Unit 5 : Syntax-Directed Translation

Syntax-directed translation (SDT) refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed.

SDT can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called **attribute grammars**.

We augment a grammar by associating **attributes** with each grammar symbol that describes its properties. With each production in a grammar, we give **semantic rules/ actions**, which describe how to compute the attribute values associated with each grammar symbol in a production.

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.

A class of syntax-directed translations called "L-attributed translations" (L for left-to-right) includes almost all translations that can be performed during parsing. Similarly, "S-attributed translations" (S for synthesized) can be performed easily in connection with a bottom-up parse.

There are two ways to represent the semantic rules associated with grammar symbols.

- Syntax-Directed Definitions (SDD)
- Syntax-Directed Translation Schemes (SDT)

Syntax-Directed Definitions

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler. If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. The attributes are evaluated by the semantic rules attached to the productions.

Example:	PRODUCTION	SEMANTIC RULE
	$E \rightarrow E1 + T$	E.code = E1.code || T.code || '+'

SDDs are highly readable and give high-level specifications for translations. But they hide many implementation details. For example, they do not specify order of evaluation of semantic actions.

Syntax-Directed Translation Schemes (SDT)

SDT embeds program fragments called semantic actions within production bodies. The position of semantic action in a production body determines the order in which the action is executed.

Example: In the rule E → E1 + T { print '+' }, the action is positioned after the body of the production.

SDTs are more efficient than SDDs as they indicate the order of evaluation of semantic actions associated with a production rule. This also gives some information about implementation details.

Inherited and Synthesized Attributes

Terminals can have synthesized attributes, which are given to it by the lexer (not the parser). There are no rules in an SDD giving values to attributes for terminals. Terminals do not have inherited attributes.

A nonterminal A can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree.

- A **synthesized attribute** for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head.

  A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
- An **inherited attribute** for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body.

  An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N. However, a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself.

Production	Semantic Rules
1) $L \rightarrow E$ **n**	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Fig 5.1: Syntax Directed Definition of simple desk calculator

Example 5.1: The SDD in Fig. 5.1 is based on grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an endmarker n. In the SDD,

each of the nonterminals has a single synthesized attribute, called val. We also suppose that the terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer.

An SDD that involves only synthesized attributes is called **S-attributed**; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

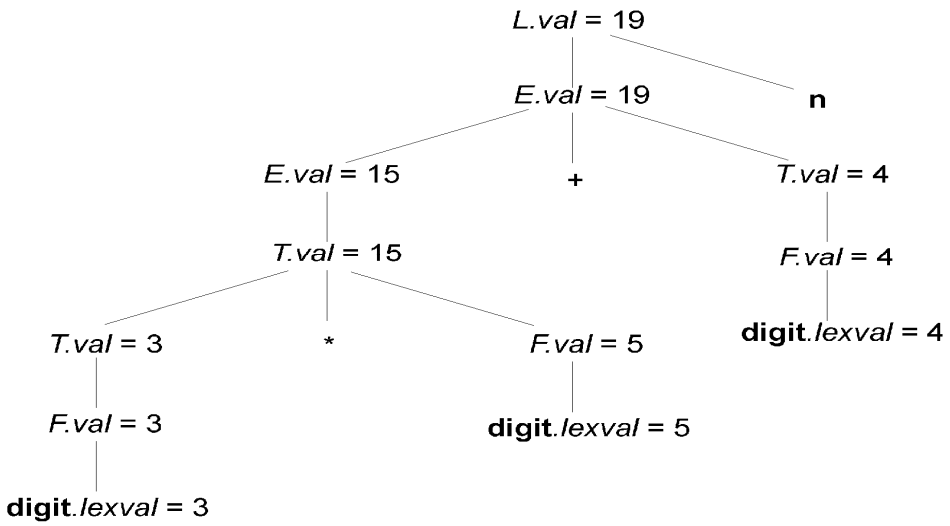
Attribute Grammar: An SDD without side effects is sometimes called an attribute grammar. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

Evaluating an SDD at the Nodes of a Parse Tree

Parse tree helps us to visualize the translation specified by SDD. The rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree.**

