Beyond the capabilities of standard Parsing Techniques

- Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- Can do the followings…
  - generate intermediate codes
  - put information into the symbol table
  - perform type checking
  - issue error messages, etc.
- Two such tools
  - Syntax directed definitions
  - Syntax directed translations

Syntax Directed Definition (SDD)

- A context-free grammar together with *attributes* and *semantic rules*.
  - Attribute may represent: Number, type, string, memory size, label, etc.
  - Values of the attributes are computed by semantic rules associated with productions.

- Example 2: Syntax-directed definition of a simple desk-calculator that evaluates expressions terminated by an end-marker **n**.

	Production	Semantic Rules
1)	$L \to E\ \mathbf{n}$	$L.val = E.val$
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)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Syntax Directed Definition (SDD)

- Example 2: Syntax-directed definition for the grammar for type declaration.

PRODUCTION	SEMANTIC RULE
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow \mathsf{int}$	$T.type := integer$
$T \rightarrow \mathsf{real}$	$T.type := real$
$L \rightarrow L_1, \mathsf{id}$	$L_1.in := L.in;$ $addtype(\mathsf{id}.entry, L.in)$
$L \rightarrow \mathsf{id}$	$addtype(\mathsf{id}.entry, L.in)$

Syntax Directed Definition (SDD)

- Does not specify explicitly the order in which the attributes can be evaluated.

- The semantic rules implicitly indicate the order (*b* depends on $c_1, c_2, \ldots, c_k$).

- More useful for specification, Hide implementation details.

- Also called Attribute Grammars

Two types of attributes

- **Synthesized Attributes.**
  - They are computed from the values of the attributes of the children nodes and the node itself.

- **Inherited Attributes.**
  - They are computed from the values of the attributes of the siblings, the parent node and the node itself.

Two types of attributes

- Each production $A \rightarrow \alpha$ is associated with a set of semantic rules $b := f(c_1, c_2, \ldots, c_k)$ where *f* is a function, and either
  - b is synthesized attribute of A if $c_1, c_2, \ldots c_k$ are attributes of the grammar symbols in α, or
  - b is inherited attribute of a grammar symbol in α , and $c_1, c_2, \ldots c_k$ are attributes of the grammar symbols in α or attribute of A.

Two types of attributes

- Synthesized Attribute

PRODUCTION	SEMANTIC RULE
$L \rightarrow E\mathsf{n}$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$

- Inherited Attribute

PRODUCTION	SEMANTIC RULE
$D \rightarrow T L$	$L.in := T.type$

Observations

- We do not allow inherited attribute at node N to be defined in terms of attribute values at the children of N.

- We do allow a synthesized attribute at node N to be defined in terms of inherited attributes at node N itself.

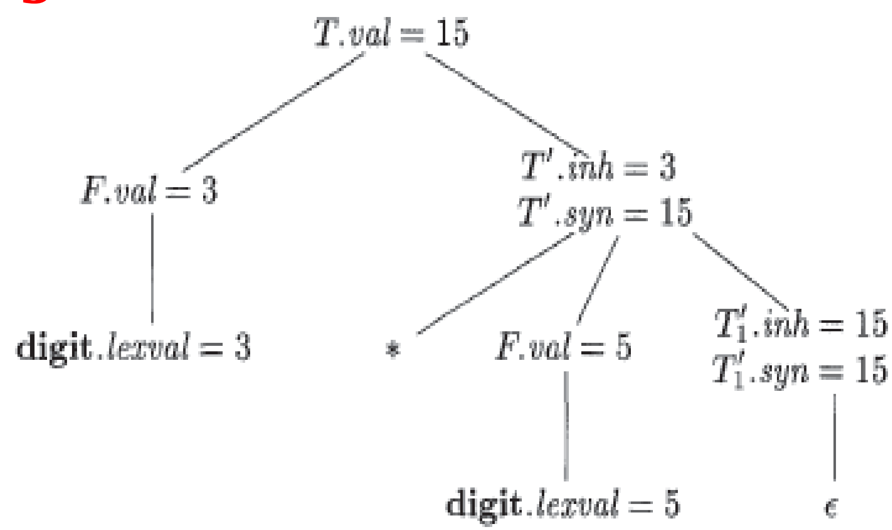
- Terminal symbols are assumed to have synthesized attributes supplied by the lexical analyzer.

Annotated Parse-Trees

- Parse-tree that also shows the values of the attributes at each node.
- Values of Attributes in nodes of annotated parse-tree are either,
  - Provided by the lexical analyzer.
  - Determined by the semantic-rules.

The annotated parse-tree for 3*5

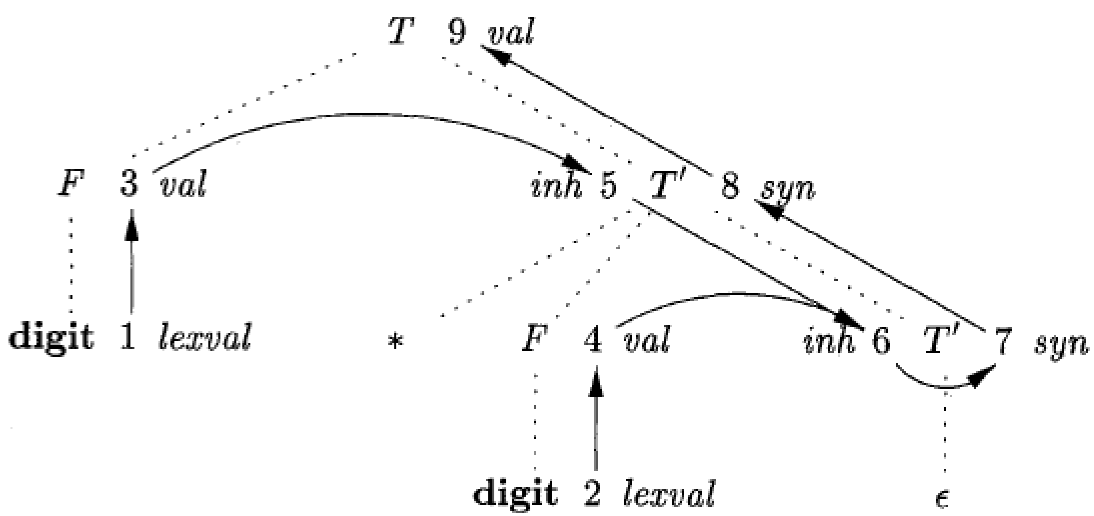
	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

