5. Syntax-Directed Translation

Semantic Analysis

Outline

- Syntax Directed Translation
  - Syntax Directed Definitions
  - Translation Schemes - Evaluation Orders of SDD's
- Applications of Syntax Directed Translation

Introduction

- We can associate information with a language construct by attaching attributes to the grammar symbols.
- A syntax directed definition specifies the evaluation rules for values of attributes by associating semantic rules with the grammar productions.

Production	*Semantic* Rule
E->E1+T	E.val=E1.val +T.val

- We may alternatively insert the semantic actions inside the grammar

E -> E1+T {print '+'}

Syntax Directed Definitions

- A SDD is a context free grammar with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
  - Example of attributes: values, types, table references, etc.
- Classification of attributes: inherited and synthesized

Syntax Tree and Attributes

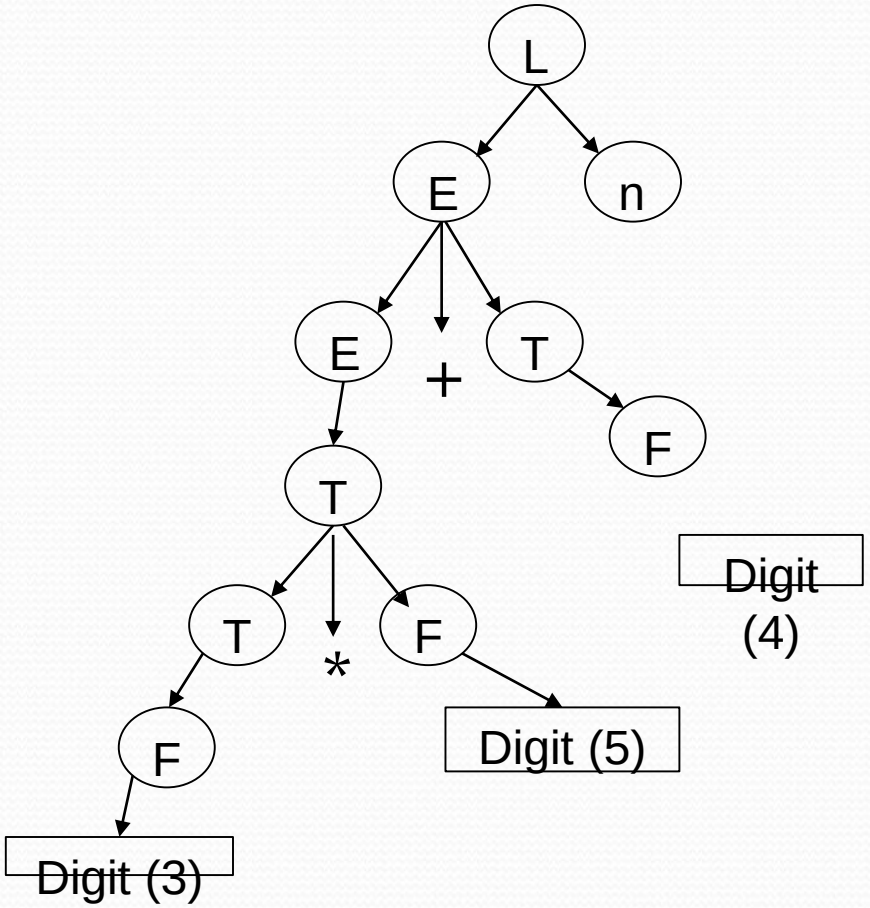
A parse tree, showing the value(s) of its attribute(s) at each node is called a syntax tree or annotated parse tree. One node may have multiple attributes.