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

Example: Annotated Parse Tree for 3*5+4n



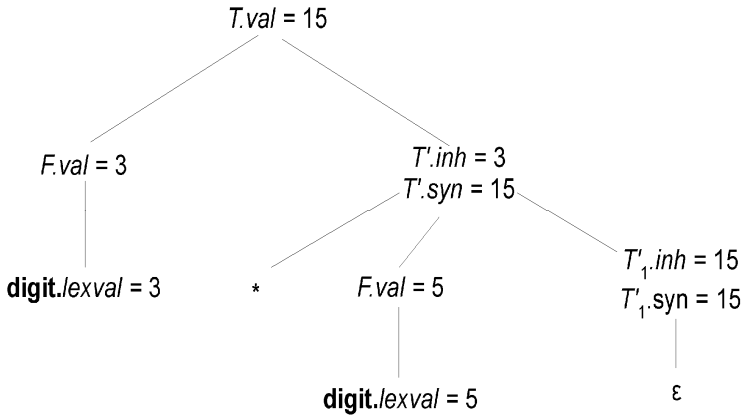
Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code. They can be used to overcome the mismatch due to grammar designed for parsing rather than translation.

In the SDD below, the nonterminal T′ has an inherited attribute *inh* as well as a synthesized attribute val. T′ inherits F.val from its left sibling F in the production $T \rightarrow F\ T'$.

Production	Semantic Rules
$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

SDD for expression grammar with inherited attributes

Annotated Parse Tree for 3*5 using the above SDD is as below.



An SDD with both inherited and synthesized attributes does not ensure any guaranteed order; even it may not have an order at all. For example, consider nonterminals A and B, with synthesized and inherited attributes A.s and B.i, respectively, along with the production and rules as in Fig.5.2. These rules are circular; it is impossible to evaluate either A.s at a node N or B.i at the child of N without first evaluating the other. The circular dependency of A.s and B.i at some pair of nodes in a parse tree is suggested by Fig.5.2.

e.g.

Production	Semantic Rules
$A \rightarrow B$	$A.s = B.i$ $B.i = A.s + 1$

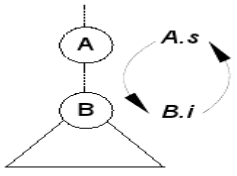


Fig 5.2: The circular dependency of *A.s* and *B.i* on one another

Evaluation Orders for SDD's

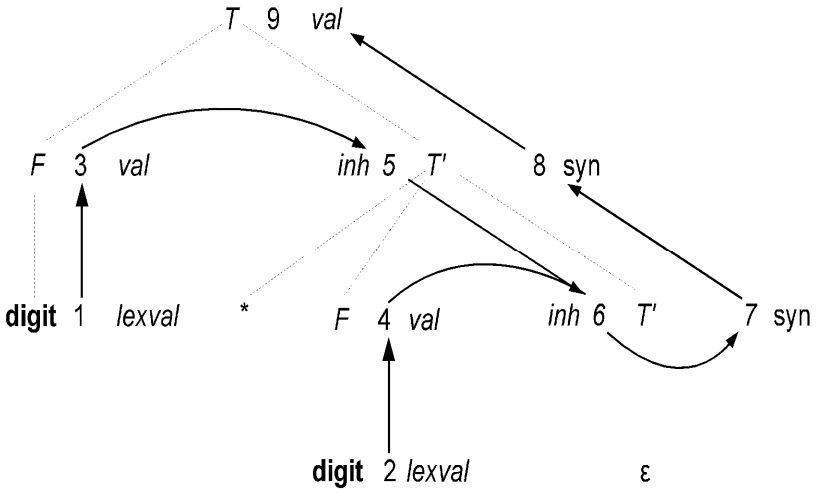
"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

A dependency graph shows the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value

of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

- Each attribute is associated to a node
- If a semantic rule associated with a production *p defines the* value of synthesized attribute *A.b in terms of the value of X.c, then graph has an edge from X.c to A.b*
- If a semantic rule associated with a production *p defines the* value of inherited attribute *B.c in terms of value of X.a, then* graph has an edge from *X.a to B.c*

Example: Dependency graph for the annotated parse tree for 3*5

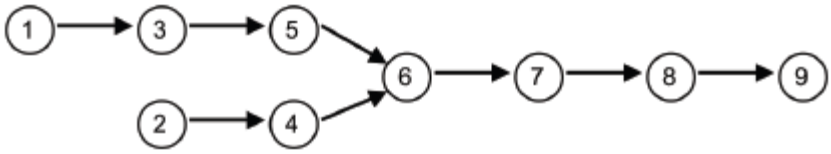


Topological Sort of the Dependency Graph

A dependency graph characterizes the possible order in which we can evaluate the attributes at various nodes of a parse tree. If there is an edge from node M to N, then attribute corresponding to M first be evaluated before evaluating N. Thus the only allowable orders of evaluation are N_1, N_2, ..., N_k such that if there is an edge from N_i to N_j then i < j. Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort.

