Unit-4 (Semantic Analysis)

Background : Parser uses a CFG(Context-free-Grammer) to validate the input string and produce output for next phase of the compiler. Output could be either a parse tree or abstract syntax tree. Now to interleave semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation.

Definition

Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. Syntax directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated to the non-terminals in their definitions.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

Example

E -> E+T | T

T -> T*F | F

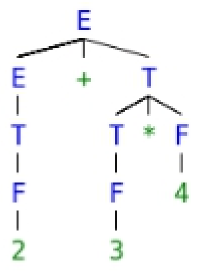
F -> INTLIT

This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example we will focus on evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

E -> E+T	{ E.val = E.val + T.val }	PR#1
E -> T	{ E.val = T.val }	PR#2
T -> T*F	{ T.val = T.val * F.val }	PR#3
T -> F	{ T.val = F.val }	PR#4
F -> INTLIT	{ F.val = INTLIT.lexval }	PR#5

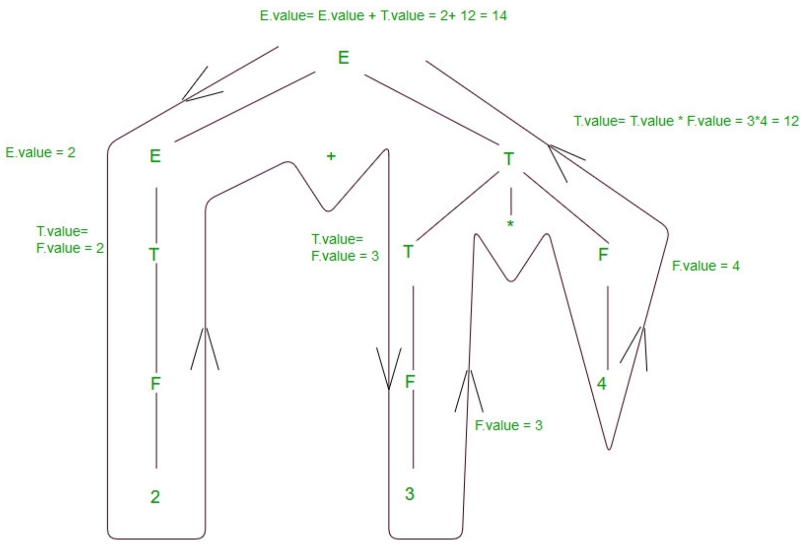
For understanding translation rules further, we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has val as attribute for both the non-terminals – E & T. Right hand side of the translation rule corresponds to attribute values of right side nodes of the production rule and vice-versa. Generalizing, SDT are augmented rules to a CFG that associate 1) set of attributes to every node of the grammar and 2) set of translation rules to every production rule using attributes, constants and lexical values.

Let's take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be



Unit-4 (Semantic Analysis)

To evaluate translation rules, we can employ one depth first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until children attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom up in left to right fashion for computing translation rules of our example.



Above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children attributes are computed before parents, as discussed above. Right hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parent.

Additional Information

Synthesized Attributes are such attributes that depend only on the attribute values of children nodes.

Thus [E -> E+T { E.val = E.val + T.val }] has a synthesized attribute val corresponding to node E. If all the semantic attributes in an augmented grammar are synthesized, one depth first search traversal in any order is sufficient for semantic analysis phase.

Inherited Attributes are such attributes that depend on parent and/or siblings attributes.

Thus [Ep -> E+T { Ep.val = E.val + T.val, T.val = Ep.val }], where E & Ep are same production symbols annotated to differentiate between parent and child, has an inherited attribute val corresponding to node T.

- Difference between synthesized and inherited attribute,

No	Synthesized Attribute	Inherited attribute
1	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.	Values of the inherited attribute at a node can be computed from the value of attribute at the parent and/or siblings of the node.
2	Pass the information from bottom to top in the parse tree.	Pass the information top to bottom in the parse tree *or* from left siblings to the right siblings

Table 3.2.1 Difference between Synthesized and Inherited attribute

Application of Syntax Directed Translation

- ➢ SDT is used for Executing Arithmetic Expression.
- ➢ In the conversion from infix to postfix expression.
- ➢ In the conversion from infix to prefix expression.
- ➢ It is also used for Binary to decimal conversion.
- ➢ In counting number of Reduction.
- ➢ In creating a Syntax tree.
- ➢ SDT is used to generate intermediate code.
- ➢ In storing information into symbol table.

