MCA Part III

Paper- XIX

Topic: Semantic Analysis

Prepared by: Dr. Kiran Pandey

School of Computer science

Email-Id: kiranpandey.nou@gmail.com

INTRODUCTION

We have learnt how a parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in syntax phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

Example:

E → E + T

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production.

SEMANTIC ANALYSIS

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”

will not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it will generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks will 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.

ATTRIBUTES OF 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.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

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 will be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories:

(i) Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

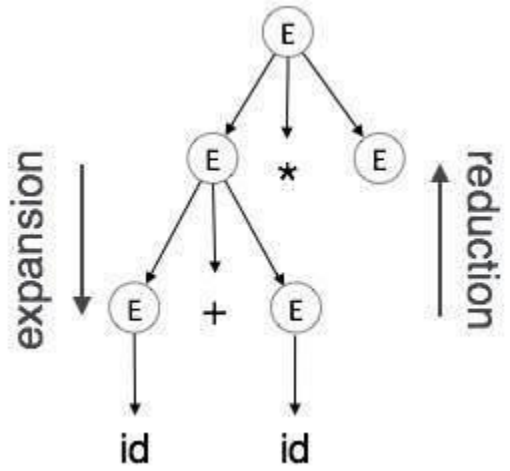
(ii) Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion: When a non-terminal is expanded to terminals as per a grammatical rule



Reduction: When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions). Semantic analysis uses Syntax Directed Translations to perform the above tasks. Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis). Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

int value = 10;

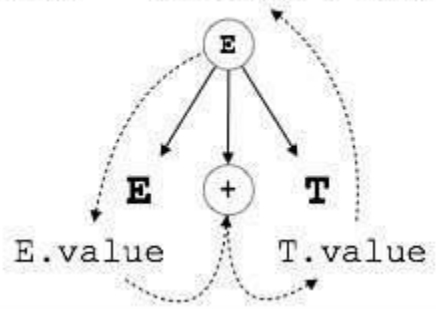
<type, “integer”>

<presentvalue, “10”>

For every production, we attach a semantic rule.

S-attributed SDT: If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).

```
E.value = E.value + T.value
```



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

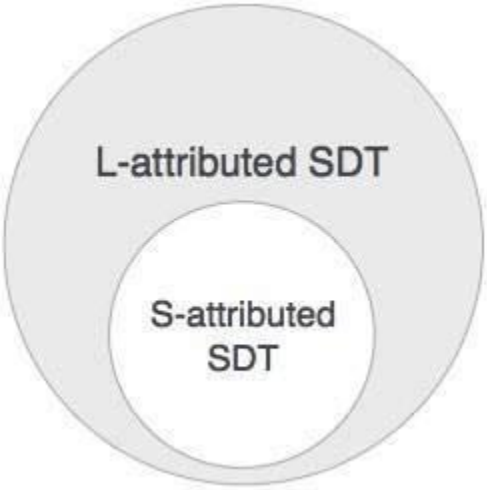
L-attributed SDT: This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

$S \rightarrow ABC$

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

SYMBOL TABLE

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbol name, type, attribute>

For example, if a symbol table has to store information about the following variable declaration:

Static int income;

then it will store the entry such as:

<income, int, static>

The attribute clause contains the entries related to the name.

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Operations on Symbol Table

A symbol table, either linear or hash, will provide the following operations.

(a) insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

int a;

will be processed by the compiler as:

insert(a,int);

(b) lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format will match the following: **lookup(symbol)**