Example: Topological sorts for the above the dependency graph is shown in a simplified fashion with the nodes indicating the nodes of the parse tree and the edges indicating precedence of evaluation. It is apparent that the last section of the dependence graph denoted by the nodes { 6 7 8 9 } needs to be evaluated in sequence. The only flexibility of the order of evaluation is on the two sub-sequences { 1 3 5 } and { 2 4 } which can with any interleaving provided the relative order of their nodes is preserved are as below.



Base sequences are {1 3 5} and {2 4} with the suffix {6 7 8 9} being constant as the last nodes of the topological sorting need to remain fixed.
1 3 5 2 4 6 7 8 9, 1 3 2 5 4 6 7 8 9, 1 2 3 5 4 6 7 8 9, 1 3 2 4 5 6 7 8 9, 1 2 3 4 5 6 7 8 9, 1 2 4 3 5 6 7 8 9, 2 1 3 5 4 6 7 8 9, 2 1 3 4 5 6 7 8 9, 2 1 4 3 5 6 7 8 9, 2 4 1 3 5 6 7 8 9

S-Attributed Definitions

An SDD is *S-attributed* if every attribute is synthesized. Attributes of an S-attributed SDD can be evaluated in bottom-up order of the nodes of parse tree. Evaluation is simple using post-order traversal.

```
postorder(N) {
        for (each child C of N, from the left)
                        postorder(C);
        evaluate attributes associated with node N;
}
```

S-attributed definitions can be implemented during bottom-up parsing as

- bottom-up parse corresponds to a postorder traversal
- postorder corresponds to the order in which an LR parser reduces a production body to its head

L-Attributed Definitions

The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left(hence "L-attributed"). Each attribute must be either

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X1\ X2 \bullet \bullet \bullet Xn$, and that there is an inherited attribute $Xi.a$ computed by a rule associated with this production. Then the rule may use only:

   (a) Inherited attributes associated with the head A.

   (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X1\ X2 \bullet \bullet \bullet Xi\text{-}1$ located to the left of Xi

   (c) Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this Xi.

Example 1: The following definition is L-attributed. Here the inherited attribute of T′ gets its values from its left sibling F. Similarly, T1′ gets its value from its parent T′ and left sibling F.

Production	Semantic Rules
$T \rightarrow F\ T'$	$T'.inh = F.val$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$

Example 2: the definitions below are not L-attributed as B.i depends on its right sibling C's attribute.

Production	Semantic Rules
$A \rightarrow B\ C$	$A.s = B.b$ $B.i = f(C.c, A.s)$

Side Effects: Evaluation of semantic rules may generate intermediate codes, may put information into the symbol table, may perform type checking and may issue error messages. These are known as side effects.

Semantic Rules with Controlled Side Effects:

In practice translation involves side effects. Attribute grammars have no side effects and allow any evaluation order consistent with dependency graph whereas translation schemes impose left-to-right evaluation and allow schematic actions to contain any program fragment.

Ways to Control Side Effects

1. Permit incidental side effects that do not disturb attribute evaluation.
2. Impose restrictions on allowable evaluation orders, so that the same translation is produced for any allowable order.

SDD For Simple Type Declarations

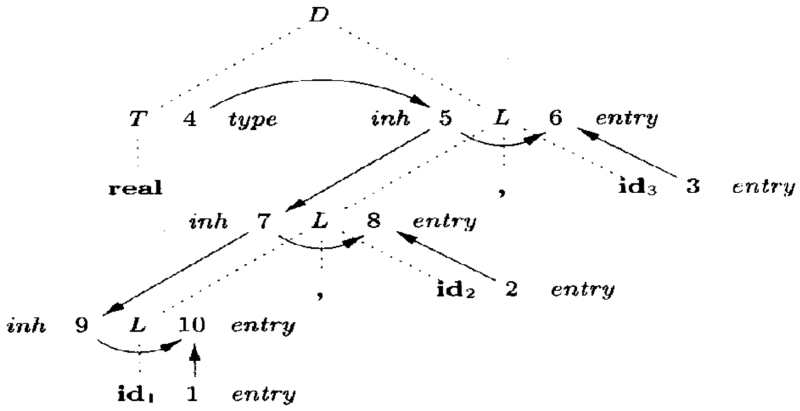
Production	Semantic Rules
1) $D \rightarrow T\ L$	$L.inh = T.type$
2) $T \rightarrow$ **int**	$T.type$ = integer
3) $T \rightarrow$ **float**	$T.type$ = float
4) $L \rightarrow L_1$, **id**	$L_1.inh = L.inh$ *addType* (**id**.*entry, L.inh*)
5) $L \rightarrow$ **id**	*addType* (**id**.*entry, L.inh*)

