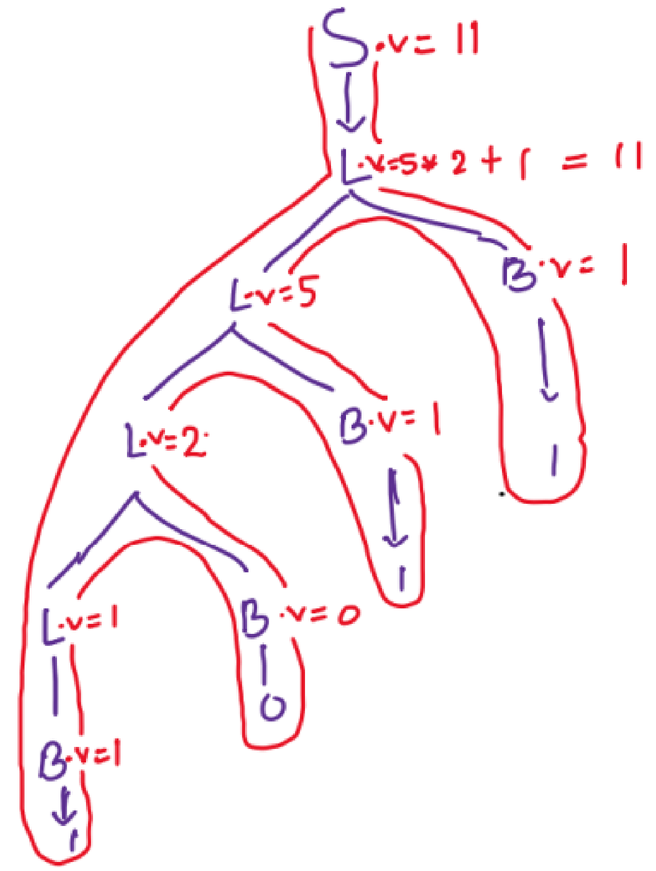
Convert 1011 into decimal number using following translation scheme.

Production	Semantic Rules
S → L	{S.VAL:= L.VAL }
L→ LB	{L.VAL := L.VAL * 2 + B.VAL }
L→ B	{L.VAL := B.VAL }
B→ 0	{B.VAL := 0 }
B→ 1	{B.VAL := 1 }

Solution:

- Draw the Parse tree according to grammar then traverse through depth first traversal.
- Wherever you find the reduction, see the production and take the action.

Output is 11



Any Questions?