Syntax directed translation

SDT

- Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser.

- Same techniques can be used for semantic analysis
  - Type checking

- A common method of Syntax-directed translation is translating a string into a sequence of actions by attaching such actions to productions of the grammar.

- Thus, parsing a string of the grammar produces a sequence of rule applications. SDT provides a simple way to attach semantics to any such syntax.

Syntax-Directed Definition (SDT)

- A syntax-directed definition (SDT) is a CFG along with attributes and rules/actions.
  - Attribute could be a value, a reference, or a table, or a string (intermediate representation).
  - Rule relates these attributes.
  - Action is fired (executed) to produce some output.

- Attributes are attached to grammar symbols.
- Semantic rules/actions are associated with productions.
- A syntax-directed definition relates attributes of grammar symbols along with giving actions to be executed.

Example

- Infix to postfix translator might have a production and rule like

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

- Where $code$ is an attribute which is string valued.
- The semantic rule says that $E.code$ is formed by concatenating $E_1.code, T.code,$ and the character $'+'$.

- Semantic action can be a piece of code embedded in to the body of a production, which is executed at a specific point of parsing.

$$E \rightarrow E_1 + T \; \{print \; `+'\}$$

- Semantic rule is easy to understand than semantic actions
  - The order where the semantic action is placed in the production determines when the action is executed.
  - In the above example, the action is done if the production is applied.

Semantic actions that can produce postfix output in bottom-up parsing

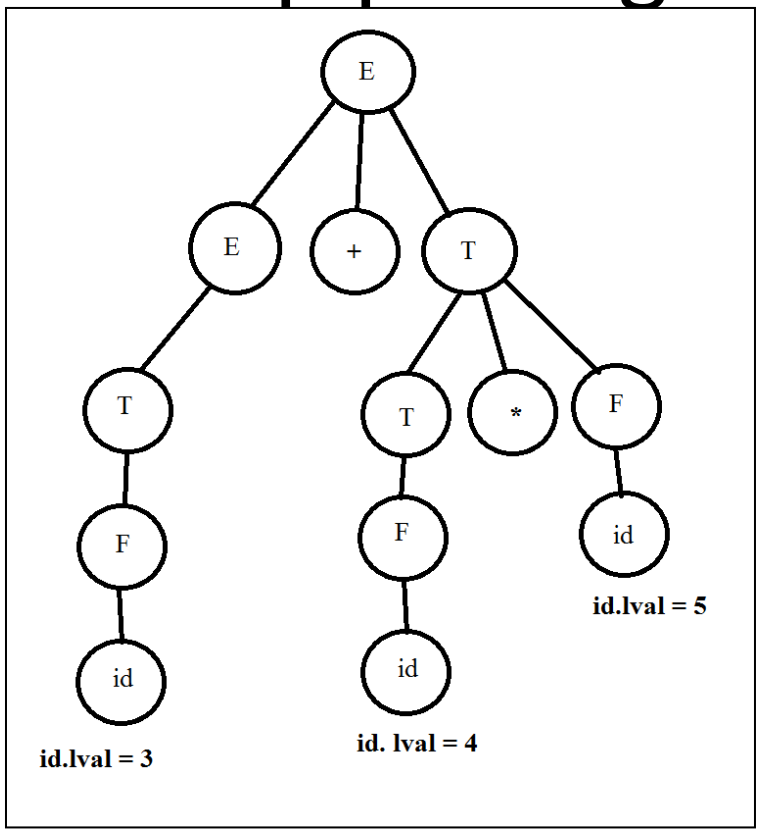
E->E+T { print('+') }

| T { }

T -> T*F { print('*') }

| F { }

F -> ***id*** {print(***id***.lval)}



Bottom-up parsing, whenever a reduction is done, will take the given action.

Semantic actions that can produce postfix output in bottom-up parsing

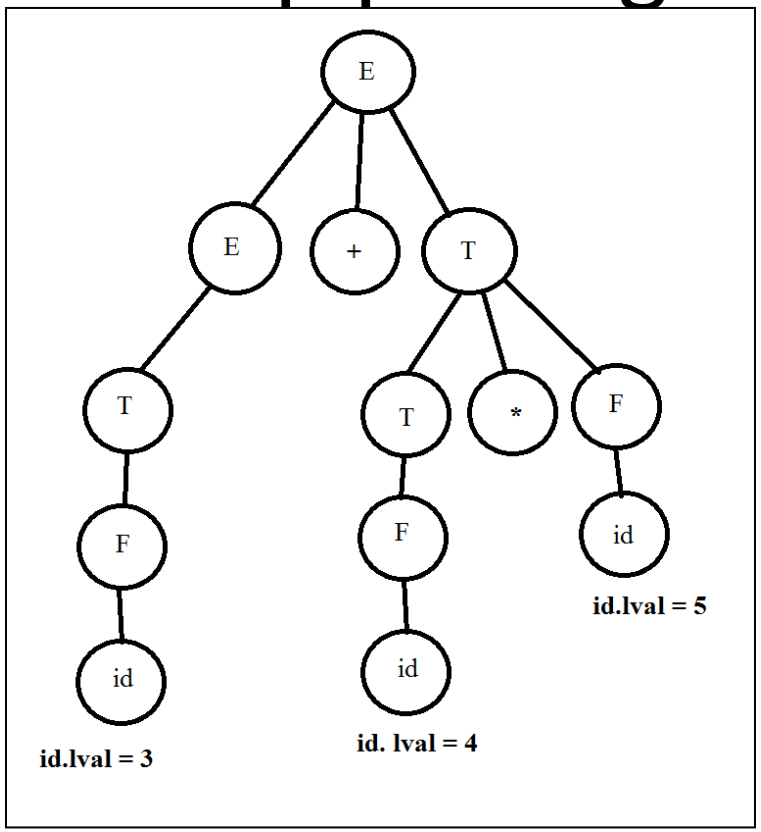
E->E+T { print('+') }

| T { }

T -> T*F { print('*') }

| F { }

F -> ***id*** {print(***id***.lval)}



Work with the example: 3+4*5

It should output: 3 4 5 * +

A Question

S -> xxW {print"1"}
S -> y {print"2"}
W -> Sz {print"3"}

Given xxxxyzz the output is

- (A) 11231
- (B) 11233
- (C) 23131
- (D) 33211

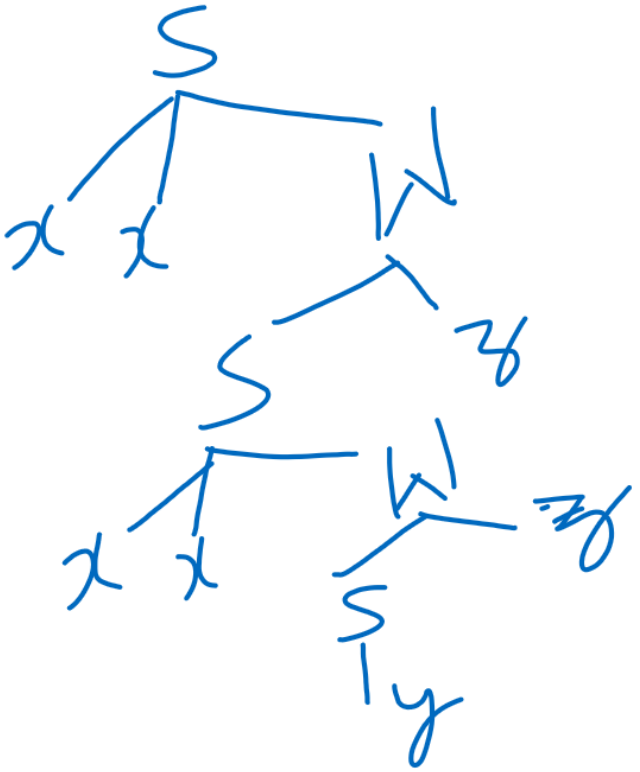
Build parse tree and follow the LR parsing

A Question

S -> xxW {print"1"}
S -> y {print"2"}
W -> Sz {print"3"}

Given xxxxyzz the output is

- (A) 11231
- (B) 11233
- (C) 23131
- (D) 33211



Build parse tree and follow the LR parsing

A Question

S -> xxW {print”1”}
S -> y {print”2”}
W -> Sz {print”3”}

Given xxxxyzz the output is

- (A) 11231
- (B) 11233
- (C) 23131
- (D) 33211

- ANSWER: 23131

- General approach: construct the parse tree and visit the nodes and evaluate attributes at nodes.
- But, there is no need to explicitly build the parse tree.
  - During parsing, as a side effect the translation can be done.

Attributes: Two types

- Synthesized attributes
  - A synthesized attribute at node N is defined only in terms of attribute values of children of N and attribute values of N itself.
- Inherited attributes
  - An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings.

✓ A grammar symbol might have only one type of attribute(s), or both types of attribute(s).

Terminals can have only synthesized attributes

Terminals can have only synthesized attributes.

Eg: ***id*** can have ***id.lval***

These are provided by the LA {No semantic rules/actions to compute these}

Example of S-attributed SDD

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Figure 5.1: Syntax-directed definition of a simple desk calculator

S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

Annotated parse tree

- A parse tree showing values of attributes is called annotated parse tree.