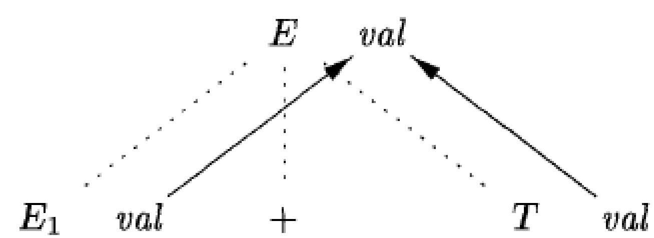


Evaluation orders for SDDs: Dependency Graphs

- Dependency graph is a useful tool for determining an evaluation order for the attribute instances in a given parse-tree.

- A dependency graph depicts the flow of information among the attribute instances in a particular parse tree.





Evaluation orders for SDDs: Dependency Graphs

Algorithm to build dependence graphs

FOR each node n in the parse tree DO

FOR each attribute a (of n) construct a node in the dependency graph

END FOR

END FOR

FOR each node n in the parse tree DO

FOR each semantic rule $b := f(c_1, c_2, c_3, \ldots, c_k)$ DO

Construct an edge from c_i to b

END FOR

END FOR

Evaluation orders for SDDs: Dependency Graphs

- The topological sort of the dependency graph shows the evaluation order

In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

Any ordering $m_1, m_2, \ldots, m_k$ such that if $m_i \rightarrow m_j$ is a link in the dependency graph then $m_i < m_j$.

SDD with controlled side-effects

- Desk calculator may print a result

	Production	Semantic Rule
1)	$L \rightarrow E\ \mathbf{n}$	$print(E.val)$

- Code generator might enter the type of identifier into a symbol table

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

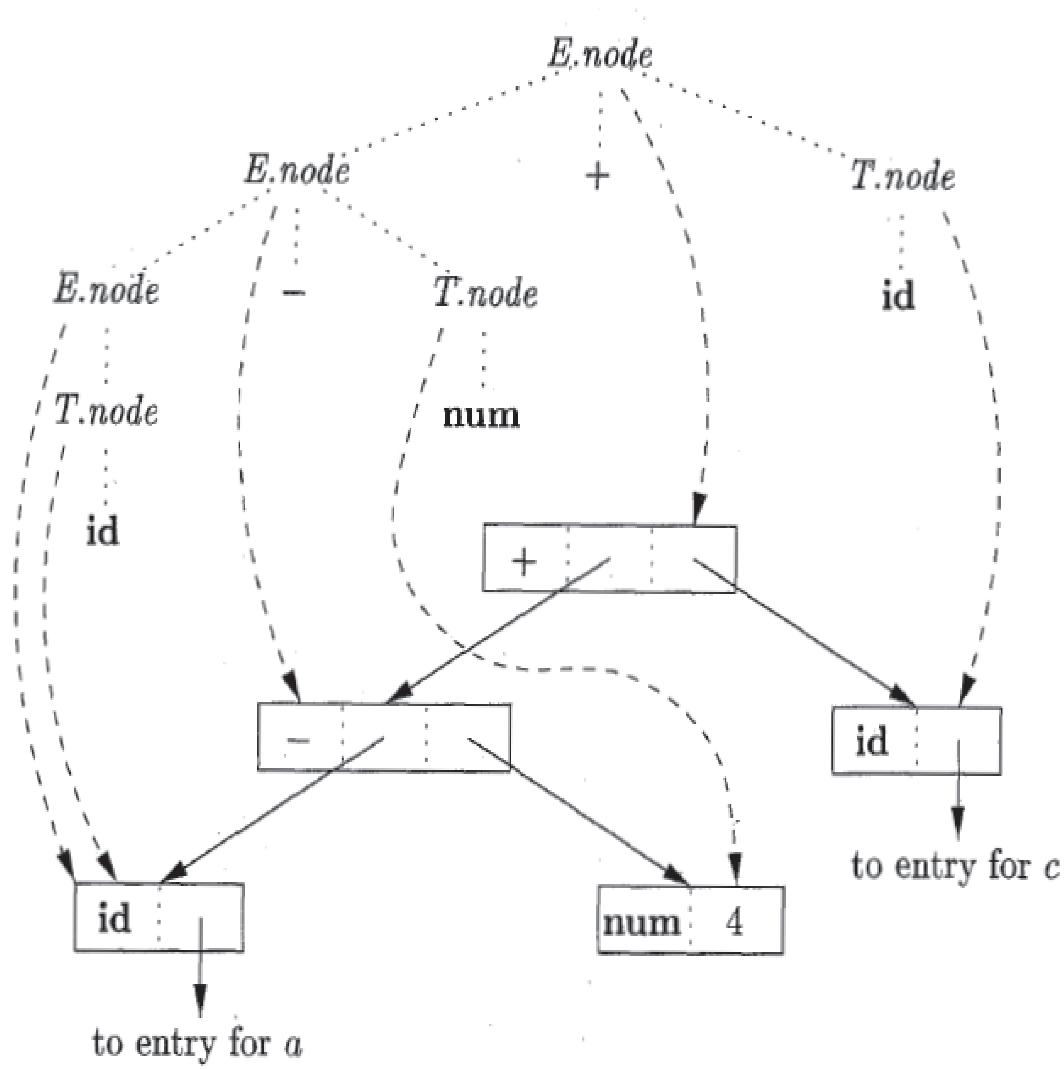
SDD to construct Syntax-tree

- The syntax tree is an abstract representation of the program constructs
- Used as intermediate representation for some compilers