Nonterminal *D* represents a declaration, which, from production 1, consists of a type *T* followed by a list *L* of identifiers. *T* has one attribute, *T.type,* which is the type in the declaration *D.* Nonterminal *L* also has one attribute, which we call *ink* to emphasize that it is an inherited attribute. The purpose of *L.inh* is to pass the declared type down the list of identifiers, so that it can be the appropriate symbol-table entries. Productions 2 and 3 each evaluate the synthesized attribute *T.type,* giving it the appropriate value, integer or float. This type is passed to the attribute *L.inh* in the rule for production 1. Production 4 passes *L.inh* down the parse tree. That is, the value L1.*inh* is computed at a parse-tree node by copying the value of *L.inh* from the parent of that node; the parent corresponds to the head of the production. Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments:

1. **id.***entry,* a lexical value that points to a symbol-table object, and
2. *L.inh,* the type being assigned to every identifier on the list.

The function *addType* properly installs the type *L.inh* as the type of the represented identifier. Note that the side effect, adding the type info to the table, does not affect the evaluation order.

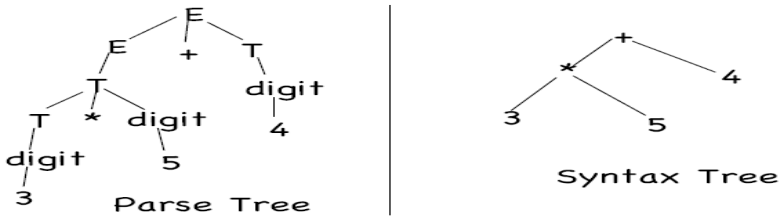
A dependency graph for the input string **float id1 , id 2, id3** is shown below.



Applications of Syntax-Directed Translations

1: Construction of Syntax Trees

SDDs are useful for is construction of syntax trees. A syntax tree is a condensed form of parse tree.



- Syntax trees are useful for representing programming language constructs like expressions and statements.
- They help compiler design by decoupling parsing from translation.
- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
    - e.g. a syntax-tree node representing an expression *E1 + E2 has* label + and two children representing the sub expressions *E1 and E2*
- Each node is implemented by objects with suitable number of fields; each object will have an *op field that is the label of the node with* additional fields as follows:

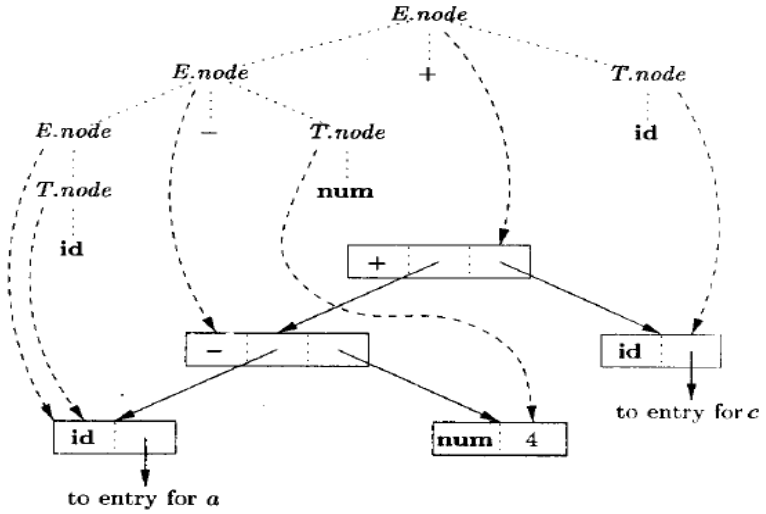
● If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function **Leaf(op, val)**

● If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function **Node(op, c1, c2,...,ck)** .