	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

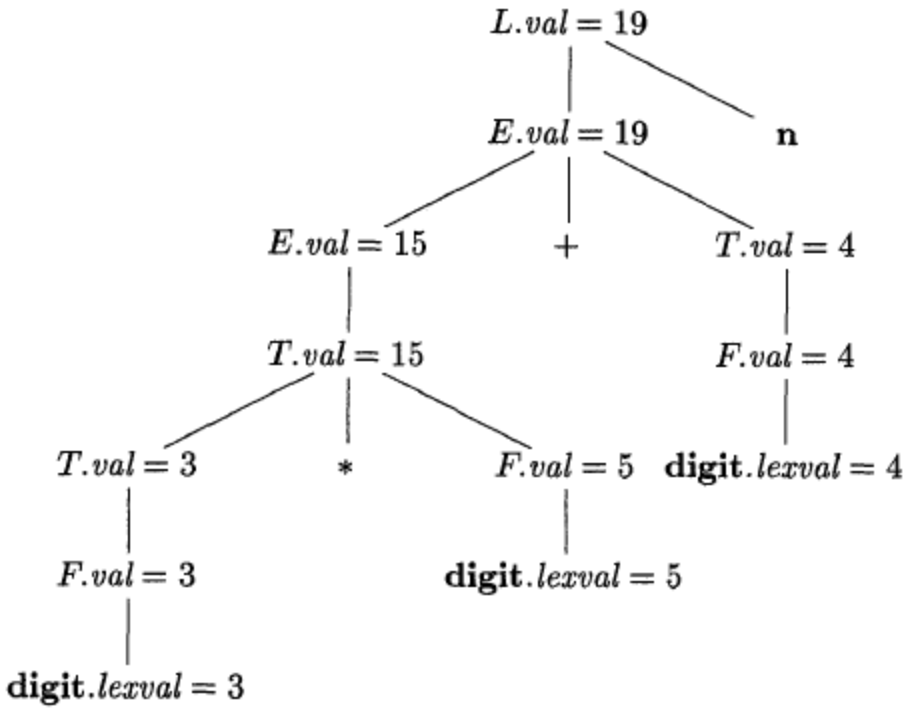


Figure 5.3: Annotated parse tree for $3 * 5 + 4$ **n**

SDD for post-fix conversion

$$E \rightarrow E_1 + T \quad \{E.code = E_1.code \,||\, T.code \,||\, '+'\}$$

$$E \rightarrow T \{E.code = T.code\}$$

$$T \rightarrow T_1 * F \quad \{T.code = T_1.code \,||\, F.code \,||\, '*'\} \,|$$

$$F \{T.code = F.code\}$$

$$F \rightarrow id \quad \{F.code = id.lexval\}$$

Problem: Find the code value of the root when the given input string is 2 + 3 * 4

SDD for post-fix conversion

$E \rightarrow E_1 + T \quad \{E.code = E_1.code \,||\, T.code \,||\, '+'\}$

$E \rightarrow T \{E.code = T.code\}$

$T \rightarrow T_1 * F \quad \{T.code = T_1.code \,||\, F.code \,||\, '*'\} \,|$

$\qquad F \{T.code = F.code\}$

$F \rightarrow id \quad \{F.code = id.lexval\}$

Note, here we are not printing anything. Only attribute values are found. It so happened that the attribute value of the root is the desired result.

! **Exercise 5.2.4:** This grammar generates binary numbers with a "decimal" point:

$$S \rightarrow L \ . \ L \mid L$$
$$L \rightarrow L \ B \mid B$$
$$B \rightarrow 0 \mid 1$$

Design an L-attributed SDD to compute $S.val$, the decimal-number value of an input string. For example, the translation of string `101.101` should be the decimal number 5.625. *Hint*: use an inherited attribute $L.side$ that tells which side of the decimal point a bit is on.

!! **Exercise 5.2.5:** Design an S-attributed SDD for the grammar and translation described in Exercise 5.2.4.

S-attributed SDD for binary number conversion to decimal number

$$N \rightarrow L_1 \bullet L_2 \{N.dv = L_1.dv + L_2.dv / 2^{L_2 \cdot c}\}$$

$$L \rightarrow L_1 B \{L.c = L_1.c + 1; L.dv = 2 \times L_1.dv + B.dv\} \mid$$

$$B\{L.c = B.c; L.dv = B.dv\}$$

$$B \rightarrow 0\{B.c = 1; B.dv = 0\} \mid$$

$$1\{B.c = 1; B.dv = 1\}$$

Question

Consider the grammar with the following translation rules and E as the start symbol.

E -> E1 #T {E.value = E1.value * T.value}

| T {E.value = T.value}

T -> T1 & F {T.value = T1.value + F.value}

|F {T.value= F.value}

F -> num {F.value = num.value}

Compute E.value for the root of the parse tree for the expression:2 # 3 & 5 # 6 &4.

a) 200

b) 180

c) 160

d) 40

Consider the grammar with the following translation rules and E as the start symbol.

E -> E1 #T {E.value = E1.value * T.value}

| T {E.value = T.value}

T -> T1 & F {T.value = T1.value + F.value}

|F {T.value= F.value}

F -> num {F.value = num.value}

Compute E.value for the root of the parse tree for the expression:2 # 3 & 5 # 6 &4.

a) 200

b) 180

c) 160

d) 40

Answer: (c)

Symbols with more than one type of attribute

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

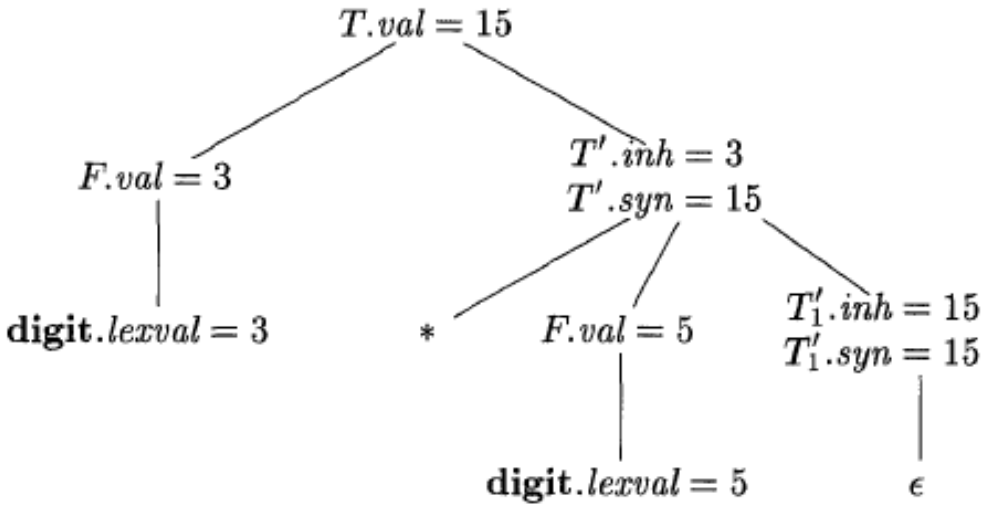


Figure 5.5: Annotated parse tree for $3 * 5$

5.2 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.

S attributed SDD

Example 5.4 : Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 \ + \ T$	$E.val = E_1.val + T.val$

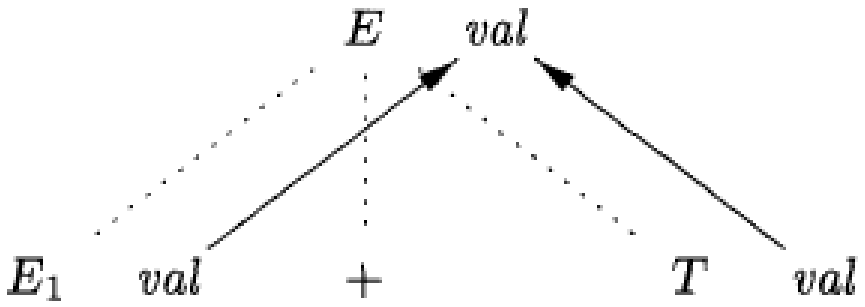


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

Attributes: values, dependency order.

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

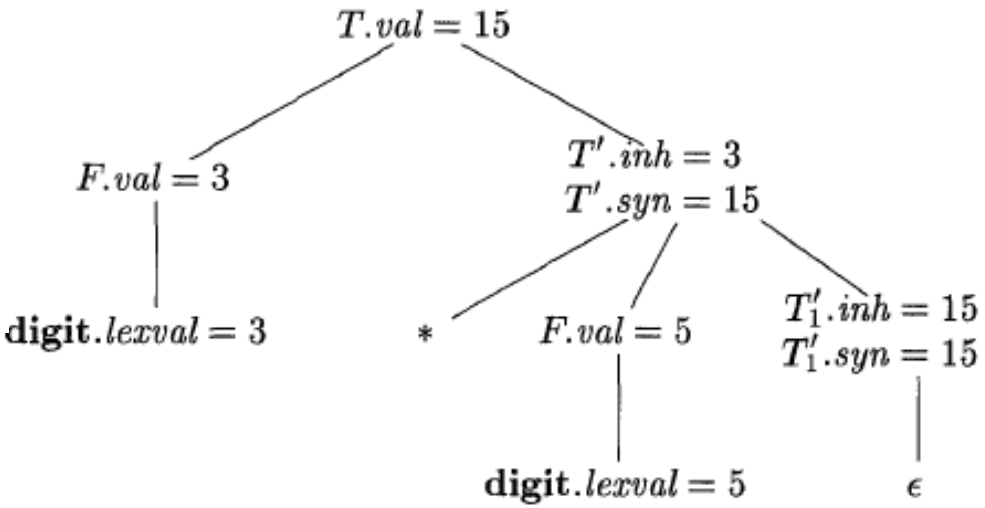


Figure 5.5: Annotated parse tree for $3 * 5$

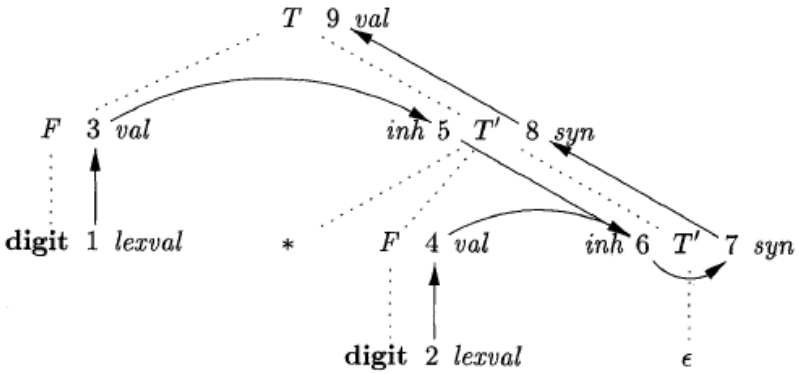


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

Circular dependency should not be present in an L-attributed definition

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$
	$B.i = A.s + 1$

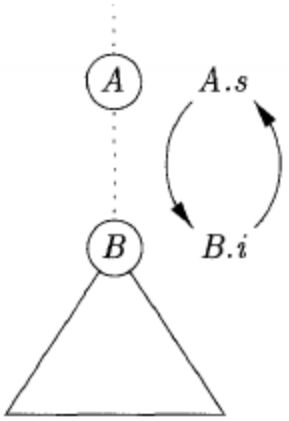


Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

Order in which attributes can be evaluated to get their values?