Parse Tree vs. Syntax Tree

L -> E n
E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> digit

3 * 5 + 4 n

Parse Tree

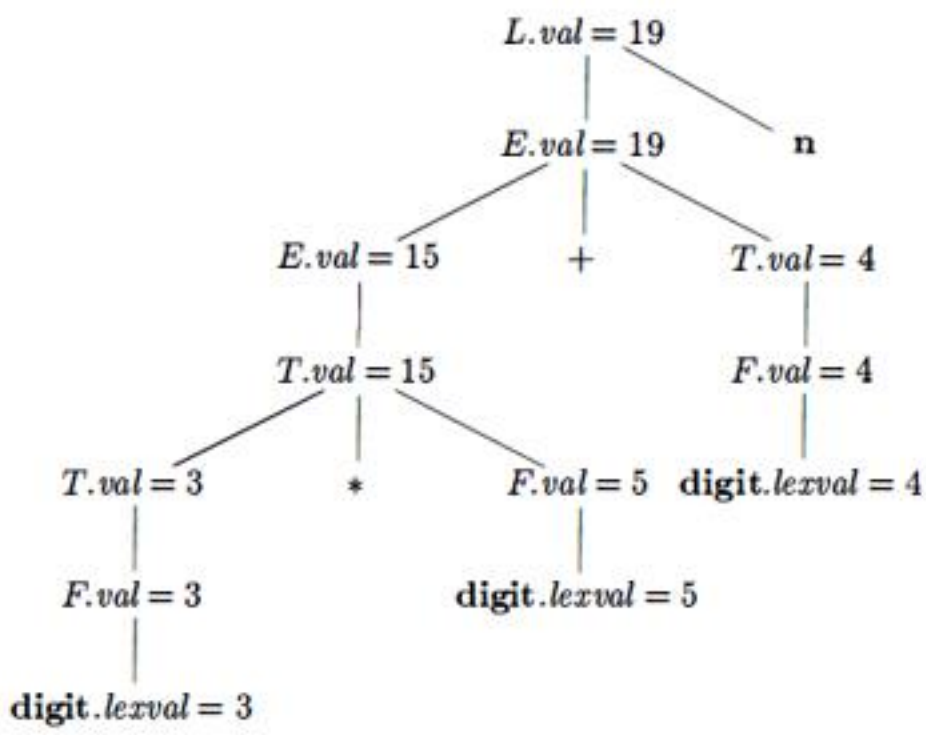


Parse Tree vs. Syntax Tree

Syntax Tree

L -> E n
E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> digit

3 * 5 + 4 n



Annotated parse tree for 3 * 5 + 4 n

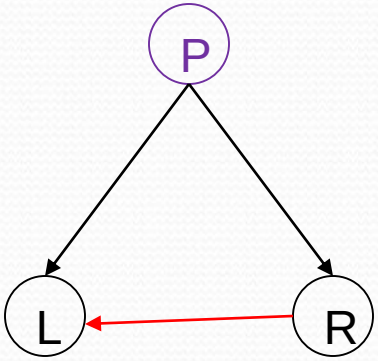
Attributes

- Synthesized attributes
  - A synthesized attribute at node N is defined only in terms of attribute values of children of N.
- 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

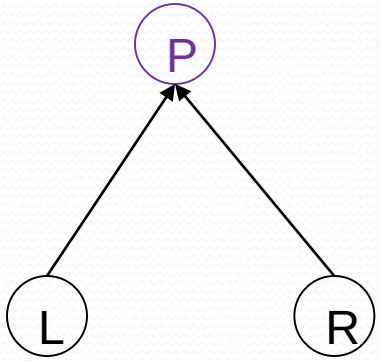
Evaluation of Attributes by Tree Traversal

Tree traversal methods:

- in-order (L – P – R)
- pre-order (P – L – R)
- post-order (L – R – P)