Example: The S-attributed definition in figure below constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node,* which represents a node of the syntax tree.

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	*E.node* = **new** *Node* ('+', E_1.*node*, *T.node*)
2) $E \rightarrow E_1 - T$	*E.node* = **new** *Node* ('−', E_1.*node*, *T.node*)
3) $E \rightarrow T$	*E.node* = *T.node*
4) $T \rightarrow (E)$	*E.node* = *T.node*
5) $T \rightarrow$ **id**	*T.node* = **new** *Leaf* (**id**, **id**.*entry*)
6) $T \rightarrow$ **num**	*T.node* = **new** *Leaf* (**num**, **num**.*val*)

Syntax tree for a-4+c using the above SDD is shown below.



Steps in the construction of the syntax tree for a-4+c

If the rules are evaluated during a post order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

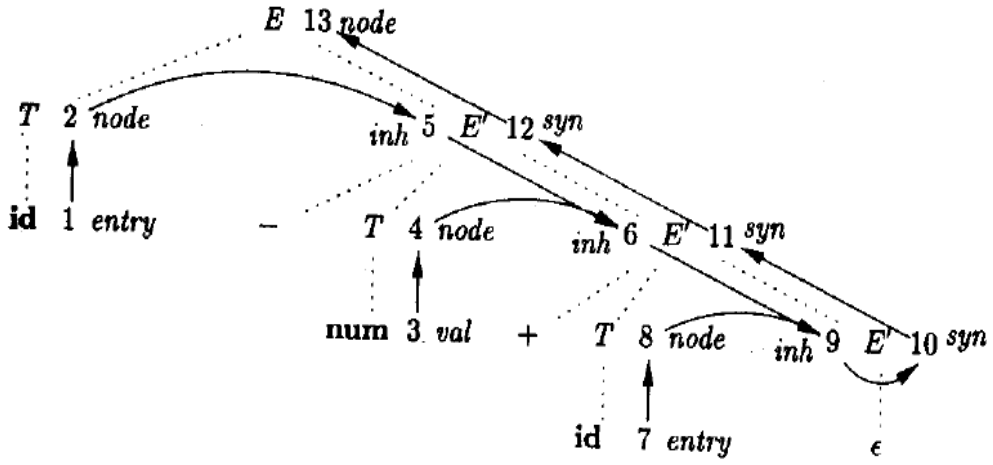
1) p_1 = **new** *Leaf* (**id**, *entry-a*);
2) p_2 = **new** *Leaf* (**num**, 4);
3) p_3 = **new** *Node* ('−', p_1, p_2);
4) p_4 = **new** *Leaf* (**id**, *entry-c*);
5) p_5 = **new** *Node* ('+', p_3, p_4);

Constructing Syntax Trees during Top-Down Parsing

With a grammar designed for top-down parsing, the same syntax trees areconstructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees. The L-attributed definition below performs the same translation as the S-attributed definition shown before.

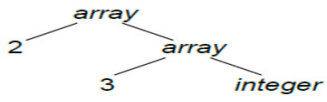
Production	Semantic Rules
1) $E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2) $E' \rightarrow +T\ E_1'$	$E_1'.inh$ = **new** *Node* ('+', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
3) $E' \rightarrow -T\ E_1'$	$E_1'.inh$ = **new** *Node* ('−', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
4) $E' \rightarrow \varepsilon$	$E'.syn = E'.inh$
5) $T \rightarrow (\ E\ )$	$T.node = E.node$
6) $T \rightarrow$ **id**	$T.node$ = **new** *Leaf* (**id**, **id**.*entry*)
7) $T \rightarrow$ **num**	$T.node$ = **new** *Leaf* (**num**, **num**.*val*)