- Topological sort of the dependency graph gives us the evaluation order.
- If there is a loop, then there is no topological sort, hence it is not feasible to evaluate some attribute values.
- Synthesized attributes are immune to this.
- But inherited attributes, if present can fall a prey to this constraint.
  - To overcome this, L-attributed SDD is carefully defined.

Two types of syntax directed translations (SDDs)

- S-attributed translations (S-attributed SDD)
  - S for synthesized. Every attribute is synthesized.
  - Can get its value from the node itself or from children.
  - Performed easily with a bottom-up parse.
    - Whenever a reduction is done, take the appropriate action.
    - Postorder traversal of the dependency graph can also do the same.
    - Dependency graph in this case is the parse tree itself. Arrows from child to parent.
- L-attributed translations (L-attributed SDD)
  - L for left to right

S-attributed ➔ postorder is enough

When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes of the parse tree.

It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evaluating the attributes at a node N when the traversal leaves N for the last time.

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

L-attributed SDD (a generalization of S-attributed SDD)

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1 X_2 \cdots X_n$, and that there is an inherited attribute $X_i.a$ 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 $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.

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

This is L-attributed SDD

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * F\, T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

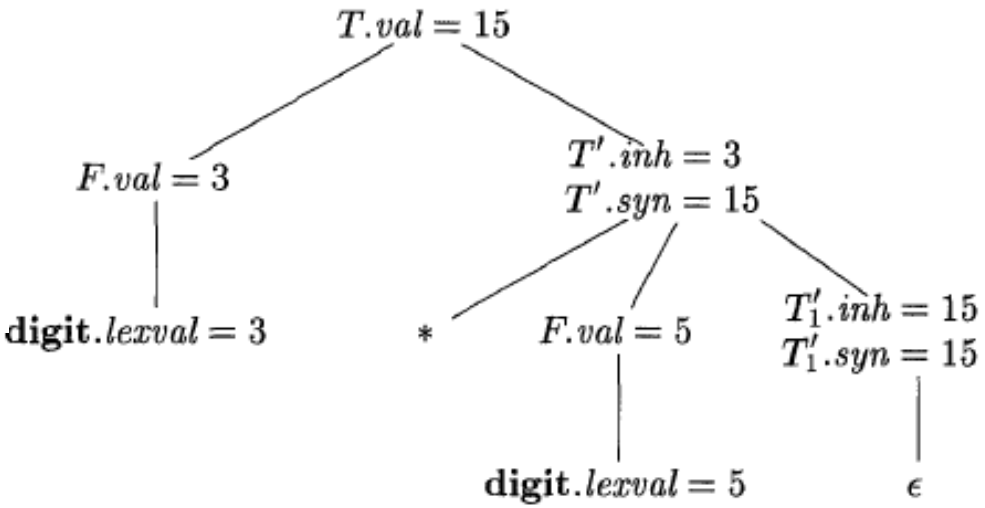


Figure 5.5: Annotated parse tree for $3 * 5$

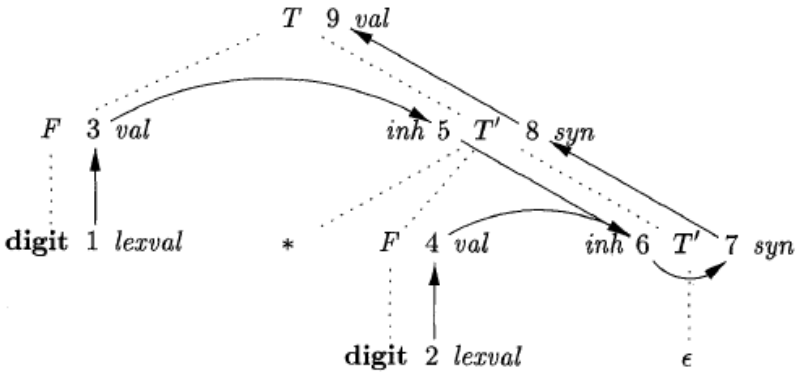


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

Example 5.9: Any SDD containing the following production and rules cannot be L-attributed:

PRODUCTION	SEMANTIC RULES
$A \rightarrow B\ C$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

Example 5.9: Any SDD containing the following production and rules cannot be L-attributed:

PRODUCTION	SEMANTIC RULES
$A \rightarrow B\ C$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

- Second rule is violating left to right constraint of L-attributed definition.

Semantic rules with side-effects

	PRODUCTION	SEMANTIC RULE
1)	$L \rightarrow E\ \mathbf{n}$	$print(E.val)$

For a desk calculator , the above semantic rule is associating a dummy attribute with the head of the production and its value is the action *print(E.val)*.

When this production is used, *E.val* is printed.

There is a dummy attribute associated with L

And this dummy attribute's value is *print(E.val)*

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$
		$addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

Figure 5.8: Syntax-directed definition for simple type declarations

addType(id.entry, L.inh) will add to the symbol table the type of id.

	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**	$L_1.inh = L.inh$
		$addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow$ **id**	$addType(\mathbf{id}.entry, L.inh)$

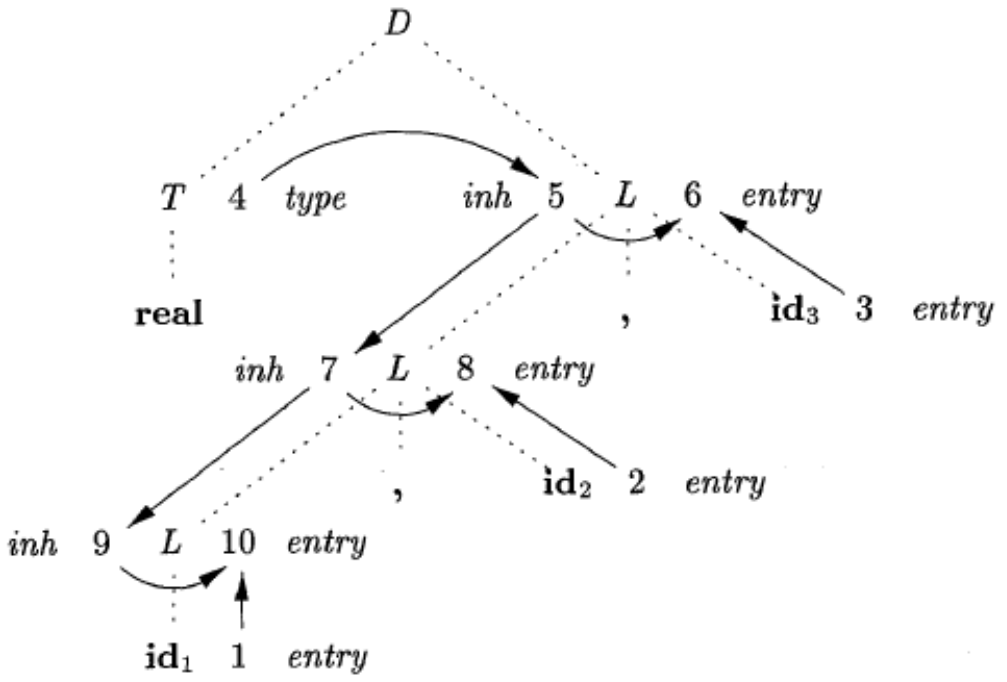


Figure 5.9: Dependency graph for a declaration **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Applications of syntax directed translations

- Construction of syntax trees (or abstract syntax trees)
- Type checking
- Intermediate code generation

5.3.1 Construction of Syntax Trees

5.3.1 Construction of Syntax Trees

A syntax-tree node representing an expression $E_1 + E_2$ has label + and two children representing the subexpressions E_1 and E_2.

5.3.1 Construction of Syntax Trees

A syntax-tree node representing an expression $E_1 + E_2$ has label + and two children representing the subexpressions E_1 and E_2.

Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

5.3.1 Construction of Syntax Trees

A syntax-tree node representing an expression $E_1 + E_2$ has label + and two children representing the subexpressions E_1 and E_2.

Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function *Leaf*(*op*, *val*) creates a leaf object.

5.3.1 Construction of Syntax Trees

A syntax-tree node representing an expression $E_1 + E_2$ has label + and two children representing the subexpressions E_1 and E_2.

Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function *Leaf*(*op*, *val*) creates a leaf object.

- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function *Node* takes two or more arguments: $Node(op, c_1, c_2, \ldots, c_k)$ creates an object with first field *op* and k additional fields for the k children $c_1, \ldots, c_k$.