Single pre-order traversal cannot evaluate all inherited attributes



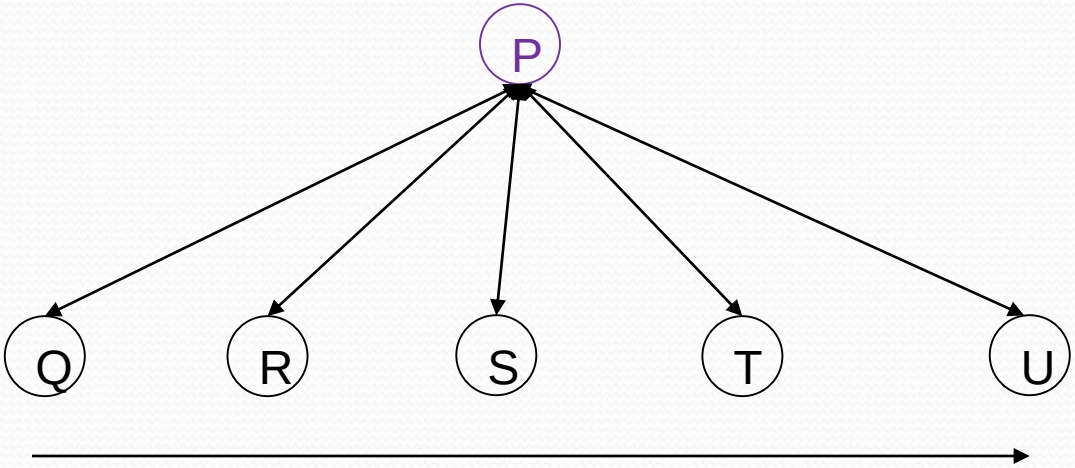
Single post-order traversal can evaluate all synthesized attributes

Evaluation of Attributes by Tree Traversal

L-Attributes (S-Attributes+subset of I–Attributes)

This set of attributes contains two types of attributes:

- Synthesized attributes
- Inherited attributes which depends on parent's and left siblings attributes



Example of S-attributed SDD

Production	Semantic Rules
1) L -> E n	L.val = E.val
2) E -> E1 + T	E.val = E1.val + T.val
3) E -> T	E.val = T.val
4) T -> T1 * F	T.val = T1.val * F.val
5) T -> F	T.val = F.val
6) F -> (E)	F.val = E.val
7) F -> digit	F.val = digit.lexval

Example of mixed attributes

Production	Semantic Rules
1) T -> FT’	T’.inh = F.val T.val = T’.syn
2) T’ -> *FT’$_1$	T’$_1$.inh = T’.inh*F.val T’.syn = T’$_1$.syn
3) T’ -> ε	T’.syn = T’.inh
4) F -> digit	F.val = digit.lexval

Evaluation orders for SDD's

- A dependency graph is used to determine the order of computation of attributes
- Dependency graph
  - For each parse tree node, the parse tree has a node for each attribute associated with that node
  - If a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b
  - If a semantic rule defines the value of inherited attribute B.c in terms of the value of X.a then the dependency graph has an edge from X.c to B.c

Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus the only allowable orders of evaluation are those sequence of nodes N1,N2,…,Nk such that if there is an edge from Ni to Nj then i<j
- Such an ordering is called a topological sort of a graph