Dependency Graph for a-4+c with L-Attributed SDD



Structure of a Type

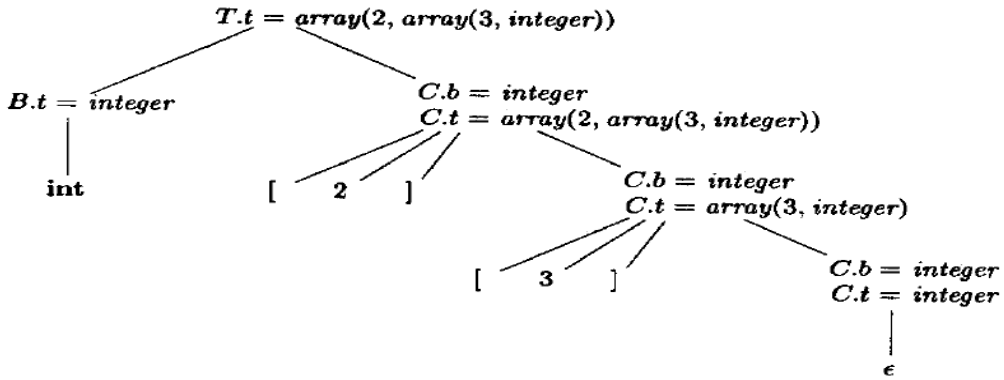
This is an example of how inherited attributes can be used to carry information one part of the parse tree to another. In C, the type int [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression array(2, array(3, integer)) is represented by the tree as shown below.



Production	Semantic Rules
1) $T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
2) $B \rightarrow$ **int**	$B.t = integer$
3) $B \rightarrow$ **float**	$B.t = float$
4) $C \rightarrow$ **[num]** C_1	$C.t = array$ (**num**.*val*, $C_1.t$) $C_1.b = C.b$
5) $C \rightarrow \varepsilon$	$C.t = C.b$

The nonterminals B and T have a synthesized attribute t representing a type. The nonterminal C has two attributes: an inherited attribute b and a synthesized attribute t. The inherited b attributes pass a basic type down the tree, and the synthesized t attributes accumulate the result.

An annotated parse tree for the input string **int [2][3]** is shown below. The corresponding type expression is constructed by passing the type integer from B, down the chain of C's through the inherited attributes b. The array type is synthesized up the chain of C's through the attributes t.



Syntax Directed Translation Schemes

SDT is a complementary notation to SDD. All applications of SDD can be implemented using SDT. SDT is a context-free grammar with program fragments called semantic actions embedded within production bodies.

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order i.e. during a pre-order traversal. Typically SDT's are implemented during parsing without building parse tree. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of action have been matched.

SDT's that can be implemented during parsing can be characterized by introducing distinct *marker nonterminals in* place of each embedded action. Each marker *M has only one production* $M \rightarrow \varepsilon$. If grammar with marker nonterminals can be parsed by a given method, then the SDT can be implemented during parsing.

Postfix Translation Schemes

The simplest implementation of SDD occurs when we can parse the grammar bottom-up and SDD is S-attributed. Here, each semantic action can be placed at the end of production and executed along with the reduction of body to the head of the production. This type of SDT is called *Postfix SDT*.

Example: Postfix SDT for implementing the desk calculator is as below.

$L \rightarrow E$ **n**	{ print (*E.val*); }
$E \rightarrow E_1 + T$	{ *E.val* = E_1.*val* + *T.val*; }
$E \rightarrow T$	{ *E.val* = *T.val*; }
$T \rightarrow T_1 * F$	{ *T.val* = T_1.*val* x *F.val*; }
$T \rightarrow F$	{ *T.val* = *F.val*; }
$F \rightarrow (E)$	{ *F.val* = *E.val*; }
$F \rightarrow$ **digit**	{ *F.val* = **digit**.*lexval*; }

Parser-Stack Implementation of Postfix SDTs

Postfix SDT's can be implemented during LR parsing. Attribute of each grammar symbol can be put on the stack in place where they can be found during reduction, i.e.. place attribute along with grammar symbol in record of stack itself.

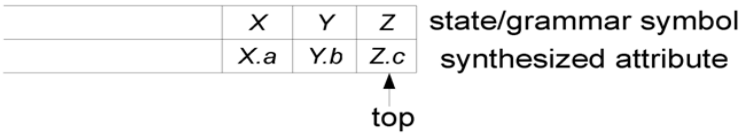


Fig. parser stack with field for synthesised attributes

To implement SDT during LR parsing, add semantic stack parallel to the parsing stack: each symbol (terminal or non-terminal) on the parsing stack stores its value on the semantic stack. It holds terminals' attributes and nonterminals' translations. When the parse is finished, the semantic stack will hold just one value: the translation of the root non-terminal (which is the translation of the whole input).