	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)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**, **id**.$entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**, **num**.$val)$

Figure 5.10: Constructing syntax trees for simple expressions

Syntax tree for $a - 4 + c$

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

Syntax tree for $a - 4 + c$

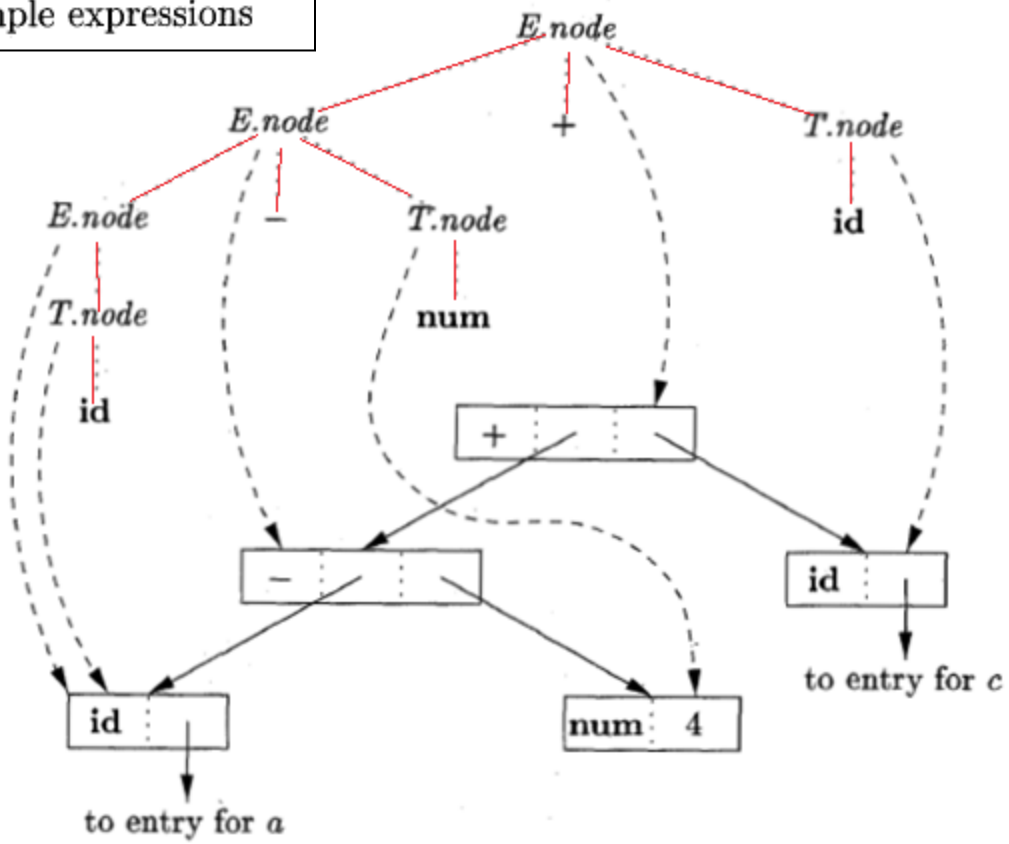
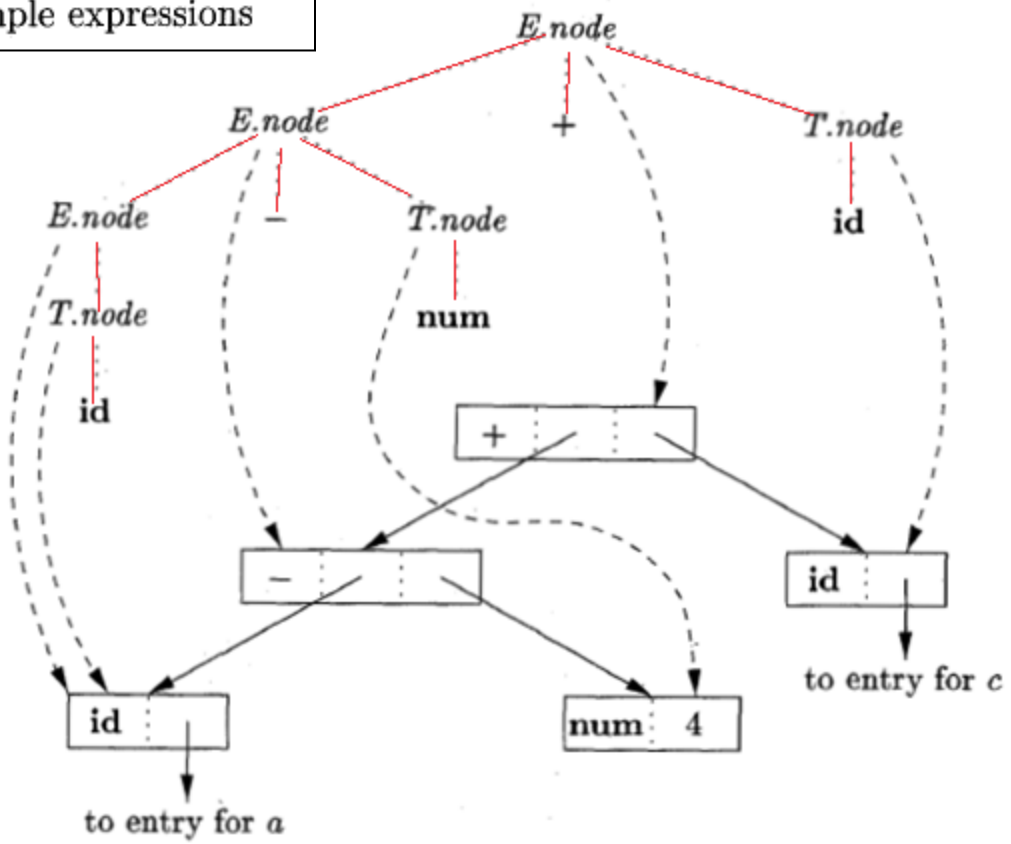


Figure 5.11: Syntax tree for $a - 4 + c$

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

Syntax tree for $a - 4 + c$



1) $p_1 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new}\ Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ Node('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new}\ Node('+', p_3, p_4);$

Steps followed in constructing the syntax tree

Figure 5.11: Syntax tree for $a - 4 + c$

Syntax tree construction in top down parsing

- Top down parsing, eliminates left recursion. Hence a new grammar is created.

Syntax tree construction in top down parsing

- Top down parsing, eliminates left recursion. Hence a new grammar is created.

	PRODUCTION
1)	$E \rightarrow T\ E'$
2)	$E' \rightarrow +\ T\ E'_1$
3)	$E' \rightarrow -\ T\ E'_1$
4)	$E' \rightarrow \epsilon$
5)	$T \rightarrow (\ E\ )$
6)	$T \rightarrow \mathbf{id}$
7)	$T \rightarrow \mathbf{num}$

Syntax tree construction in top down parsing

- Top down parsing, eliminates left recursion. Hence a new grammar is created.

	PRODUCTION
1)	$E \rightarrow T\ E'$
2)	$E' \rightarrow +\ T\ E'_1$
3)	$E' \rightarrow -\ T\ E'_1$
4)	$E' \rightarrow \epsilon$
5)	$T \rightarrow (\ E\ )$
6)	$T \rightarrow \mathbf{id}$
7)	$T \rightarrow \mathbf{num}$

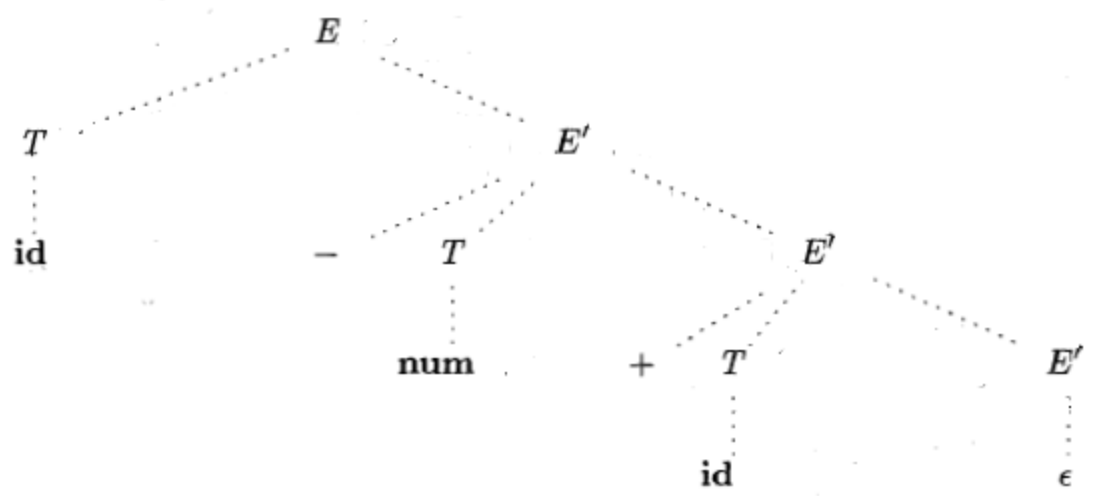
- With this, using only S attributed SDD we cannot create a syntax tree.

Hence, we use inherited attributes

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.13: Constructing syntax trees during top-down parsing

The parse tree for $a - 4 + c$



- The annotated parse tree along with order of evaluation is shown in the next slide.