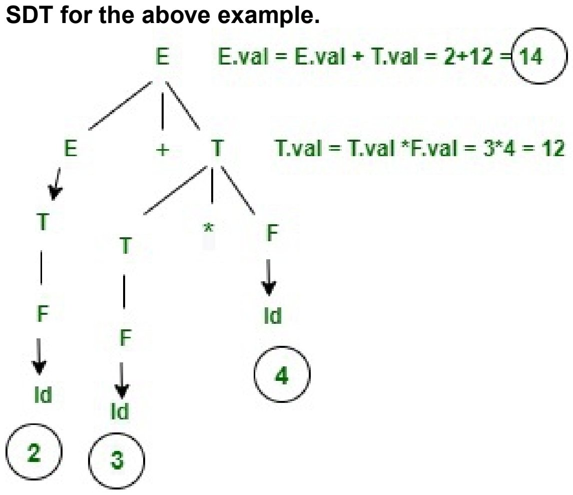
- SDT is commonly used for type checking also.

Example :
Here, we are going to cover an example of application of SDT for better understanding the SDT application uses. let's consider an example of arithmetic expression and then you will see how SDT will be constructed.

Let's consider Arithmetic Expression is given.

Input : 2+3*4
output: 14

SDT for the above example.


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

Semantic Action is given as following.

E -> E+T { E.val = E.val + T.val then print (E.val)}
|T { E.val = T.val}
T -> T*F { T.val = T.val * F.val}
|F { T.val = F.val}
F -> Id {F.val = id}

Syntax Directed Definition

Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production X –> a is associated with it a set of production rules of the form s = f(b1, b2, ……bk) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

Example:

E --> E1 + T { E.val = E1.val + T.val}
Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features –

High level specification
Hides implementation details
Explicit order of evaluation is not specified

Construction of syntax tree for expression

Unit-4 (Semantic Analysis)

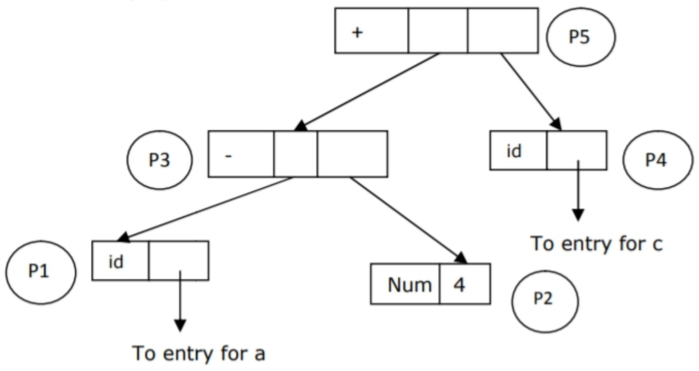
We use the following function to create binary operator. Each function returns a pointer to a newly created node.
1. Mknode(op,left,right) creates an operator node with label op and two fields containing pointers to left and right.

2. Mkleaf(id, entry) creates an identifier node with label id and a field containing entry, a pointer to the symbel table entry for the identifier.

3. Mkleaf(num, val) creates a number node with label num and a field containing val,thevalue of the number.

Example: construct syntax tree for a-4+c

P1:mkleaf(id, entry for a);
P2:mkleaf(num, 4);
P3:mknode('-',p1,p2);
P4:mkleaf(id, entry for c);
P5:mknode('+',p3,p4);



Bottom-Up Evaluation of S-attributed definitions

- An attribute grammar is a formal way to define attributes for the productions of a formal grammar, associating these attributes to values.
- The evaluation occurs in the nodes of the abstract syntax tree, when the language is processed by some parser or compiler.
- The attributes are divided into two groups:
  Synthesized attributes - The value of a synthesized attribute is computed from the values of attributes at the children of that node in the parse tree.
  Inherited attributes - The value of a inherited attribute is computed from the values of attributes at the siblings and parent of that node.
- In some approaches, synthesized attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down and across it.
- Syntax-directed definitions with only synthesized attributes are called S-attributes. This is commonly used in LR parsers.
- Only synthesized attributes appear in the syntax-directed definition in the following table for constructing the syntax tree for an expression.
-

S.no	Production	Semantic rule
1.	E -> E1 + T	E.node = new Node ('+', E1.node, T.node)
2	E -> E1 - T	E.node = new Node ('-', E1.node, T.node)
3	E -> T	E.node = T.node
4	T -> (E)	T.node = E.node
5	T -> id	T.node = new Leaf (id, id.entry)
6	T -> num	T.node = new Leaf (num, num.val)

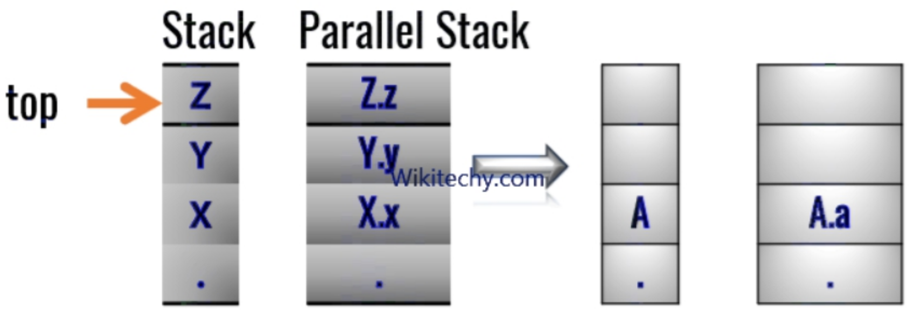
Unit-4 (Semantic Analysis)

Synthesized Attributes on the Parser Stack
A translator for an S-attributed definition can often be implemented with the help of an LR parser generator.
From an S-attributed definition, the parser generator can construct a translator that evaluates attributes as it parses the input.
A bottom-up parser uses a stack to hold information about subtrees that have been parsed. We can use extra fields in the parser stack to hold the values of synthesized attributes.
We put the values of the synthesized attributes of the grammar symbols into a parallel stack
When an entry of the parser stack holds a grammar symbol X the corresponding entry in the parallel stack will hold the synthesized attributes of the symbol X.

Example :

A -> XYZ

A.a = f(X.x , Y.y , Z.z) -> Synthesized attributes



Parser Stack

Bottom-Up Evaluation of L-attributed definitions

Introduction: if bottom-up can be done every translation so that top-down. More precisely, given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse.

The "trick" has three parts:

1. Start with the SDT constructed as in Section 5.4.5, which places embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.

2. Introduce into the grammar a marker nonterminal in place of each embedded action. Each such place gets a distinct marker, and there is one production for any marker M, namely M -> e.

3. Modify the action a if marker nonterminal M replaces it in some production A a {a} (5, and associate with M -> e an action a' that

   a. Copies, as inherited attributes of M, any attributes of A or symbols of a that action a needs.

   b. Computes attributes in the same way as a, but makes those attributes be synthesized attributes of M.

This change appears illegal, since typically the action associated with production M —>• e will have to access attributes belonging to grammar symbols that do not appear in this production. However, we shall implement the actions on the LR parsing stack, so the necessary attributes will always be available a known number of positions down the stack.

Example: Suppose that there is a production A -» B C in an LL grammar, and the inherited attribute B.i is computed from inherited attribute A.i by some formula B.i = f(A.i). That is, the fragment of an SDT we care about is

A-> {B.i = f{A.i);} B C

We introduce marker M with inherited attribute M.i and synthesized attribute M.s. The former will be a copy of A.i and the latter will be B.i. The SDT will be written

A -)> M B C

M - {M.i = A.i; M.s = f {M.i);}

Notice that the rule for M does not have A.i available to it, but in fact we shall arrange that every inherited attribute for a nonterminal such as A appears on the stack immediately below where the reduction to A will later take place.

Thus, when we reduce e to M, we shall find A.i immediately below it, from where it may be read. Also, the value of M.s, which is left on the stack along with M, is really B.i and properly is found right below where the reduction to B will later occur.