Semantic Actions during Parsing

- when shifting
  - push the value of the terminal on the semantic stack
- when reducing
  - pop k values from the semantic stack, where k is the number of symbols on production's RHS
  - push the production's value on the semantic stack

Production	Actions
$L \rightarrow E$ **n**	{ print (*stack* [*top* -1].*val*); *top* = *top* - 1 ;}
$E \rightarrow E_1 + T$	{ *stack* [*top* - 2].*val* = *stack*[*top* - 2].*val* + *stack*[*top*].*val*; *top* = *top* - 2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack* [*top* - 2].*val* = *stack*[*top* - 2].*val* x *stack*[*top*].*val*; *top* = *top* - 2; }
$T \rightarrow F$	
$F \rightarrow (E)$	{ *stack* [*top* - 2].*val* = *stack*[*top* - 1].*val; top* = *top* - 1; }
$F \rightarrow$ **digit**	

The SDT for implementing the desk calculator on a bottom-up parsing stack is as above.

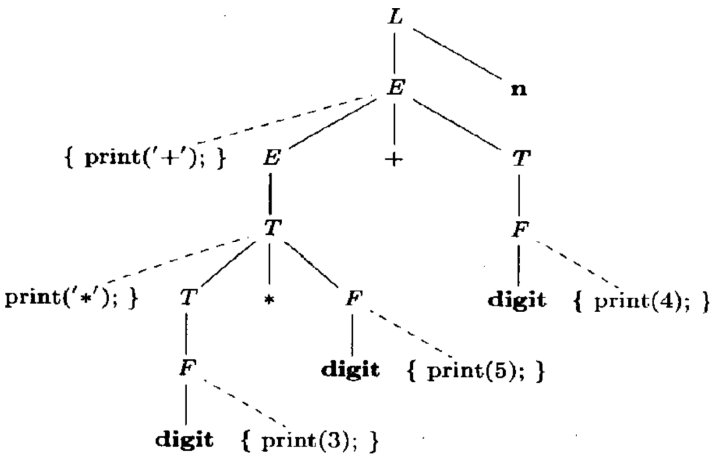
SDTs with Actions inside Productions

- Action can be placed at any position in the production body.
- Action is performed immediately after all symbols left to it are processed.
- Given B —> X { a } Y , an action a is done after
  - we have recognized X (if X is a terminal), or
  - all terminals derived from X (if X is a nonterminal).
- If bottom-up parser is used, then action a is performed as soon as X appears on top of the stack.
- If top-down parser is used, then action a is performed
  - just before Y is expanded (if Y is nonterminal), or
  - check Y on input (if Y is a terminal).
- Any SDT can be implemented as follows:
  - Ignoring actions, parse input and produce parse tree.
  - Add additional children to node N for action in α, where A → α.
  - Perform preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

SDT for infix-to-prefix translation during parsing

1) $L \rightarrow E$ **n**
2) $E \rightarrow$ { print('+'); } $E_1 + T$
3) $E \rightarrow T$
4) $T \rightarrow T_1 * F$ { print('*'); }
5) $T \rightarrow F$
6) $F \rightarrow (E)$
7) $F \rightarrow$ **digit** { print(**digit**.*lexval*); }

Parse Tree with Actions Embedded



Eliminating Left Recursion from SDTs

Grammar with left recursion cannot be parsed using top-down parser. In case of SDT we treat action as terminal symbol in the production. Then we use the following rule of transforming grammar to non left-recursive form.

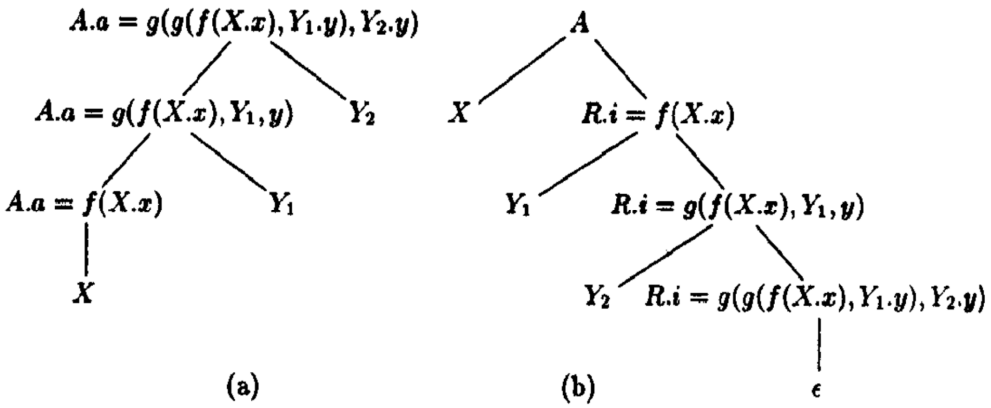
$A \rightarrow A\alpha \mid \beta$ can be transformed to $A \rightarrow \beta R$, $R \rightarrow \alpha R \mid \varepsilon$