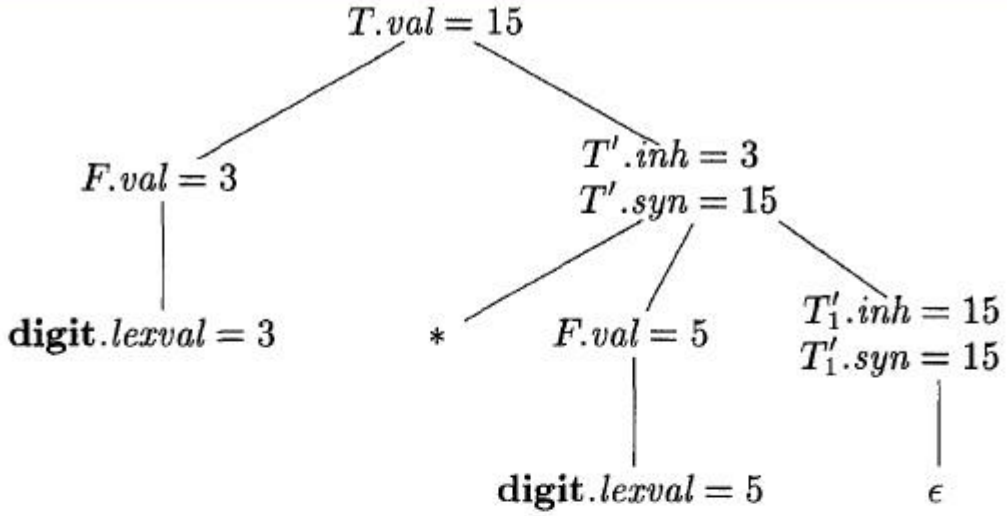
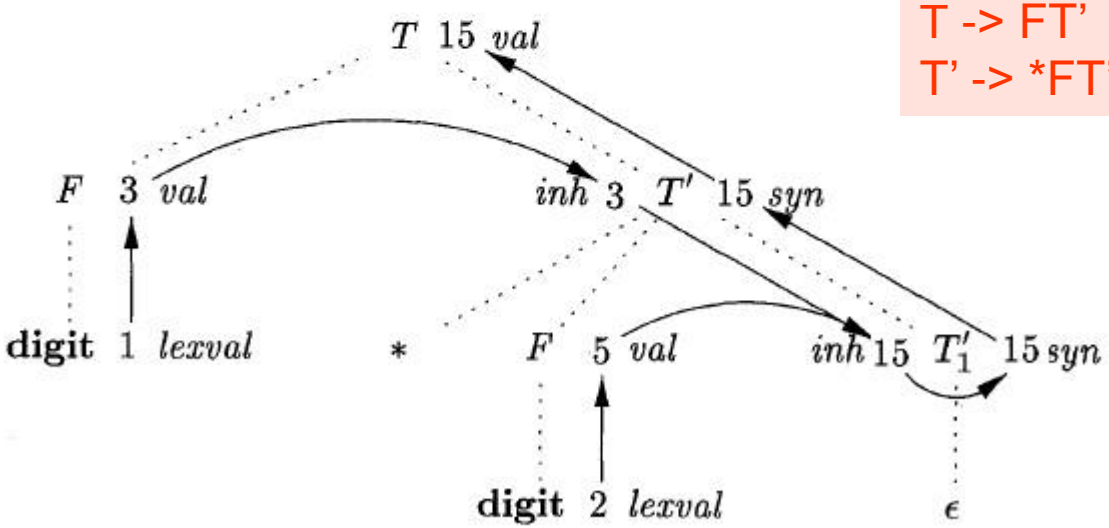


Figure 5.5: Annotated parse tree for $3 * 5$

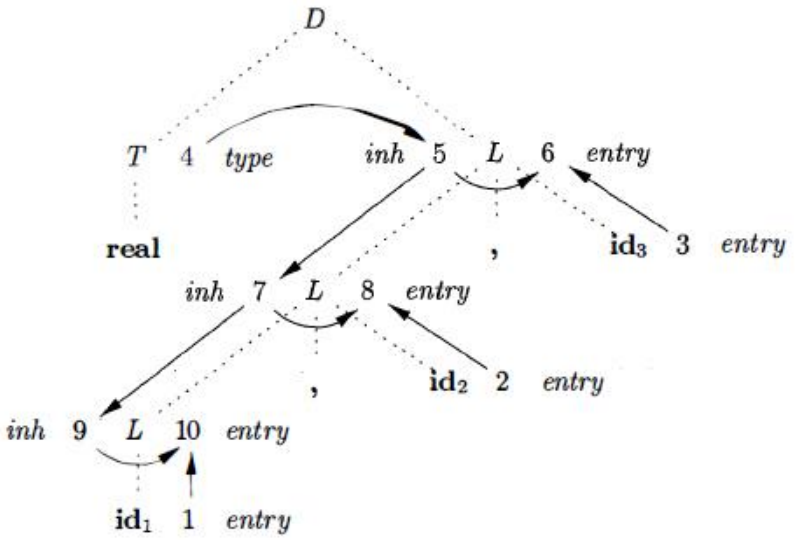
T -> FT'
T' -> *FT'1



Dependency graph for the annotated parse tree of Fig. 5.5

	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)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

Syntax-directed definition for simple type declarations



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

S-Attributed definitions

- An SDD is S-attributed if every attribute used in SDD is synthesized
- We can have a post-order traversal of parse-tree to evaluate attributes in S-attributed definitions

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

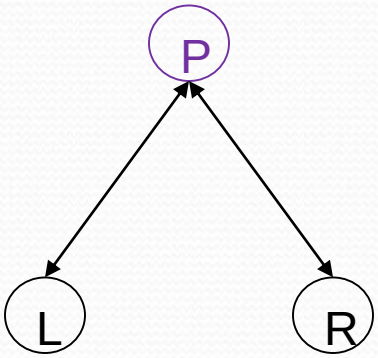
- S-Attributed definitions can be implemented during bottom-up parsing without the need to explicitly create parse trees