The annotated parse tree along with order of evaluation

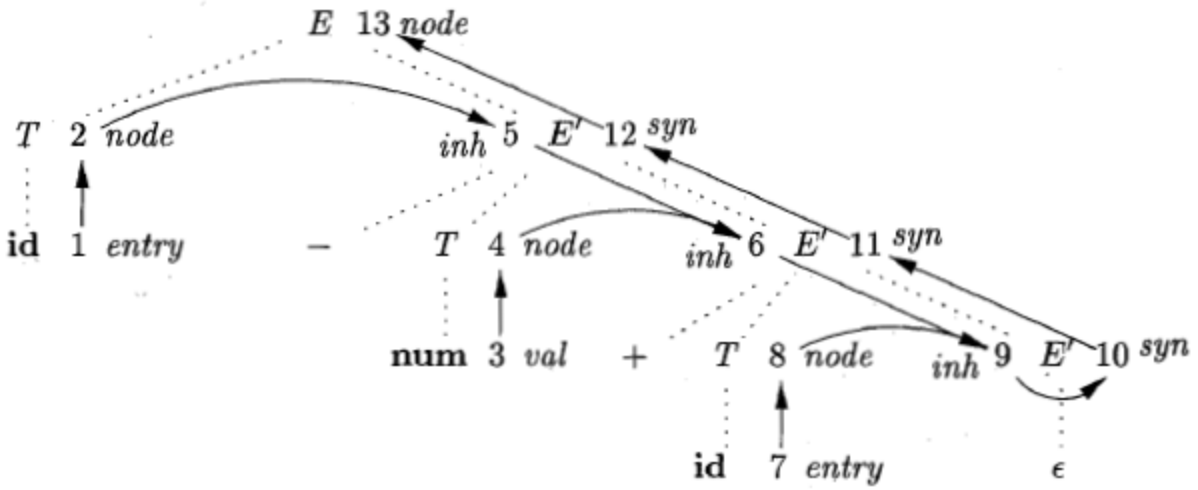


Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

- We get the same syntax tree as derived from S attributed SDD.

Type

- In C, `int [2][3]` is a type ➔it is an array of two elements, where each element is an array of 3 integers.
- The corresponding type expression is array(2,array(3,integer)). Second argument is a type, first argument is the number of elements.

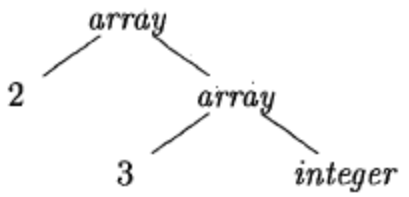


Figure 5.15: Type expression for **int**[2][3]

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \mathbf{num}\ ]\ C_1$	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

Consider **int[2][3]**

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array($**num**$.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

Consider **int[2][3]**

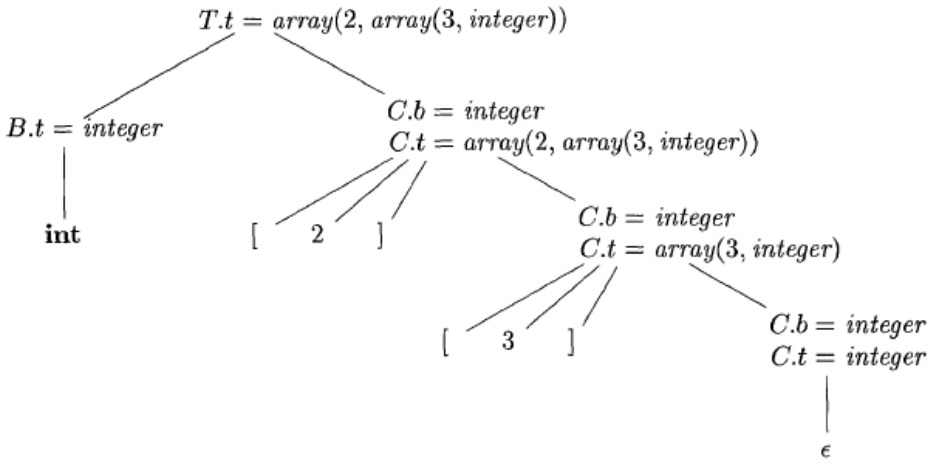


Figure 5.17: Syntax-directed translation of array types

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

Consider **int[2][3]**

To see the order of evaluation, let us give numbers to the nodes of the parse tree.

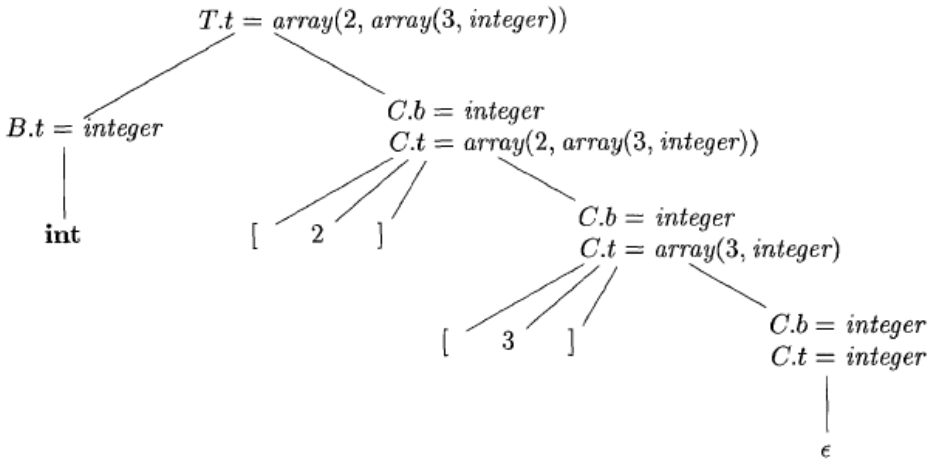


Figure 5.17: Syntax-directed translation of array types

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array($**num**$.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

Consider **int[2][3]**

Order of evaluation
B.t at 2
C.b at 3
C.b at 4
C.b at 5
C.t at 5
C.t at 4
C.t at 3
T.t at 1

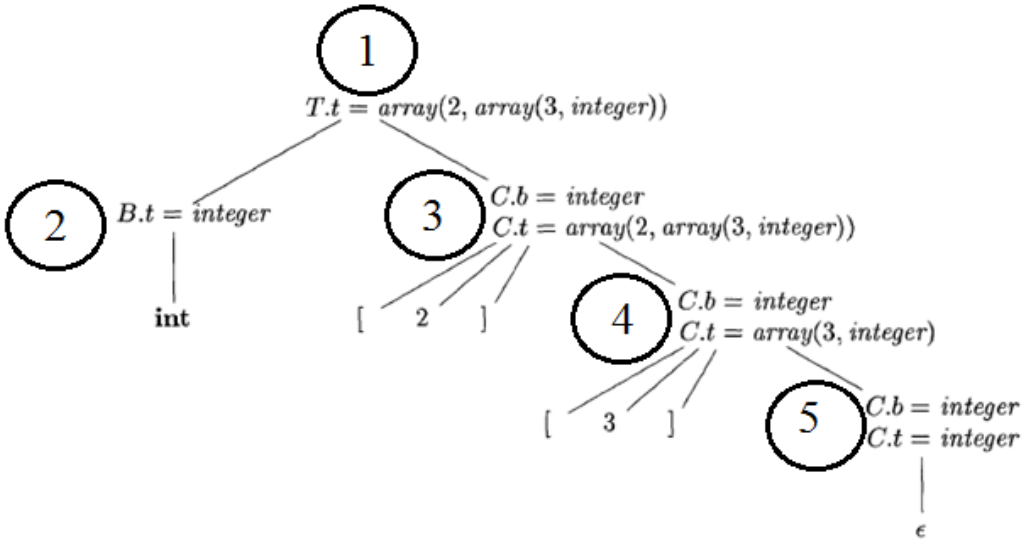


Figure 5.17: Syntax-directed translation of array types

5.4 Syntax-Directed Translation Schemes

- These are complementary ways of achieving same thing as SDDs.
- A syntex-directed translation scheme (SDT) is a CFG with program fragments embedded within production bodies.
  - These program fragments called semantic actions can appear at any position within a production body.

An example: infix to prefix

- Any SDT can be implemented by first building the parse tree and then traversing left-to-right depth-first order (i.e., preorder traversal).

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

Now, consider the input string **3*5+4**

Now, consider the input string **3*5+4**

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

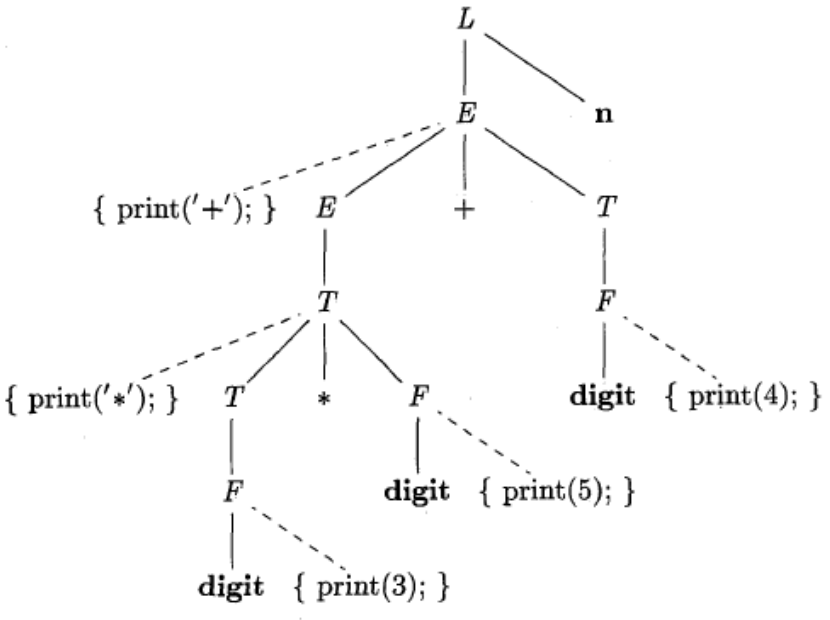


Figure 5.22: Parse tree with actions embedded

Input: 3*5+4

Output: + * 3 5 4

How SDTs are implemented

- They are implemented during parsing (without building a parse tree)
- The two classes of SDDs (viz., S-attributed and L-attributed) can be implemented, when,

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.
2. The underlying grammar is LL-parsable, and the SDD is L-attributed.