In both forms, A is defined by $\beta(\alpha)^*$

Example: Given $E \rightarrow E_1 + T$ { print('+'); } and $E \rightarrow T$. Here α = + T { print('+'); } and β = T . Modified grammar will be: $E \rightarrow T\ R$, $R \rightarrow +\ T$ { print('+'); } R , $R \rightarrow \varepsilon$.

More general example: $A \rightarrow A_1\ Y$ {A.a=g(A_1.a, Y.y)}
$A \rightarrow X$ {A.a = f(X.x)} can be rewritten as

$A \rightarrow X \{ R.i = f(X.x) \}\ R \{A.a = R.s\}$
$R \rightarrow Y \{ R_1.i = g(R.i, Y.y) \}\ R_1 \{ R.s = R_1.s \}$
$R \rightarrow \varepsilon \{ R.s = R.i \}$



SDTs for L-Attributed Definitions

- It is necessary that the underlying grammar can be parsed top-down
- Rules to modify L-attributed SDD to SDT
  - Embed the action for computing inherited attributes for a nonterminal A immediately before A in production body
  - Place the action for computing synthesized attribute for the head at the end of the body of production

Example 1: This example is motivated by languages for typesetting mathematical formulas. Eqn is an early example of such a language. It illustrates how the techniques of compiling can be used in language processing for applications other than programming languages.

Typesetting

- Eqn language : a sub i sub j => a_{ij}
- Simple grammar for boxes
  $B \rightarrow B_1\ B_2 \mid B_1$ **sub** $B_2 \mid (\ B_1\ ) \mid$ **text**

Corresponding to these four productions, a box can be either
1. Two boxes, juxtaposed, with the first, B1, to the left of the other, B2.
2. A box and a subscript box. The second box appears in a smaller size, lower, and to the right of the first box.
3. A parenthesized box, for grouping of boxes and subscripts.
4. A text string, that is, any string of characters.

Figure 5.24: Constructing larger boxes from smaller ones

SDD for Typesetting Boxes

$S \rightarrow B$	$B.ps = 10$
$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
$B \rightarrow (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
$B \rightarrow$ **text**	$B.ht = getHt(B.ps,$ **text**$.lexval)$ $B.dp = getDp(B.ps,$ **text**$.lexval)$

SDT for typesetting boxes

$S \rightarrow$ B	{ $B.ps = 10$; }
$B \rightarrow$ B_1 B_2	{ $B_1.ps = B.ps$; } { $B_2.ps = B.ps$; } { $B.ht = \max(B_1.ht, B_2.ht)$; $B.dp = \max(B_1.dp, B_2.dp)$; }
$B \rightarrow$ B_1 **sub** B_2	{ $B_1.ps = B.ps$; } { $B_2.ps = 0.7 \times B.ps$; } { $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$; $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$; }
$B \rightarrow$ (B_1)	{ $B_1.ps = B.ps$; } { $B.ht = B_1.ht$; $B.dp = B_1.dp$; }
$B \rightarrow$ **text**	{ $B.ht = getHt(B.ps,$ **text**$.lexval)$; $B.dp = getDp(B.ps,$ **text**$.lexval)$; }

Example 2: The second example is about the generation of intermediate code for a typical programming-language construct: a form of while-statement.

SDD for while-statements

$S \rightarrow$ **while** (C) S_1 L_1 = *new* ();
L_2 = *new* ();
S_1.*next* = $L1$;
C.false = *S.next*;
C.true = $L2$;
S.code = **label** || $L1$ || *C.code*
|| **label** || $L2$ || S_1.*code*

SDT for while-statements

$S \rightarrow$ **while** ({ L_1 = *new* (); L_2 = *new* ();
C.false = *S.next*; *C.true* = $L2$; }
C) { S_1.*next* = $L1$; }
S_1 { *S.code* = **label** || $L1$ || *C.code*
|| **label** || $L2$ || S_1.*code*; }