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

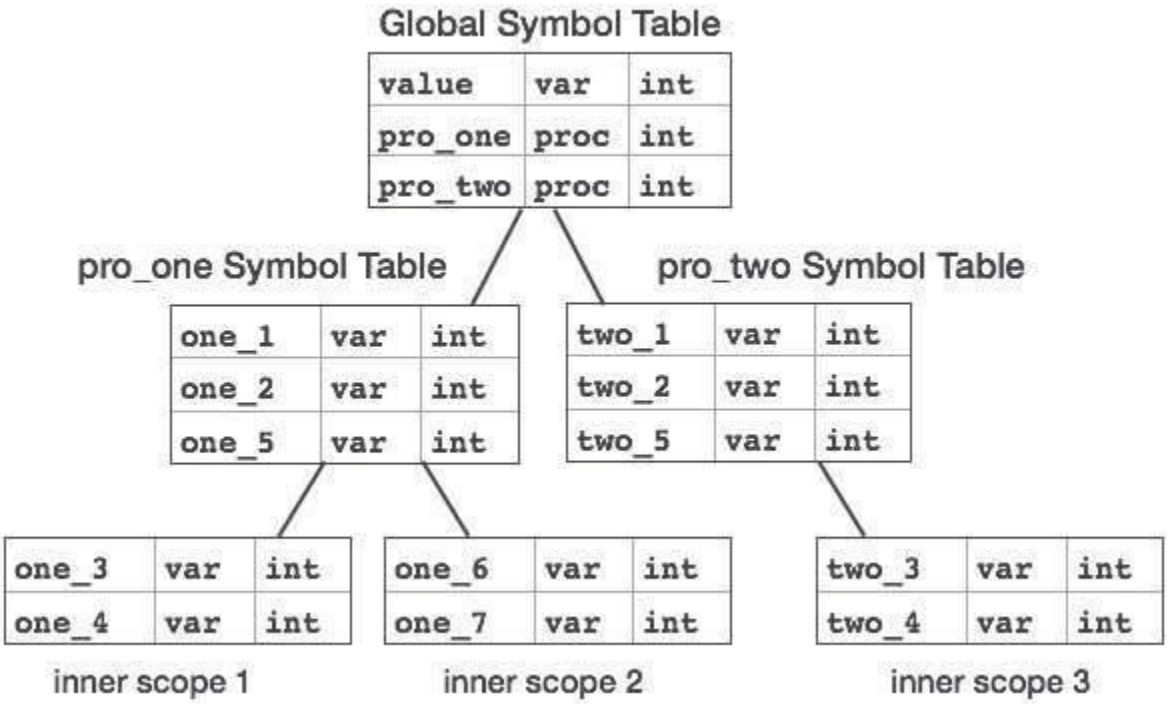
Scope Management

A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program. To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;


void pro_one()
  {
  int one_1;
  int one_2;


    {           \
    int one_3;    |_  inner scope 1
    int one_4;    |
    }           /


  int one_5;


    {           \
    int one_6;    |_  inner scope 2
    int one_7;    |
    }           /
  }


void pro_two()
  {
  int two_1;
  int two_2;
```

```
    {           \
    int two_3;     |_  inner scope 3
    int two_4;     |
    }           /


  int two_5;
  }
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains names for one global variable (int value) and two procedure names, which will be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.

PARSE TREE

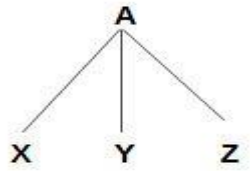
Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.

• Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.

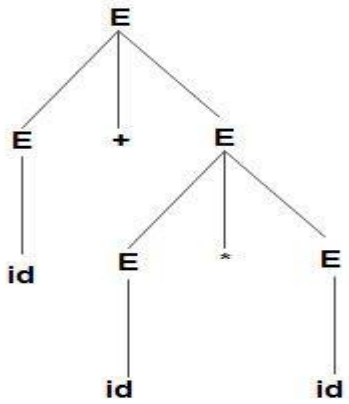
• Leaves of parse tree represent terminals.

• Each interior node represents productions of grammar.

• If *A -> xyz* is a production, then the parse tree will have *A* as interior node whose children are *x, y* and *z* from its left to right.



Construct parse tree for E --> E + E I E * E I **id**



Parse trees can be used to represent real-world constructions like sentences or mathematical expressions.