L-Attributed definitions

- A SDD is L-Attributed if the edges in dependency graph goes from Left to Right but not from Right to Left.
- More precisely, each attribute must be either
  - Synthesized
  - Inherited, but if there is a production A->X1X2…Xn and there is an inherited attribute Xi.a computed by a rule associated with this production, then the rule may only use:
    - Inherited attributes associated with the head A
    - Either inherited or synthesized attributes associated with the occurrences of symbols X1,X2,…,Xi-1 located to the left of Xi
    - Inherited or synthesized attributes associated with this occurrence of Xi itself, but in such a way that there is no cycle in the graph

L-Attributed definitions

- L-Attributed definitions can be evaluated by *depth first order* traversal of tree.
- Depth first order is a combination of *pre-order* and *post-order* traversal.
- Each sub-tree is evaluated in P-L-R-P order. Hence each internal node visited twice. On first visit inherited attributes and on second visit synthesized attributes will be evaluated.



Applications of Syntax Directed Translation

- Construction of syntax trees
  - Leaf nodes: Leaf(operand, val)
  - Interior node: Node(operator,c1,c2,…,ck)
- Evaluation of expression
- Type checking and intermediate code Generation

Construction of Syntax Trees

For S-Attributed Definitions with Bottom-up parsing

	Production	*Semantic Rules*
1)	E -> E1 + T	E.node=new_node(‘+’, E1.node,T.node)
2)	E -> E1 - T	E.node=new_node(‘-’, E1.node,T.node)
3)	E -> T	E.node = T.node
4)	T -> (E)	T.node = E.node
5)	T -> id	T.node = new_Leaf(id,id.entry)
6)	T -> num	T.node = new_Leaf(num,num.val)

Construction of Syntax Trees

For L-Attributed Definitions with Top-down parsing

	Production	*Semantic Rules*
1)	E -> TE’	E.node=E’.syn E’.inh=T.node
2)	E’ -> + TE_1’	E_1’.inh=new_node(‘+’, E’.inh,T.node) E’.syn=E_1’.syn
3)	E’ -> -TE_1’	E_1’.inh=new_node(‘+’, E’.inh,T.node) E’.syn=E_1’.syn
4)	E’ -> $\in$	E’.syn = E’.inh
5)	T -> (E)	T.node = E.node
6)	T -> id	T.node=new_Leaf(id,id.entry)
7)	T -> num	T.Node= new_Leaf(num,num.val)

Evaluation of Attributes

Top-down evaluation

Production	*Semantic Rules*
1) T -> FT’	T’.inh = F.value T.value = T’.syn
2) T’ -> *FT’$_1$	T’$_1$.inh = T’.inh*F.value T’.syn = T’$_1$.syn
3) T’ -> ε	T’.syn = T’.inh
4) F -> digit	F.value = digit.lexvalue
5) F-> **id**	F.value = symbol_table[id.entry].value

Evaluation of Attributes

Bottom-up evaluation of synthesized attributes

Production	*Semantic Rules*
L -> E n	{print(stack[top-1].value); top=top-1;}
E -> E1 + T	{stack[top-2].value=stack[top-2].value+stack[top].value; top=top-2;}
E -> T	
T -> T1 * F	{stack[top-2].val=stack[top-2].val+stack[top].val; top=top-2;}
T -> F	
F -> (E)	{stack[top-2].val=stack[top-1].val; top=top-2;}
F -> digit	{top=top+1; stack[top].value = digit.lex_value}
F -> **id**	{top=top+1; stack[top].value = symbol_table[id.entry].value}

Evaluation of Attributes

Bottom-up evaluation of inherited attributes

	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)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

Syntax-directed definition for simple type declarations

int id1, id2, id3

Type checking and intermediate code Generation

Will be discussed in another chapter.