Doing SDT along with parsing?

- During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action are matched.
- This can be achieved by the usage of distinct ***marker nonterminals*** in place of each embedded action.
- Each ***marker*** *M* has only one production $M \to \epsilon$
- If the grammar with markers can be parsed, then the SDT can be done during parsing.

Example: SDT for infix to prefix translation.

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & \{\ \mathrm{print}('+');\ \}\ \ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & \{\ \mathrm{print}('*');\ \}\ \ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \mathrm{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

Example: SDT for infix to prefix translation.

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & \{\ \text{print}('+');\ \}\ \ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & \{\ \text{print}('*');\ \}\ \ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & M_2\ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & M_4\ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \} \\
M_2 & \rightarrow & \epsilon \\
M_4 & \rightarrow & \epsilon.
\end{array}
$$

Top-down parsing using stack can do the translation as a side-effect of the parsing

Example: SDT for infix to prefix translation.

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & \{\ \text{print}('+');\ \}\ \ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & \{\ \text{print}('*');\ \}\ \ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & M_2\ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & M_4\ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \} \\
M_2 & \rightarrow & \epsilon \\
M_4 & \rightarrow & \epsilon.
\end{array}
$$

Top-down parsing using stack can do the translation as a side-effect of the parsing

Can't we do with LR parsing?

Example: SDT for infix to prefix translation.

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & \{\ \text{print}('+');\ \}\ \ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & \{\ \text{print}('*');\ \}\ \ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & M_2\ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & M_4\ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \} \\
M_2 & \rightarrow & \epsilon \\
M_4 & \rightarrow & \epsilon.
\end{array}
$$

Assume some stage of parsing we got a digit as the next token, then, a shift-reduce parser has conflicts between shifting the digit, reducing by $M_2 \rightarrow \epsilon$ and reducing by $M_4 \rightarrow \epsilon$.

Example: SDT for infix to prefix translation.

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & \{\ \text{print}('+');\ \}\ \ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & \{\ \text{print}('*');\ \}\ \ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & M_2\ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & M_4\ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \} \\
M_2 & \rightarrow & \epsilon \\
M_4 & \rightarrow & \epsilon.
\end{array}
$$

Assume some stage of parsing we got a digit as the next token, then, a shift-reduce parser has conflicts between shifting the digit, reducing by $M_2 \rightarrow \epsilon$ and reducing by $M_4 \rightarrow \epsilon$.

So this SDT cannot be implemented along with LR parsing.

Example: SDT for infix to prefix translation.

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & \{\ \text{print}('+');\ \}\ \ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & \{\ \text{print}('*');\ \}\ \ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \}
\end{array}
$$

$$
\begin{array}{lcl}
L & \rightarrow & E\ \mathbf{n} \\
E & \rightarrow & M_2\ E_1 + T \\
E & \rightarrow & T \\
T & \rightarrow & M_4\ T_1 * F \\
T & \rightarrow & F \\
F & \rightarrow & (\ E\ ) \\
F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \} \\
M_2 & \rightarrow & \epsilon \\
M_4 & \rightarrow & \epsilon.
\end{array}
$$

Assume some stage of parsing we got a digit as the next token, then, a shift-reduce parser has conflicts between shifting the digit, reducing by $M_2 \rightarrow \epsilon$ and reducing by $M_4 \rightarrow \epsilon$.

So this SDT cannot be implemented along with LR parsing.

We should have used one more Marker for this.

5.4.1 Postfix Translation Schemes

5.4.1 Postfix Translation Schemes

By far the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed.

5.4.1 Postfix Translation Schemes

By far the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed.

In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production.

5.4.1 Postfix Translation Schemes

By far the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed.

In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production.

SDT's with all actions at the right ends of the production bodies are called *postfix SDT's*.

Example 5.14: The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser. □

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Figure 5.1: Syntax-directed definition of a simple desk calculator

Example 5.14: The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser. □

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Figure 5.1: Syntax-directed definition of a simple desk calculator

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

Figure 5.18: Postfix SDT implementing the desk calculator

5.4.2 Parser-Stack Implementation of Postfix SDT's

- In stack along with symbols (or states) values can be stored.
- Whenever a symbol is pushed on to stack its values also can be pushed.
- On reduction the popped values are used in finding the production's head value, which is pushed along with that symbol.
- Postfix action is carried out after this.

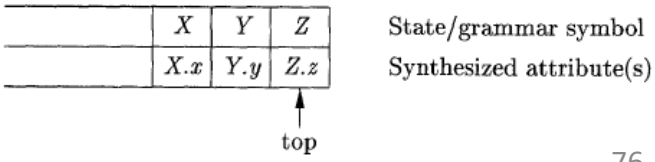


Figure 5.19: Parser stack with a field for synthesized attributes

5.4.3 SDT's With Actions Inside Productions

$B \rightarrow X \ \{a\} \ Y$

- If the parse is bottom-up, then we perform action a as soon as this occurrence of X appears on the top of the parsing stack.
- If the parse is top-down, we perform a just before we attempt to expand this occurrence of Y (if Y a nonterminal) or check for Y on the input (if Y is a terminal).

In general, for any SDT, first build the parse tree then do preorder traversal…

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node N, say one for production $A \to \alpha$. Add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.
3. Perform a preorder traversal (see Section 2.3.4) of the tree, and as soon as a node labeled by an action is visited, perform that action.

Informally, Top-down left to right traversal will do this.

An example

- Any SDT can be implemented by first building the parse tree and then traversing left-to-right depth-first order (i.e., preorder traversal).

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

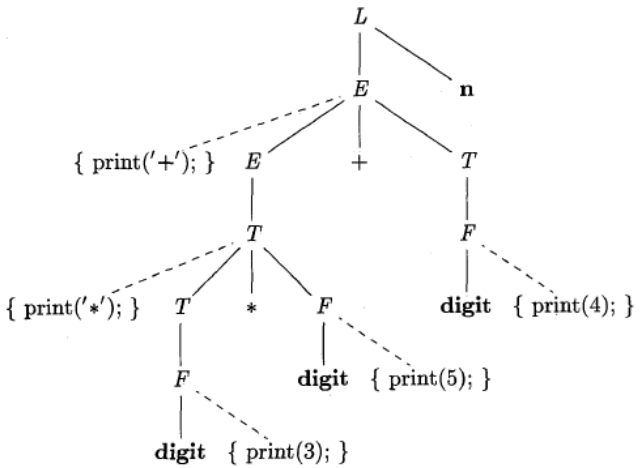


Figure 5.22: Parse tree with actions embedded

If we visit the nodes in preorder, we get the prefix form of the expression: $+ * 3\ 5\ 4$.

Input: 3*5+4

Output: $+ * 3\ 5\ 4$

Why we require actions within productions

- If we are employing top-down parsing like LL parsing then left recursion needs to be eliminated.
- Now the semantic action has to be associated with the new generated grammar.
- Here, actions, within production body is needed.

Recall how left recursion is eliminated …

Recall how left recursion is eliminated …

$A \rightarrow A\alpha \mid \beta$ is replaced by

$A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \epsilon$

Recall how left recursion is eliminated …

$A \rightarrow A\alpha \mid \beta$ is replaced by

$A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \epsilon$

Consider, $E \rightarrow E_1 + T \quad \{ \text{print}('+'); \}$
$E \rightarrow T$

Here, $\alpha = + \; T \; \{ \text{print}('+'); \}$

Recall how left recursion is eliminated …

$A \rightarrow A\alpha \mid \beta$ is replaced by

$A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \epsilon$

Consider, $E \rightarrow E_1 + T \quad \{ \text{print}('+'); \}$
$E \rightarrow T$

Here, $\alpha = + \; T \; \{ \text{print}('+'); \}$

So the transformed SDD will be

$E \rightarrow T \; R$
$R \rightarrow + \; T \; \{ \text{print}('+'); \} \; R$
$R \rightarrow \epsilon$

Recall how left recursion is eliminated …

$A \rightarrow A\alpha \mid \beta$ is replaced by

$A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \epsilon$

Consider, $E \rightarrow E_1 + T \quad \{ \text{print}('+'); \}$
$E \rightarrow T$

Here, $\alpha = + \; T \; \{ \text{print}('+'); \}$

So the transformed SDD will be

$E \rightarrow T \; R$
$R \rightarrow + \; T \; \{ \text{print}('+'); \} \; R$
$R \rightarrow \epsilon$

- When transforming the grammar, treat the actions as if they were terminal symbols.

- When transforming the grammar, treat the actions as if they were terminal symbols.

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string. The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up.

- When transforming the grammar, treat the actions as if they were terminal symbols.