	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)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

Syntax tree from 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) $p_5 = \textbf{new } Node('+', p_3, p_4);$



- Build dependency graph for *a-4+c* and evaluate the attributes in a topological order to build syntax-tree.

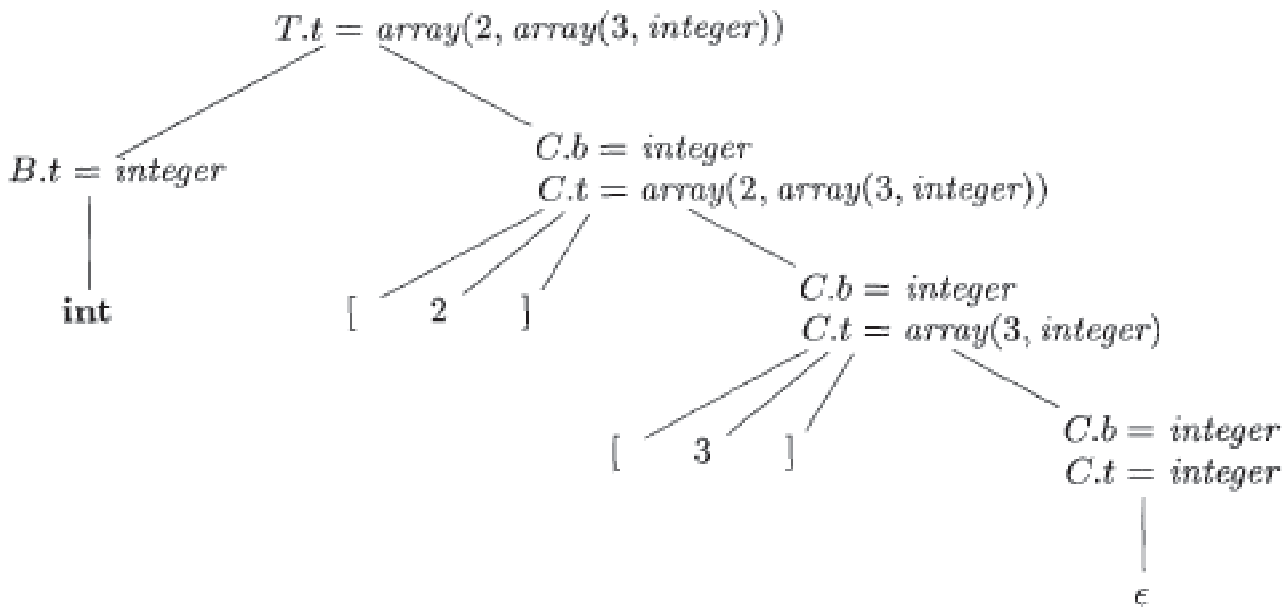
	Production	Semantic Rules
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E'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

The structure of a type

- SDD to generate either basic type or an array type: *inherited attribute b and synthesized attribute t*

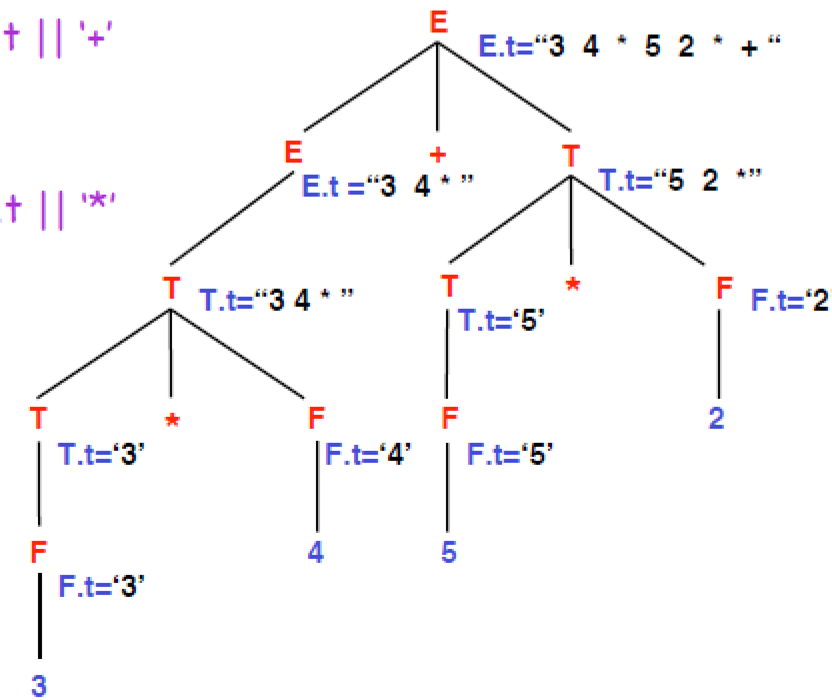
PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

```
int[2][3] ≡ array(2,array(3,integer))
```



SDD to generate postfix expression

$E \rightarrow E_1 + T$	$E.t := E_1.t \mid\mid T.t \mid\mid$ '+'
$\mid T$	$E.t := T.t$
$T \rightarrow T_1 * F$	$T.t := T_1.t \mid\mid F.t \mid\mid$ '*'
$\mid F$	$T.t := F.t$
$F \rightarrow (E)$	$F.t := E.t$
$\mid$ num	$F.t :=$ num.t



Input string: 3 * 4 + 5 * 2

Problems with dependence graphs

- This method is time consuming due to the construction of the dependency graph.

- This method fails if the dependency graph has a cycle: We need a test for non-circularity (checking for Directed Acyclic Graph);

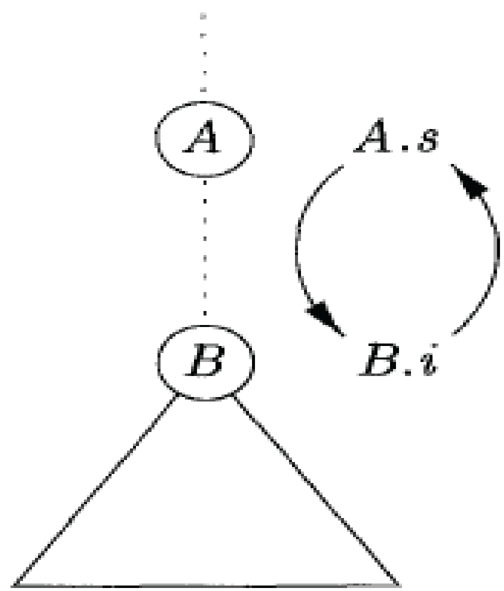
If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree.

If there are no cycles, however, then there is always at least one topological sort.

Circular Dependency

For SDD's with both inherited and synthesized attributes, there is no guarantee that there exists one order in which to evaluate attributes at nodes

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



Solutions

- Translation can be implemented using classes of SDD's that guarantee an evaluation order.

- Design the syntax directed definition in such a way that attributes can be evaluated with a *fixed order avoiding to build the* dependency graph (method followed by many compilers).

Two Classes of SDDs

- **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.

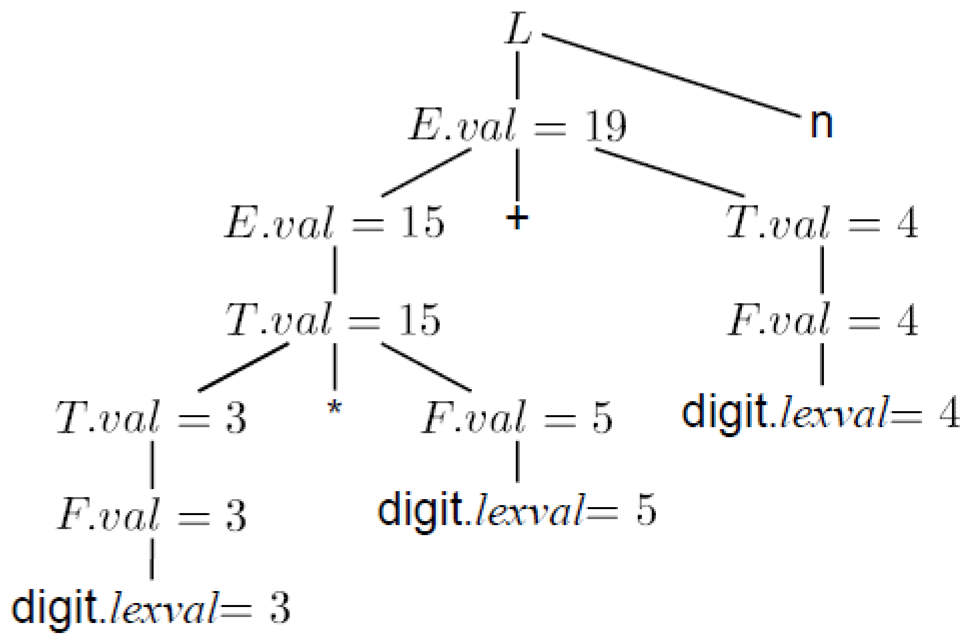
- **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.

S-attributed definition

- An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

	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

Annotated parse-tree for 3*5+4n



Evaluation of S-attributed SDDs

- Can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

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

L-attributed definition

Each attribute must be either

1. **Synthesized**

   or

2. **Inherited:**

   if $A \rightarrow X_1X_2 \ldots X_n$, **and** 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) **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**.

- Inherited Attributes are useful for expressing the dependence of a construct on the context in which it appears.
- Use both synthesized and inherited attributes.
- Inherited attributes that *do not depend from right children can be evaluated* by a classical PreOrder traversal of the parse-tree.

Is this SDD L-attributed?

	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)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**.$lexval$

Is this SDD L-attributed?

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

Evaluating L attributed definition

- The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

Algorithm: L-Eval(n: Node)

Input: Node of an annotated parse-tree.

Output: Attribute evaluation.

Begin

    For each child m of n, from left-to-right Do

    Begin

        Evaluate inherited attributes of m;

        L-Eval(m)

    End;

    Evaluate synthesized attributes of n

End.

Syntax directed translation Scheme

Syntax-Directed Translation scheme
(SDT)
=
CFG + program fragments embedded within production bodies

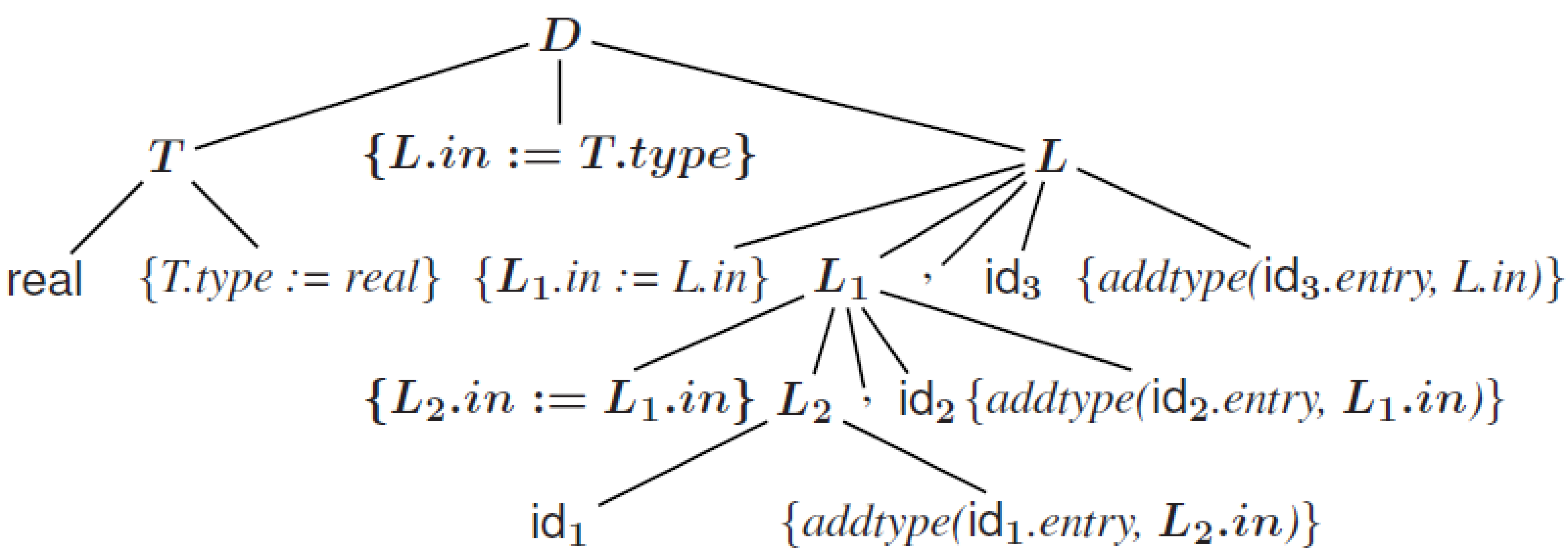
- **Definition:** A Translation Scheme is a context-free grammar in which
  - Attributes are associated with grammar symbols;
  - Semantic Actions are enclosed between braces {} and are inserted within the right-hand side of productions.
  - Semantic Actions are treated as terminal symbols

$$A \rightarrow \{ \ldots \}\ X\ \{ \ldots \}\ Y\ \{ \ldots \}$$

Semantic Actions

Example SDT: Real id1, id2, id3

$$
\begin{aligned}
D &\rightarrow\ T\ \{L.in := T.type\}\ L\\
T &\rightarrow\ \text{int}\ \ \{T.type := integer\}\\
T &\rightarrow\ \text{real}\ \ \{T.type := real\}\\
L &\rightarrow\ \{L_1.in := L.in\}\ L_1,\ \text{id}\ \ \{addtype(\text{id}.entry, L.in)\}\\
L &\rightarrow\ \text{id}\ \ \{addtype(\text{id}.entry, L.in)\}
\end{aligned}
$$



Syntax directed translation Scheme

- More implementation oriented than syntax directed definitions, as they indicate order of evaluation of semantic rules and attributes.

- Yacc uses Translation Schemes.

Implementation of SDT

- To avoid building parse-tree, SDTs can be implemented during parsing for two classes of SDDs
  - The underlying grammar is LR-parsable, and the SDD is S-attributed.
  - The underlying grammar is LL-parsable, and the SDD is L-attributed.

SDT for S-attributed definitions

- 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

- Postfix SDT: all actions at the right ends of the production bodies

$$
\begin{array}{lcll}
L & \to & E\ \mathbf{n} & \{\ \text{print}(E.val);\ \} \\
E & \to & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \to & T & \{\ E.val = T.val;\ \} \\
T & \to & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \to & F & \{\ T.val = F.val;\ \} \\
F & \to & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \to & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

Postfix SDT implementing the desk calculator

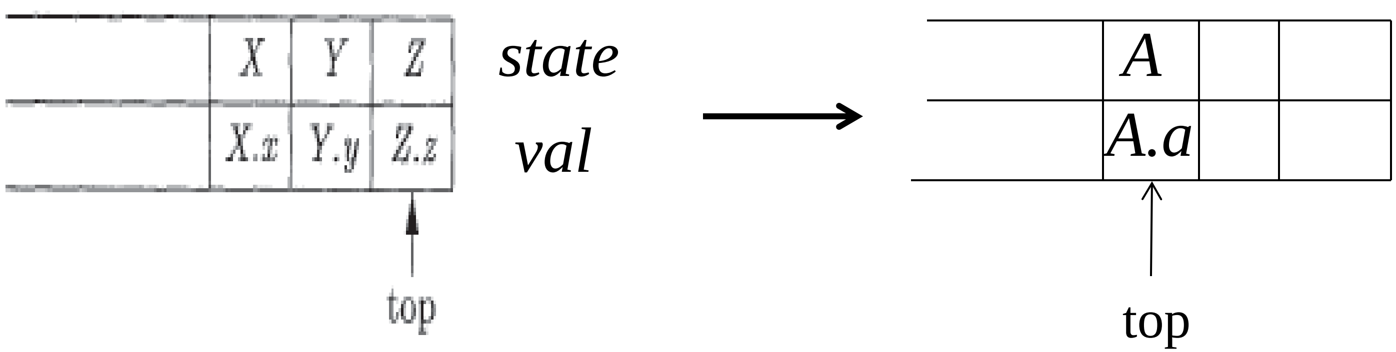
Parser-Stack Implementation of Postfix SDT's

- Synthesized Attributes can be evaluated by a bottom-up parser, i.e. simply by extending the stack of an LR-Parser.
  - The parser keeps the values of the synthesized attributes in its stack.
  - Whenever a reduction $A \rightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.

Parser-Stack Implementation of Postfix SDT's

- Consider a stack: elements with two fields *state* and *val*
- The current top of the stack is indicated by the pointer *top*.
- We evaluate the values of the attributes during reductions

$A \rightarrow XYZ \qquad A.a = f\,(X.x,\ Y.y,\ Z.z)$ where all attributes are synthesized



Parser-Stack Implementation of Postfix SDT's

PRODUCTION	ACTIONS
$L \to E\ \mathbf{n}$	$\{\ \text{print}(stack[top-1].val);$ $top = top - 1;\ \}$
$E \to E_1 + T$	$\{\ stack[top-2].val = stack[top-2].val + stack[top].val;$ $top = top - 2;\ \}$
$E \to T$	
$T \to T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;$ $top = top - 2;\ \}$
$T \to F$	
$F \to (\ E\ )$	$\{\ stack[top-2].val = stack[top-1].val;$ $top = top - 2;\ \}$
$F \to \mathbf{digit}$	

Implementing the desk calculator on a bottom-up parsing stack

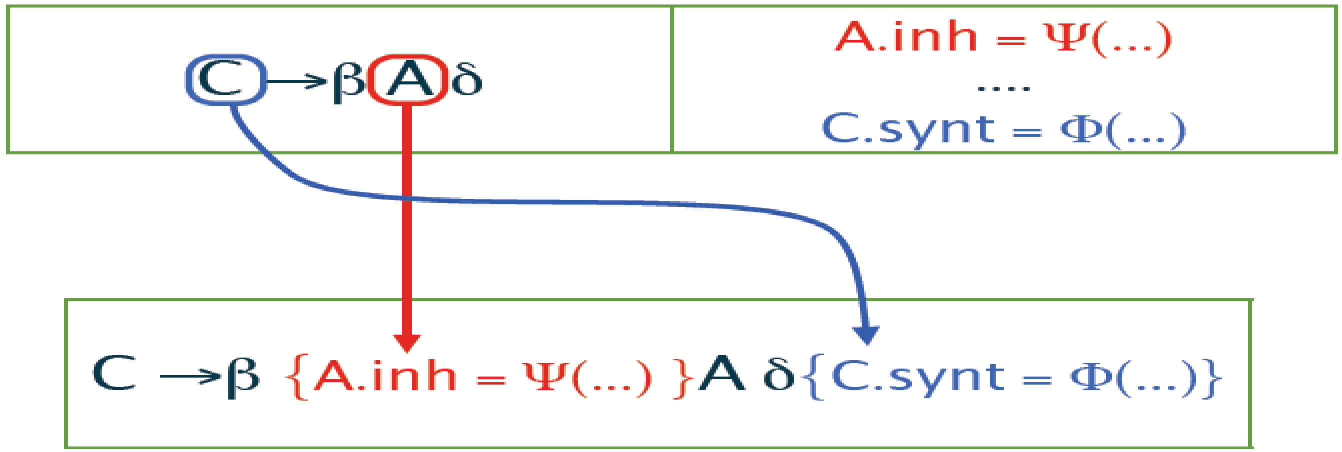
L	$\to$	$E\ \mathbf{n}$	$\{\ \text{print}(E.val);\ \}$
E	$\to$	$E_1 + T$	$\{\ E.val = E_1.val + T.val;\ \}$
E	$\to$	T	$\{\ E.val = T.val;\ \}$
T	$\to$	$T_1 * F$	$\{\ T.val = T_1.val \times F.val;\ \}$
T	$\to$	F	$\{\ T.val = F.val;\ \}$
F	$\to$	$(\ E\ )$	$\{\ F.val = E.val;\ \}$
F	$\to$	$\mathbf{digit}$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

Parser-Stack Implementation of Postfix SDT's

- Stack states are replaced by their corresponding grammar symbol;

INPUT	*state*	*val*	PRODUCTION USED
3*5+4 **n**	–	–	
*5+4 **n**	3	3	
*5+4 **n**	F	3	$F \rightarrow$ **digit**
*5+4 **n**	T	3	$T \rightarrow F$
5+4 **n**	T *	3 _	
+4 **n**	T * 5	3 _ 5	
+4 **n**	T * F	3 _ 5	$F \rightarrow$ **digit**
+4 **n**	T	15	$T \rightarrow T * F$
+4 **n**	E	15	$E \rightarrow T$
4 **n**	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	$F \rightarrow$ **digit**
n	E + T	15 _ 4	$T \rightarrow F$
n	E	19	$E \rightarrow E + T$
	E **n**	19 _	
	L	19	$L \rightarrow E$ **n**

SDT for L-attributed definitions



- The rules for turning an L-attributed SDD into an SDT are as follows:
  - 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.
  - Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

- Turn this L-attributed SDD into SDT

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

Example SDT: Type declaration

$$D \rightarrow \quad T \ \{L.in := T.type\} \ L$$

$$T \rightarrow \quad \textsf{int} \ \ \{T.type := integer\}$$

$$T \rightarrow \quad \textsf{real} \ \ \{T.type := real\}$$

$$L \rightarrow \quad \{L_1.in := L.in\} \ L_1, \textsf{id} \ \ \{addtype(\textsf{id}.entry, L.in)\}$$

$$L \rightarrow \quad \textsf{id} \ \ \{addtype(\textsf{id}.entry, L.in)\}$$

Implementing L-attributed SDD *in conjunction with an LL-parser*.

- We cannot use a recursive descent parser (or any other top-down parser) for a grammar that contains left-recursive productions

- So we need a method to transform grammars containing left-recursion into grammars without left-recursion

- The inherited attributes of a non-terminal A are placed in the stack along A. The code to evaluate these attributes will be represented by "action-record" placed immediately above A.

- The synthesized attributes for a non-terminal A are placed in a separate "synthesized-record" that is placed immediately below A.

Example: SDT for "while statement"

- SDD for while statement

$$
\begin{array}{ll}
S \rightarrow \textbf{while} \ (\ C\ )\ S_1 & L1 = new(); \\
& L2 = new(); \\
& S_1.next = L1; \\
& C.false = S.next; \\
& C.true = L2; \\
& S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code
\end{array}
$$

- SDT for while statement

$$
\begin{array}{lll}
S \rightarrow & \textbf{while} \ (& \{ L1 = new(); L2 = new(); C.false = S.next; C.true = L2; \} \\
& C\ ) & \{ S_1.next = L1; \} \\
& S_1 & \{ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code; \}
\end{array}
$$

Example: Various attributes

We use the following attributes to generate the proper intermediate code:

1. The inherited attribute $S.next$ labels the beginning of the code that must be executed after S is finished.

2. The synthesized attribute $S.code$ is the sequence of intermediate-code steps that implements a statement S and ends with a jump to $S.next$.

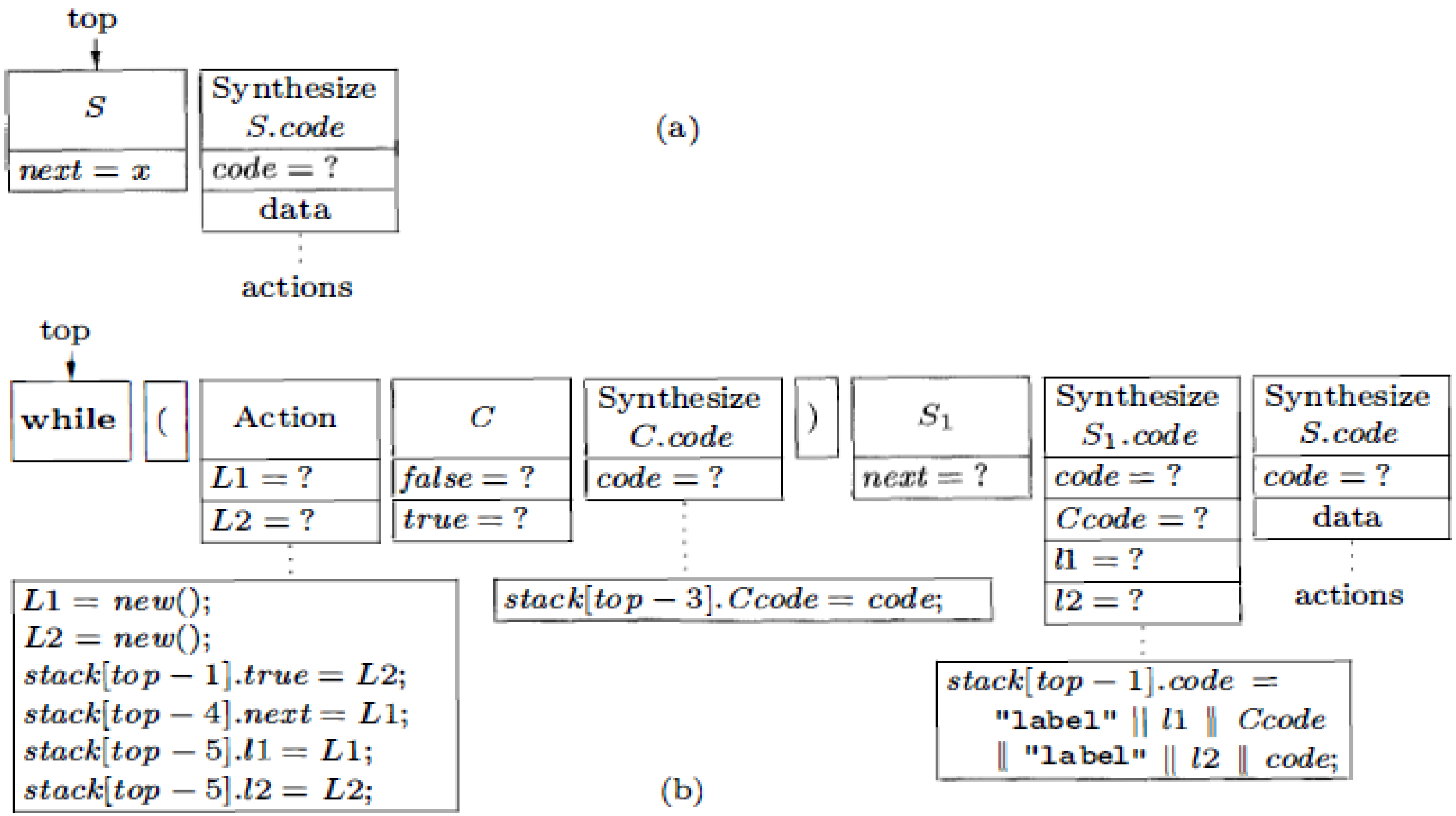
3. The inherited attribute $C.true$ labels the beginning of the code that must be executed if C is true.

4. The inherited attribute $C.false$ labels the beginning of the code that must be executed if C is false.

5. The synthesized attribute $C.code$ is the sequence of intermediate-code steps that implements the condition C and jumps either to $C.true$ or to $C.false$, depending on whether C is true or false.

Example: stack implementation

- Example: Expansion of *S with synthesized attribute constructed on the stack*



Implementing L-attributed SDD *in conjunction with an LR-parser*

Moving Actions to the End of Productions

One essential for bottom-up parsing is that actions take place only at reduction time, so all actions must be at the right end of productions. Thus, if we have a production like:

$$A \rightarrow B \, \{ \text{ action } \} \, C$$

we introduce a marker M, rewriting the SDT as

$$\begin{aligned} A &\rightarrow BMC \\ M &\rightarrow \epsilon \, \{ \text{ action } \} \end{aligned}$$

Keeping Inherited Attributes Immediately Below Their Nonterminal

The second necessary trick is to keep each inherited attribute of some nonterminal, say A, immediately below A on the stack; that is, it is associated with the grammar symbol immediately to the left of A in a sentential form.

- We must have these attributes available *before* we reduce to A, in fact, immediately before we start to reduce an input substring to A.
- It is possible to keep the needed attributes more than one position below that of A on the stack, but the position must not depend on what is below A on the stack.

Example:

Suppose we have production $A \rightarrow BC$. Inherited attributes for B can only depend on inherited attributes of A. If we have a rule $B.i := f(A.i)$, we can introduce a marker M with rules:

$$
\begin{aligned}
&A \rightarrow MBC \\
&M \rightarrow \epsilon \\
&\qquad \{\ M.i := A.i;\ M.s := f(M.i)\ \}
\end{aligned}
$$

Now $B.i$ is available on the stack (as $M.s$) immediately below where B will eventually appear, even though we have just now begun to recognize B.

Similarly, an inherited attribute of C can depend on inherited attributes of A and all attributes of B.

- Inherited attributes of A can be copied to a new marker N, preceding C.
- Attributes of B will appear with B on the stack before we begin the recognition of C. They also can be copied to N when we reduce ϵ to N.

Problems in implementing during parsing

- It is impossible to implement this SDT during either topdown or bottom-up parsing, because the parser would have to perform critical actions, like printing instances of * or +, long before it knows whether these symbols will appear in its input.

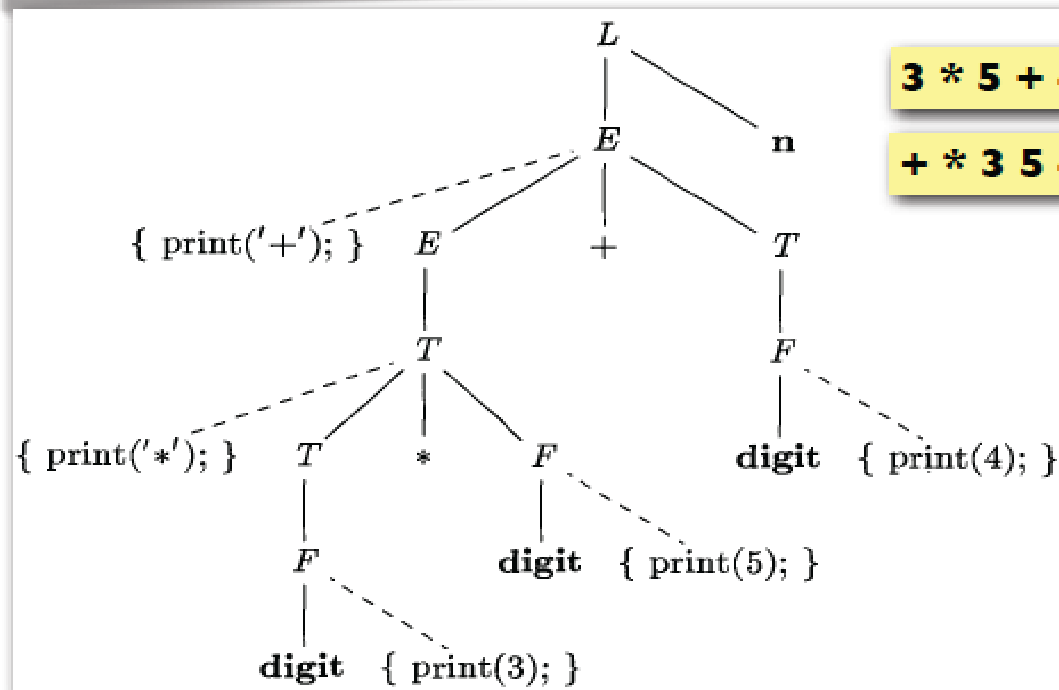
$$
\begin{array}{lllll}
1) & L & \rightarrow & E\ \mathbf{n} \\
2) & E & \rightarrow & \{\ \text{print}('+');\ \}\ \ E_1 + T \\
3) & E & \rightarrow & T \\
4) & T & \rightarrow & \{\ \text{print}('*');\ \}\ \ T_1 * F \\
5) & T & \rightarrow & F \\
6) & F & \rightarrow & (\ E\ ) \\
7) & F & \rightarrow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

Problematic SDT for infix-to-prefix translation during parsing

Implementation of SDT

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 → α (α=β{**a**}δ) 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 a 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.



3 * 5 + 4

+ * 3 5 4

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