Syntax directed translation schemes

- An SDT is a Context Free grammar with program fragments embedded within production bodies
- Those program fragments are called semantic actions
- They can appear at any position within production body
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order (depth first visit)
- Typically SDT's are implemented during parsing without building a parse tree

SDT's with actions inside productions

- For a production B->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 parser stack
  - If the parser is top down we perform "a" just before we expand Y

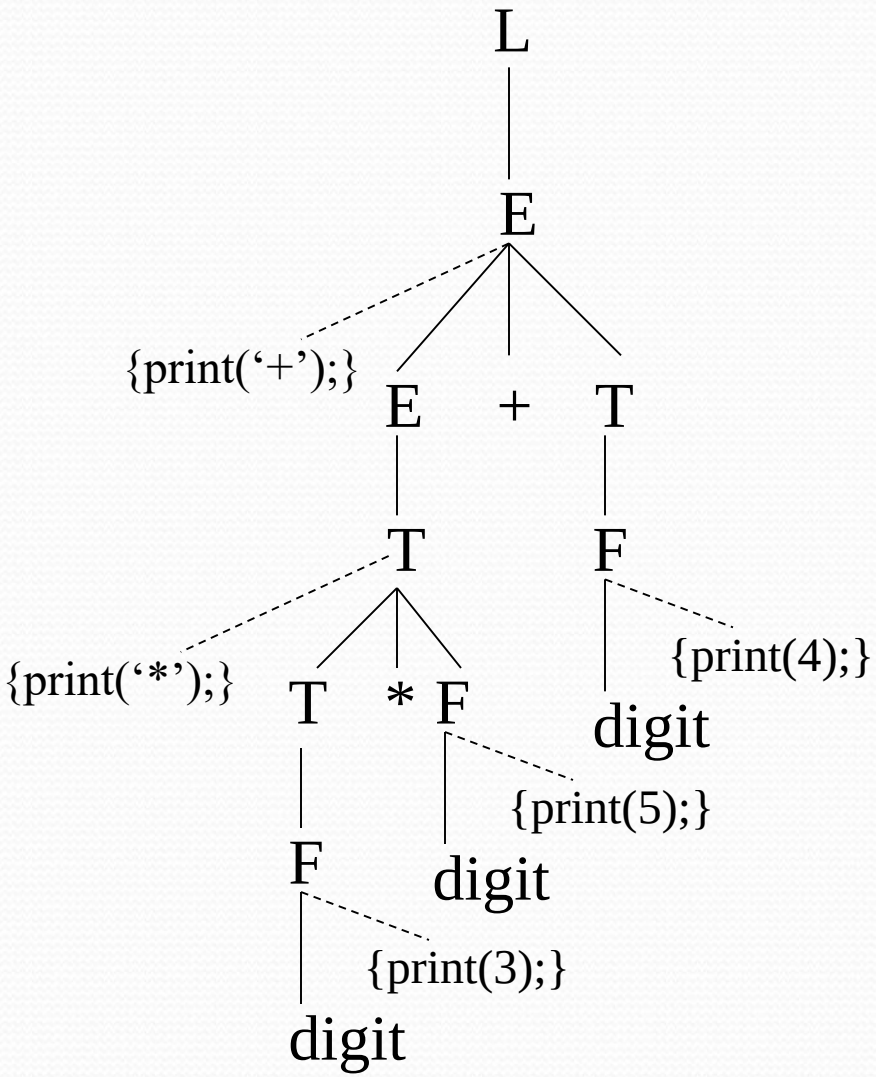
•Sometimes we cant do things as easily as explained above

•One example is when we are parsing this SDT with a bottom-up parser

1) L -> E n
2) E -> {print('+');} E1 + T
3) E -> T
4) T -> {print('*');} T1 * F
5) T -> F
6) F -> (E)
7) F -> digit {print(digit.lexval);}

SDT's with actions inside productions (cont)

- Any SDT can be implemented as follows
  1. Ignore the actions and produce a parse tree
  2. Examine each interior node N and add actions as new children at the correct position
  3. Perform a post-order traversal and execute actions when their nodes are visited



SDT's for L-Attributed definitions

- We can convert an L-attributed SDD into an SDT using following two rules:
  - Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A. if several inherited attributes of A are dpendent on one another in an acyclic fashion, order them so that those needed first are computed first
  - Place the action of a synthesized attribute for the head of a production at the end of the body of the production

End