Left recursion with semantic rules

$$
\begin{array}{lcl}
A & \rightarrow & A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\
A & \rightarrow & X\ \{A.a = f(X.x)\}
\end{array}
$$

==Can be converted to=>

$$
\begin{array}{lcl}
A & \rightarrow & X\ R \\
R & \rightarrow & Y\ R \mid \epsilon
\end{array}
$$

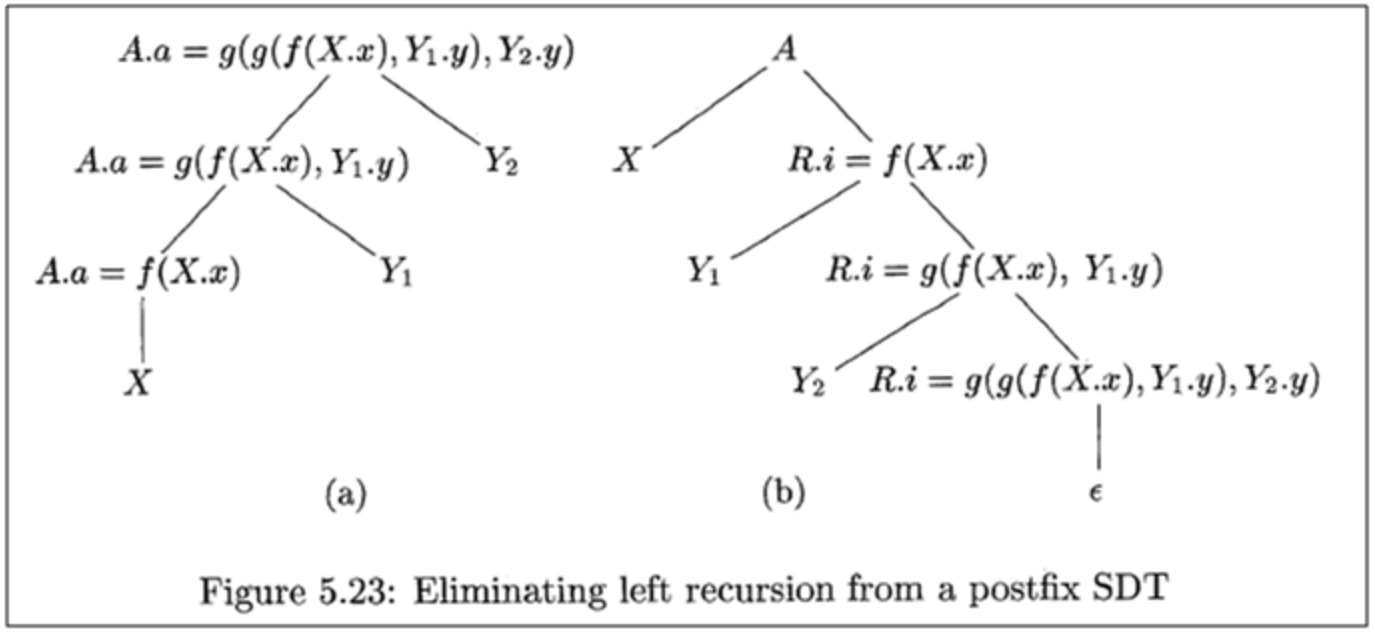
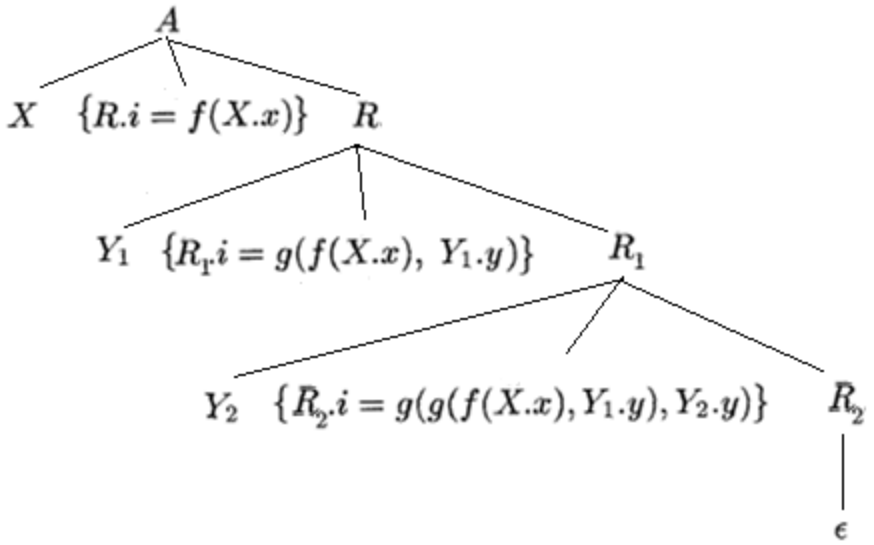


Figure 5.23: Eliminating left recursion from a postfix SDT

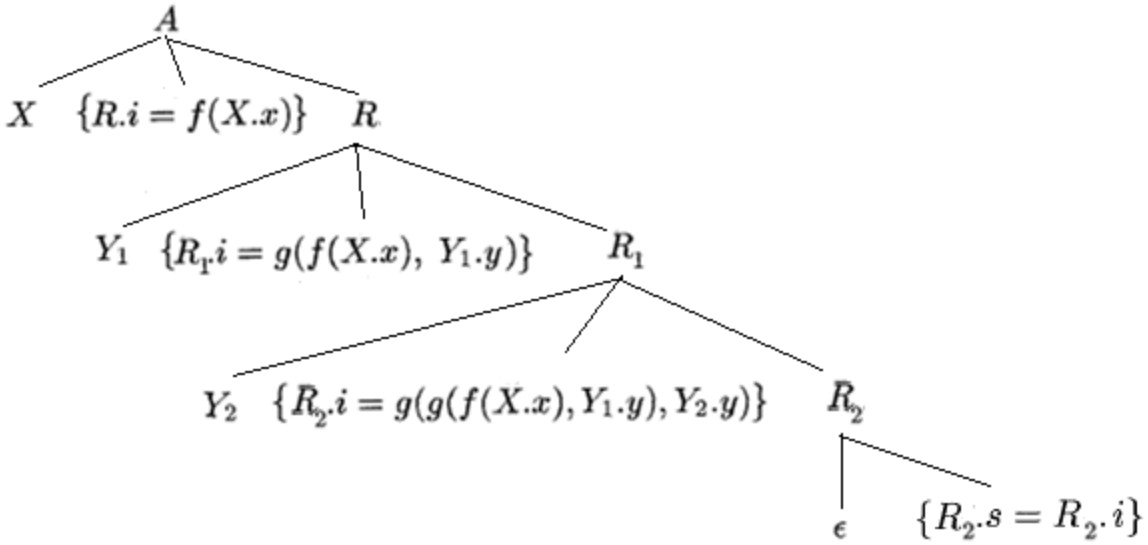
For new non-terminal ***R*** a new inherited attribute ***R.i*** is created and is calculated as shown. Note, finally we require ***A.a*** ??

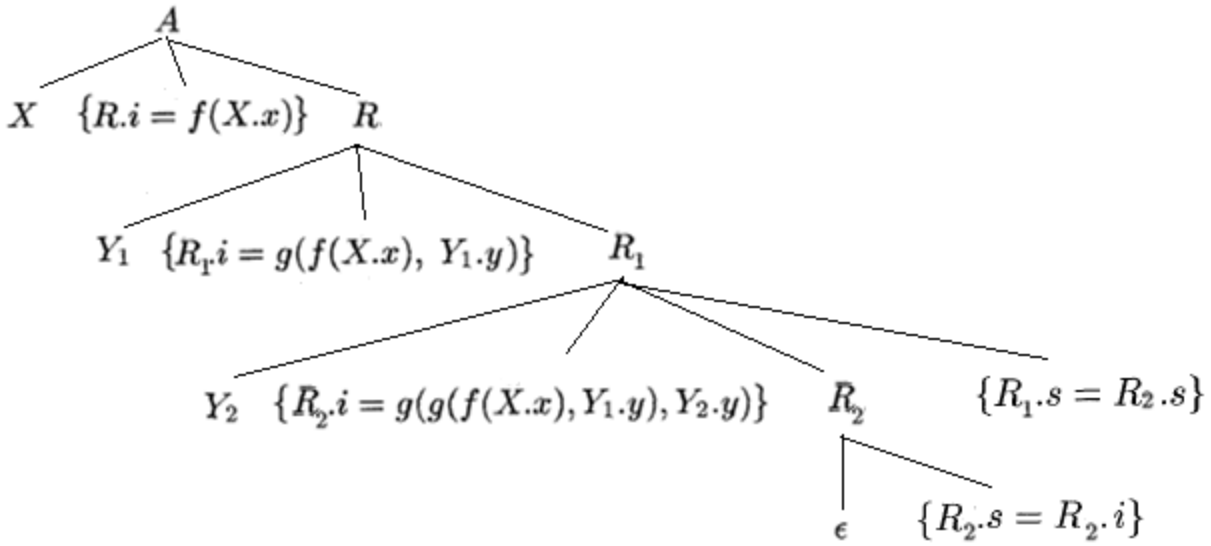
Actions needs to be done in the shown order.

