# \*Syntax Directed Translation

**Background:** Parser uses a CFG(Context-free-Grammar) 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 combine semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation.

In syntax directed translation, along with the grammar we associate(connect) some informal notations and these notations are called as **semantic rules.**

So, we can say that

* Grammar + semantic rule = SDT (syntax directed translation)
* In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
* In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
* In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

**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.

### **Example:1**

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.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

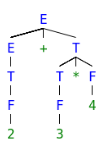
**E.val** is one of the attributes of E.

**num.lexval** is the attribute returned by the lexical analyzer.

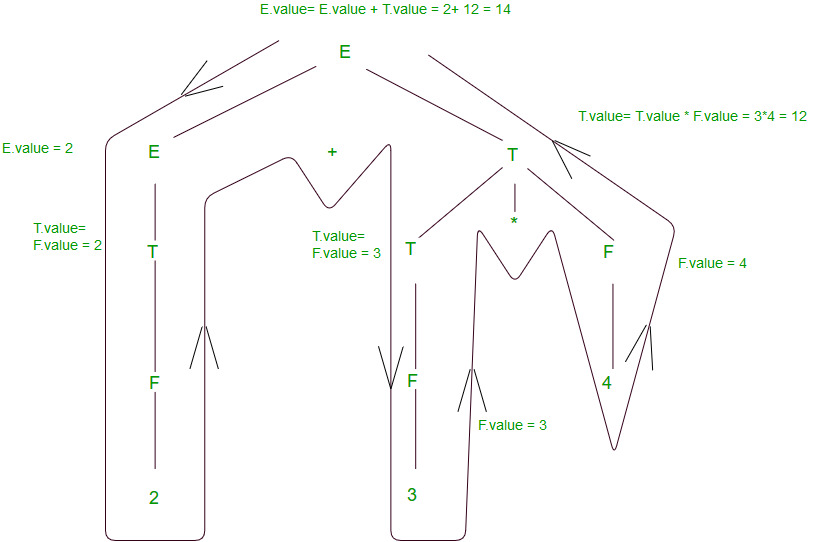
For understanding translation rules further, we take the first SDT augmented to [ E -> E+T ] production rule. The translation rule in consideration has **Val** as attribute for both the non-terminals E & T. Right hand side of the translation rule corresponds to attribute values of rightside nodes of 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



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.

**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

**Types of attributes –** There are two types of attributes:

**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.

**Example:**

E --> E1 + T { E.val = E1.val + T.val}

In this, E.val derive its values from E1.val and T.val

**Computation of Synthesized Attributes –**

* Write the SDD using appropriate semantic rules for each production in given grammar.
* The annotated parse tree is generated and attribute values are computed in bottom up manner.
* The value obtained at root node is the final output.

**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.

**Example:**

A --> BCD { C.in = A.in, C.type = B.type }

**Computation of Inherited Attributes –**

* Construct the SDD using semantic actions.
* The annotated parse tree is generated and attribute values are computed in top down manner.

**Example:** Consider the following grammar

S --> T L

T --> int

T --> float

T --> double

L --> L1, id

L --> id

The SDD for the above grammar can be written as follow



Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double. Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal. Using function Enter\_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

# \*Syntax directed translation scheme

* The Syntax directed translation scheme is a context -free grammar.
* The syntax directed translation scheme is used to evaluate the order of semantic rules.
* In translation scheme, the semantic rules are embedded within the right side of the productions.
* The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

**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}

# \*Application of Syntax Directed Translation

**Syntax Directed Translation:**  
It is used for semantic analysis and SDT is basically used to construct the parse tree with Grammar and Semantic action. In Grammar, need to decide who has the highest priority will be done first and in semantic action, will decide what type of action done by grammar.

**Example:**

SDT = Grammar + Semantic Action

Grammar = E -> E1+E2

Semantic action= if (E1.type != E2.type) then print "type mismatching"

**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.

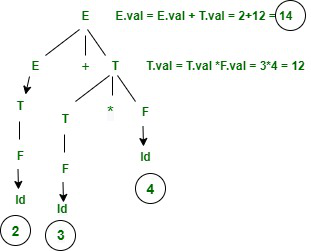
**Example:**  
Here, we are going to cover an example of application of SDT for better understanding the SDT application uses. let’s consider an example of arithmetic expression and then you will see how SDT will be constructed.

Let’s consider Arithmetic Expression is given.

Input: 2+3\*4

output: 14

SDT for the above example.



*SDT for 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}