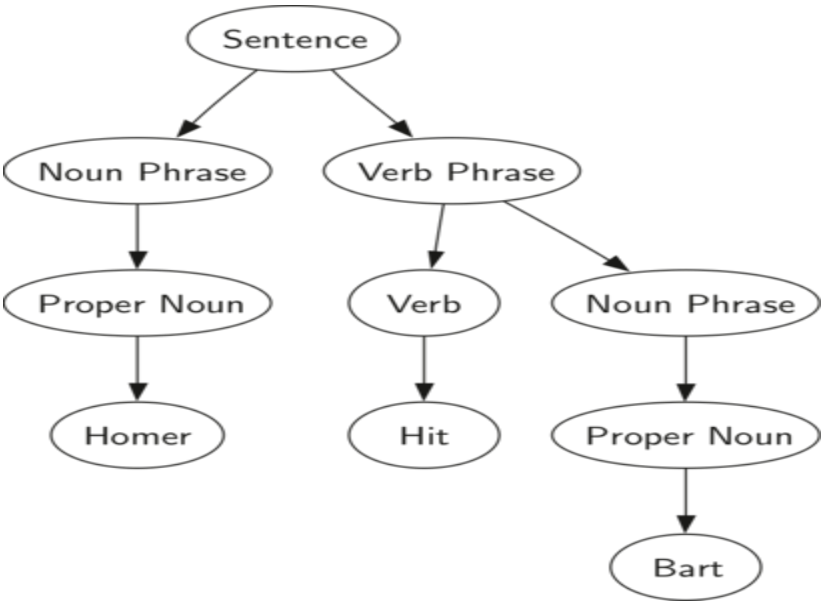
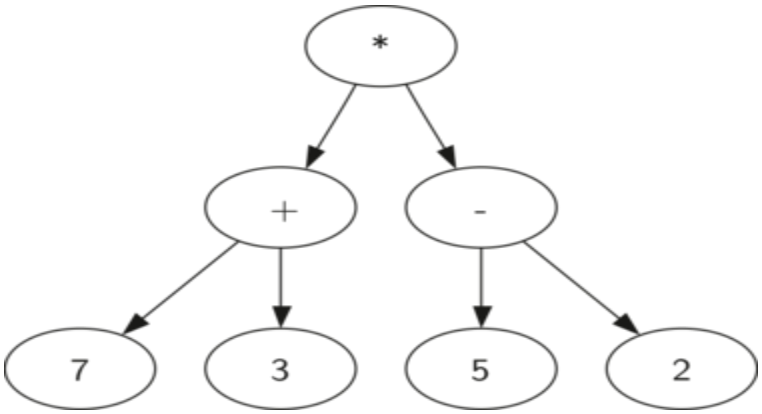
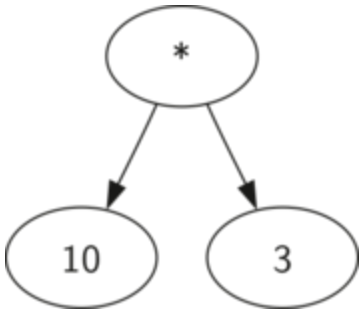


Figure above shows the hierarchical structure of a simple sentence. Representing a sentence as a tree structure allows us to work with the individual parts of the sentence by using subtrees.



We can also represent a mathematical expression such as ((7+3)∗(5−2))((7+3)∗(5−2)) as a parse tree, as shown in Figure above . We have already looked at fully parenthesized expressions, so what do we know about this expression? We know that multiplication has a higher precedence than either addition or subtraction. Because of the parentheses, we know that before we can do the multiplication we must evaluate the parenthesized addition and subtraction expressions. The hierarchy of the tree helps us understand the order of evaluation for the whole expression. Before we can evaluate the top-level multiplication, we must evaluate the addition and the subtraction in the subtrees. The addition, which is the left subtree, evaluates to 10. The subtraction, which is the right subtree, evaluates to 3. Using the hierarchical structure of trees, we can simply replace an entire subtree with one node once we have evaluated the expressions in the children. Applying this replacement procedure gives us the simplified tree shown in Figure below.



In the rest of this section we are going to examine parse trees in more detail. In particular we will look at

- How to build a parse tree from a fully parenthesized mathematical expression.
- How to evaluate the expression stored in a parse tree.
- How to recover the original mathematical expression from a parse tree.

The first step in building a parse tree is to break up the expression string into a list of tokens. There are four different kinds of tokens to consider: left parentheses, right parentheses, operators, and operands. We know that whenever we read a left parenthesis we are starting a new expression, and hence we should create a new tree to correspond to that expression. Conversely, whenever we read a right parenthesis, we have finished an expression. We also know that operands are going to be leaf nodes and children of their operators. Finally, we know that every operator is going to have both a left and a right child.

Using the information from above we can define four rules as follows:

1. If the current token is a '(', add a new node as the left child of the current node, and descend to the left child.
2. If the current token is in the list ['+','-','/','*'], set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.
3. If the current token is a number, set the root value of the current node to the number and return to the parent.
4. If the current token is a ')', go to the parent of the current node.

ABSTRACT SYNTAX TREE

Abstract syntax trees are data structures widely used in compilers to represent the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

An abstract syntax tree (AST) is a way of representing the syntax of a programming language as a hierarchical tree-like structure. This structure is used for generating symbol tables for compilers and later code generation. The tree represents all of the constructs in the language and their subsequent rules.

Design

The design of an AST is often closely linked with the design of a compiler and its expected features.

Core requirements include the following:

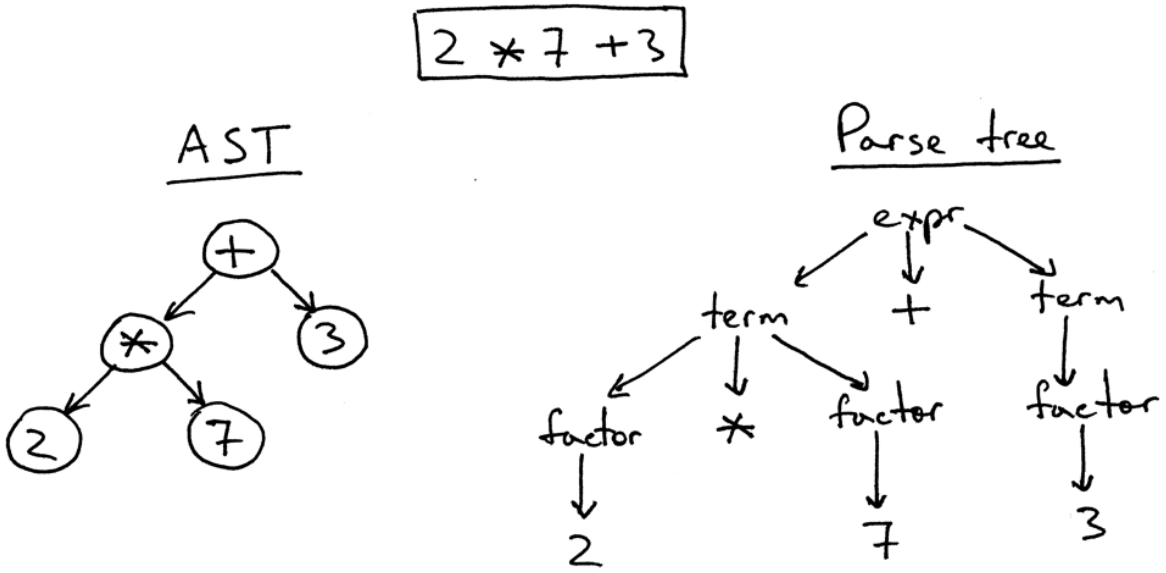
- Variable types must be preserved, as well as the location of each declaration in source code.
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

These requirements can be used to design the data structure for the AST.

Some operations will always require two elements, such as the two terms for addition. However, some language constructs require an arbitrarily large number of children, such as argument lists passed to programs from the command shell. As a result, an AST used to represent code written in such a language has to also be flexible enough to allow for quick addition of an unknown quantity of children.

Another major design requirement for an AST is that it will be possible to unparse an AST into source code form.[*why?*] The source code produced will be sufficiently similar to the original in appearance and identical in execution, upon recompilation.

An example is given below:



An abstract syntax tree represents all of the syntactical elements of a programming language, similar to syntax trees that linguists use for human languages. The tree focuses on the rules rather than elements like braces or semicolons that terminate statements in some languages. The tree is hierarchical, with the elements of programming statements broken down into their parts. For example, a tree for a conditional statement has the rules for variables hanging down from the required operator.

ASTs are widely used in compilers to check code for accuracy. If the generated tree contains errors, the compiler prints an error message. ASTs are used because some constructs cannot be represented in a context-free grammar, such as implicit typing. ASTs are highly specific to programming languages, but research is underway on universal syntax trees.

QUESTIONS

1. **Define semantic analysis.**
2. **What is a grammar?**
3. **Define symbol table. Discuss the attributes of symbol table.**
4. **What is a parse tree?**
5. **Compare a parse tree with an abstract syntax tree.**