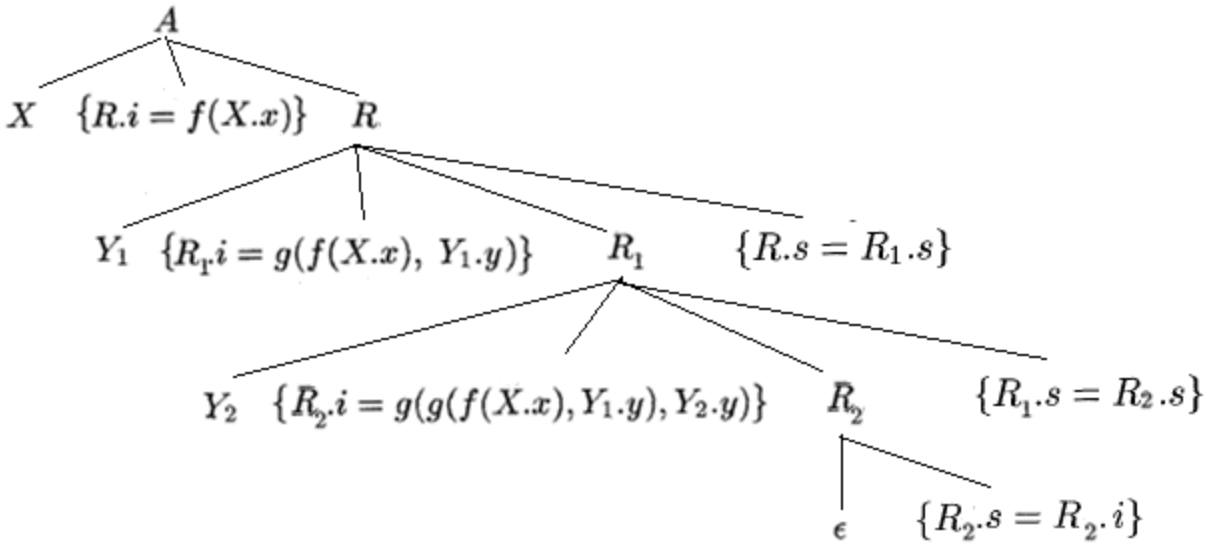


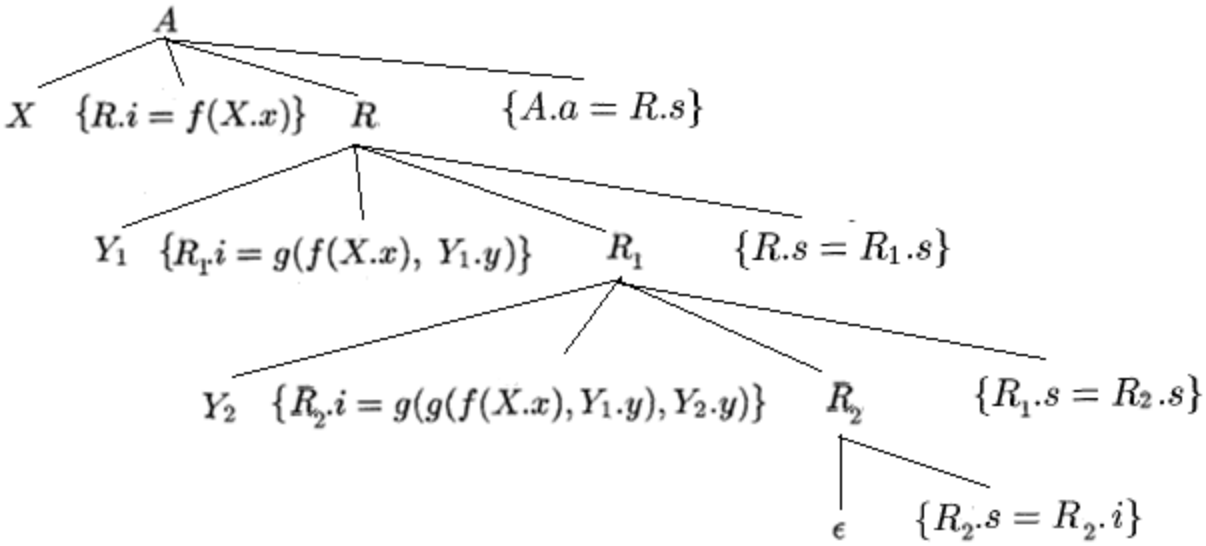
We want $A.a$ to be $R_2.i$

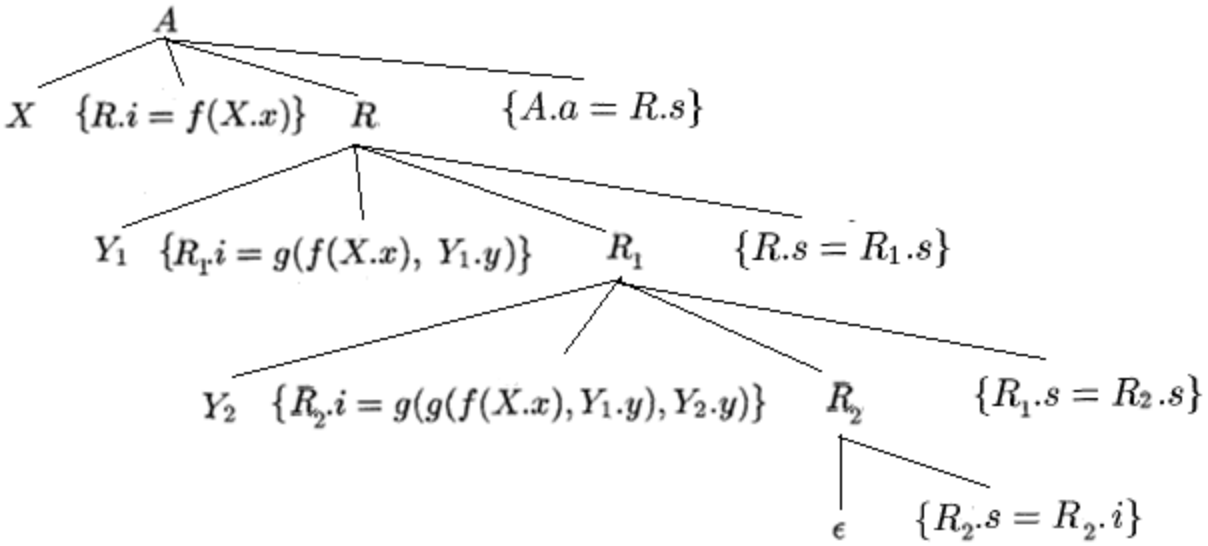
A sequence of actions should bring $R_2.i$ to the root


A
X
{R.i = f(X.x)}
R
Y1
{R1.i = g(f(X.x), Y1.y)}
R1
Y2
{R2.i = g(g(f(X.x), Y1.y), Y2.y)}
R2
ϵ
{R2.s = R2.i}


A
X
{R.i = f(X.x)}
R
Y1
{R1.i = g(f(X.x), Y1.y)}
R1
Y2
{R2.i = g(g(f(X.x), Y1.y), Y2.y)}
R2
{R1.s = R2.s}
ϵ
{R2.s = R2.i}


A
X
{R.i = f(X.x)}
R
Y1
{R1.i = g(f(X.x), Y1.y)}
R1
{R.s = R1.s}
Y2
{R2.i = g(g(f(X.x),Y1.y),Y2.y)}
R2
{R1.s = R2.s}
ϵ
{R2.s = R2.i}


A
X
{R.i = f(X.x)}
R
{A.a = R.s}
Y1
{R1.i = g(f(X.x), Y1.y)}
R1
{R.s = R1.s}
Y2
{R2.i = g(g(f(X.x), Y1.y), Y2.y)}
R2
{R1.s = R2.s}
ϵ
{R2.s = R2.i}



So the SDT has to be

$$
\begin{array}{lcl}
A & \rightarrow & X \ \{R.i = f(X.x)\} \ R \ \{A.a = R.s\} \\
R & \rightarrow & Y \ \{R_1.i = g(R.i, Y.y)\} \ R_1 \ \{R.s = R_1.s\} \\
R & \rightarrow & \epsilon \ \{R.s = R.i\}
\end{array}
$$

5.4.5 SDT's for L-Attributed Definitions

5.4.5 SDT's for L-Attributed Definitions

The rules for turning an L-attributed SDD into an SDT are as follows:

5.4.5 SDT's for L-Attributed Definitions

The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production. If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.

5.4.5 SDT's for L-Attributed Definitions

The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production. If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

Example 5.18: This example is motivated by languages for typesetting mathematical formulas. **Eqn** is an early example of such a language; ideas from **Eqn** are still found in the TEX typesetting system, which was used to produce this book.

Figure 5.24: Constructing larger boxes from smaller ones

$B \rightarrow B_1\ B_2 \mid B_1\ \textbf{sub}\ B_2 \mid (\ B_1\ ) \mid \textbf{text}$

	PRODUCTION	SEMANTIC RULES
1)	$S \rightarrow B$	$B.ps = 10$
2)	$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \rightarrow B_1\ \textbf{sub}\ B_2$	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4)	$B \rightarrow (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \rightarrow \textbf{text}$	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

Reading Assignment:

Read the Dragon book !

Example 5.19: For this example, we only need one production:

$$S \rightarrow \textbf{while} \ (\ C \ ) \ S_1$$

Example 5.19: For this example, we only need one production:

$$S \rightarrow \textbf{while} \; (\; C \;) \; S_1$$

- L_1, L_2 , C.true, C.false, S.next are labels
- S.code, C.code are attributes

Example 5.19: For this example, we only need one production:

$$S \rightarrow \textbf{while} \; (\; C \;) \; S_1$$

- L_1, L_2 , C.true, C.false, S.next are labels
- S.code, C.code are attributes

C.code:

if C

goto C.true

else

goto C.false

Example 5.19: For this example, we only need one production:

$$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$$

- L_1, L_2 , C.true, C.false, S.next are labels
- S.code, C.code are attributes

C.code:

```
if C
        goto C.true
else
        goto C.false
```

S.code:

```
Label L1
        C.code
Label L2
        S1.code
        goto S1.next
Label S.next
```

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$	$L1 = new();$ $L2 = new();$ $S_1.next = L1;$ $C.false = S.next;$ $C.true = L2;$ $S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code \parallel \text{goto}\ S_1.next \parallel \textbf{label} \parallel S.next;$

Figure 5.27: SDD for while-statements

$S \rightarrow \textbf{while}\ ($	$\{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \}$
$C\ )$	$\{\ S_1.next = L1;\ \}$
S_1	$\{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code \parallel \text{goto}\ S_1.next \parallel \textbf{label} \parallel S.next;\}$

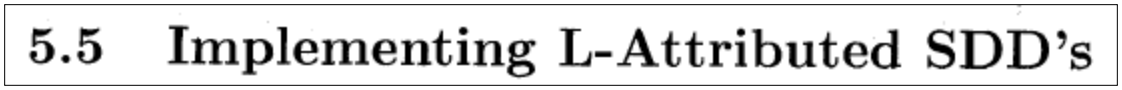
Figure 5.28: SDT for while-statements

;

- The following is not covered and is not part of your syllabus.

5.5 Implementing L-Attributed SDD's

- The following is not covered and is not part of your syllabus.



- Next: