UNIT – 5

SYNTAX-DIRECTED TRANSLATION

-Compiled by: Namratha Nayak

Introduction

- Associate information with a language construct by attaching attributes to the grammar symbols
- A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions

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

$E \rightarrow E_1 + T \quad \{ \text{print } '+' \}$

- Syntax-directed definitions are more readable
- Most general approach to syntax-directed translation is :
  - Construct a parse tree or syntax tree
  - Compute the values of attributes at the nodes of the tree by visiting nodes of the tree

Syntax-Directed Definitions

- A *syntax-directed definition* (SDD) is a CFG together with attributes and rules
  - *Attributes* are associated with grammar symbols
  - *Rules* are associated with productions to compute values for attributes
  - If X is a symbol and a is one of its attributes, we $X.a$ denotes the value of a at a particular parse tree node labeled X
  - Nodes in a grammar are records with fields for holding information
- Attributes may be of any kind :numbers, types, table references, or strings
- Two types of attributes for nonterminals
  - Synthesized attribute
  - Inherited attribute

Inherited and Synthesized Attributes

- A *synthesized attribute* for a nonterminal *A* at a parse-tree node *N* is defined by the semantic rule associated with the production at *N*
  - The production must have *A* as its head
  - A synthesized attribute at node *N* is defined 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 the semantic rule associated with the production at the parent of
  - 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
- Synthesized attributes can be defined in terms of inherited attribute values at node *N* itself
- Terminals can have synthesized attributes but not inherited attributes

Inherited and Synthesized Attributes

- Syntax-Directed Definition of a simple desk calculator

	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

  - Each of the nonterminals has a single synthesized attribute, called *val*
  - Suppose that the terminal **digit** has a synthesized attribute *lexval*
  - An SDD that involves only synthesized attributes is called *S-attributed*
    - Each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production

Evaluating an SDD at the Nodes of a Parse Tree

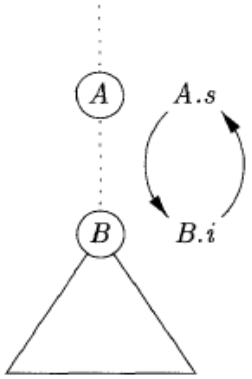
- The rules of an SDD are applied first by constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes
- A parse tree showing the values of its attributes is called an *annotated parse tree*
- In what order do we evaluate attributes?
  - Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends
  - In previous example, if all attributes are synthesized , then we must evaluate the *val* attributes at all of the children of a node before we can evaluate the *val* attribute at the node itself
  - With synthesized attributes, we can evaluate attributes in any bottom-up order

Evaluating an SDD at the Nodes of a Parse Tree

- For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order to evaluate attributes
  - Consider nonterminals A and B, with synthesized and inherited attributes $A.s$ and $B.i$, along with the production and rules

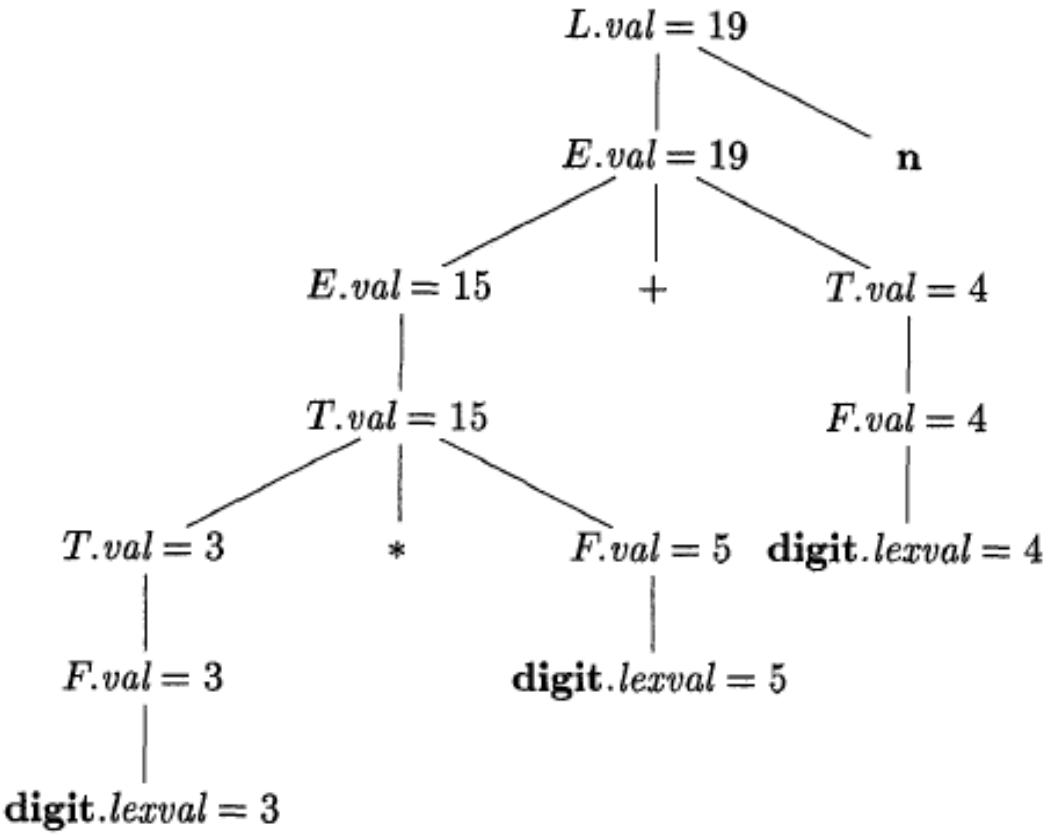
PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$
	$B.i = A.s + 1$

  - The rules are circular; it is impossible to evaluate either $A.s$ at node N or $B.i$ at the child of N without first evaluating the other



Evaluating an SDD at the Nodes of a Parse Tree

- Annotated parse tree for the input string 3 * 5 + 4n



Evaluating an SDD at the Nodes of a Parse Tree

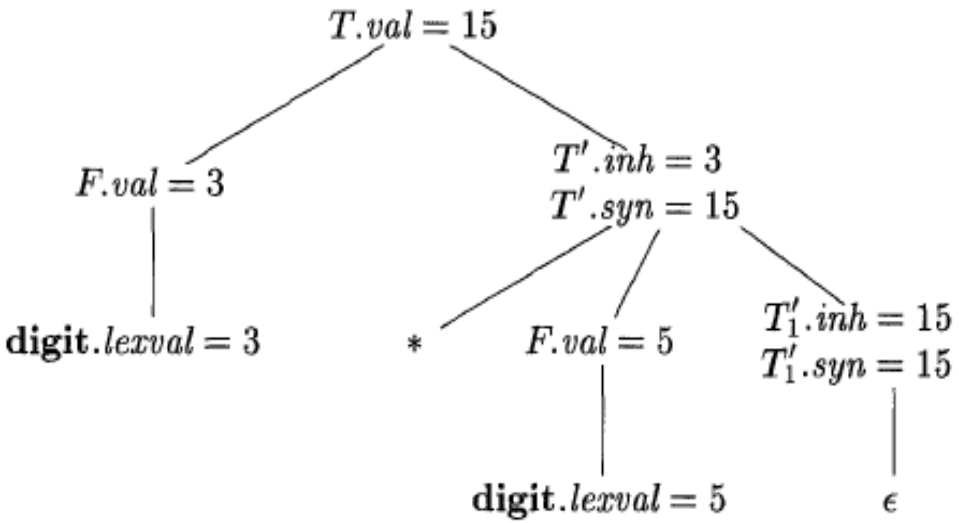
- The SDD computes terms like 3*5 and 3*5*7
  - Top-down parse of input 3*5 begins with $T \rightarrow FT'$; here F generates 3 but the operator * is generated by T'
  - Operand 3 appears in a different sub tree of the parse tree from *, so an inherited attribute will be used to pass the operand to the operator

	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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

  - Each of the nonterminals T and F has a synthesized attribute *val;* the nonterminal T' has two attributes: an inherited and synthesized attribute

Evaluating an SDD at the Nodes of a Parse Tree

- Annotated Parse tree for 3*5



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
- Define two important classes of SDD's
  - The "S-attributed" SDD
  - The "L-attributed" SDD

Dependency Graphs

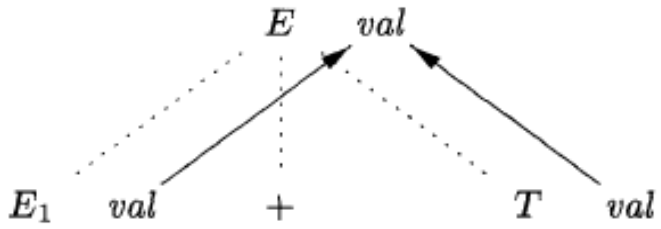
- A dependency graph depicts the flow of information among the attribute instances in a particular parse tree
  - An edge from one attribute instance to another means that the value of the first is needed to compute the second
  - Edges express constraints implied by the semantic rules
- In detail:
  - For each parse-tree node, say a node labeled by grammar symbol *X*, the graph has a node for each attribute associated with *X*
  - Suppose that 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 the dependency graph has an edge from *X.c* to *A.b*
  - Suppose that a semantic rule associated with a production *p* 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.a* to *B.c*

Dependency Graphs

- **Example 1:**

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

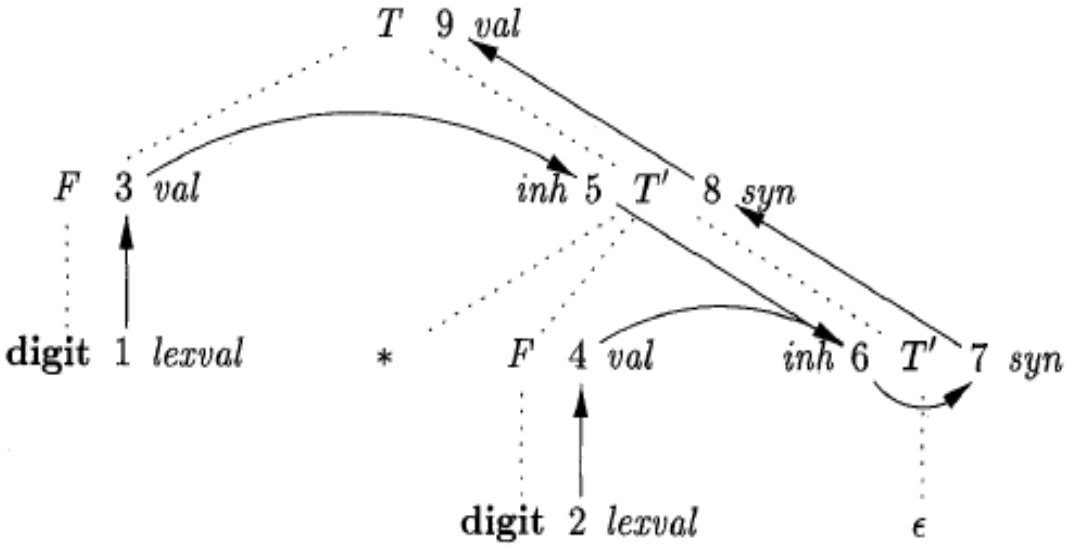
- Portion of the dependency graph for every parse tree where this production is used



  - Solid lines represent dependency graph edges
  - Dotted lines are the parse tree edges

Dependency Graphs

- Example 2 : Dependency graph for the annotated parse tree of 3*5



Ordering the Evaluation of Attributes

- Dependency graph characterizes the orders in which we can evaluate the attributes at the various nodes
  - If there is an edge from node *M* to node *N*, then the attribute corresponding to *M* must be evaluated before the attribute of *N*
  - Thus, the only allowable orders of evaluation are those sequence of nodes $N_1, N_2, \ldots\ldots, N_k$, such that if there is an edge of the graph from N_i to N_j, then $i < j$
  - Such an ordering is called a *topological sort* of the graph
- If there is any cycle in the graph, then there are no topological sorts
- The previous dependency graph has no cycles
  - One topological sort is 1,2,…..,9 and other is 1, 3, 5, 2, 4, 6, 7, 8, 9

S-Attributed Definitions

- Translations can be implemented using classes of SDD's that guarantee an evaluation order, since they do not permit cycles
- An SDD is *S-attributed* if every attribute is synthesized
- When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes
  - Evaluate attributes by performing a postorder traversal of the parse tree and evaluating the attributes at 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;
}
```

- S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal

L-Attributed Definitions

- The idea is that, between the attributes associated with a production body, dependency graph edges can go from left to right and not right to left
- Each attribute must be either
  - Synthesized, or
  - Inherited, but with rules limited as follows. Suppose there is a production A $\rightarrow X_1, X_2, \ldots, X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. 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 in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i

L-Attributed Definitions

	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)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

PRODUCTION	SEMANTIC RULE
$T \rightarrow F\ T'$	$T'.inh = F.val$
$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$

In each of the cases, the rules use information "from above or from the left". The remaining attributes are synthesized. Such an SDD is *L-attributed*

L-Attributed Definitions

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

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

  - First rule defines a synthesized attribute *A.s* in terms of an attribute at a child (that is, a symbol within the production body)
  - Second rule defines an inherited attribute *B.i*, so the entire cannot be S-attributed
  - The SDD cannot be L-attributed, because the attribute *C.c* is used to help define *B.i*, and *C* is to the right of *B* in the production body
  - While attributes at siblings in a parse tree may be used in L-attributed SDD's, they must be to the left of the symbol whose attribute is being defined

Semantic Rules with Controlled Side Effects

- Translations may involve side effects
  - A desk calculator may print a result,
  - Code generator might enter the type of an identifier into a symbol table
- Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph
- Control side effects in SDD's in one of the following ways
  - Permit incidental side effects that do not constrain attribute evaluation. That is, permit side effects when evaluation based on any topological sort of the graph produces a "correct" translation
  - Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the graph

	PRODUCTION	SEMANTIC RULE
1)	$L \rightarrow E$ **n**	$print(E.val)$

Semantic Rules with Controlled Side Effects

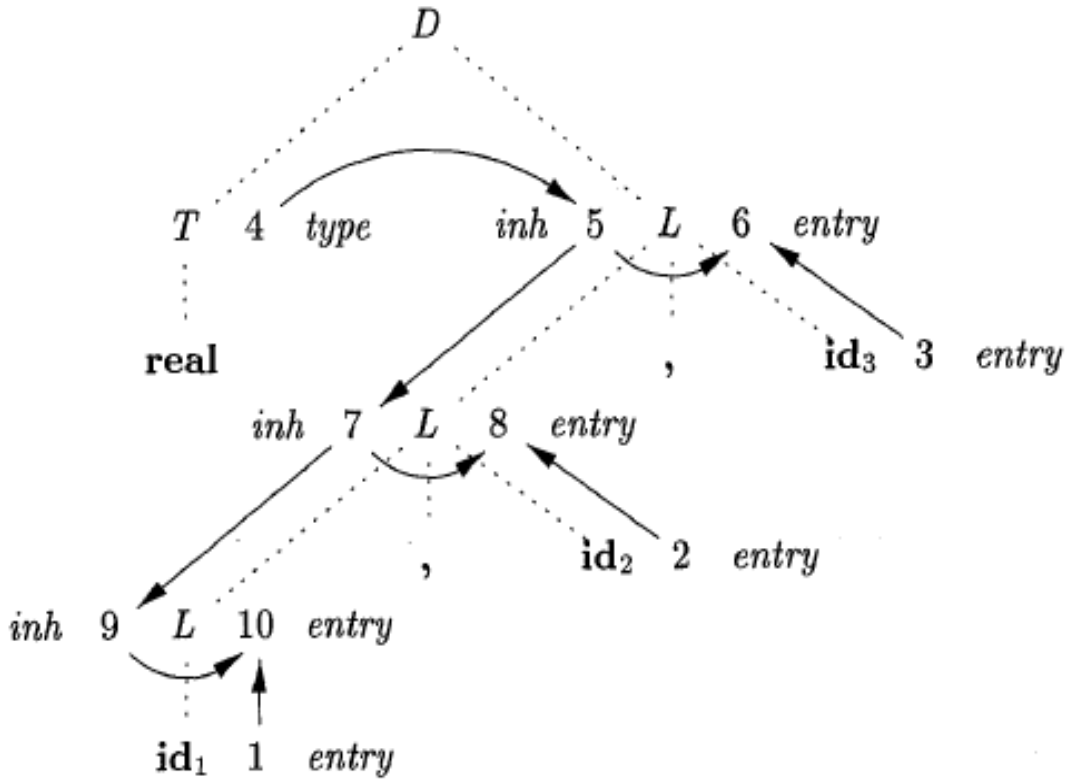
- **Example** : SDD for simple type declaration D consisting of a basic type T followed by a list L of identifiers

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

- Productions 4 and 5 have a function *addType* with two arguments
  - **id.***entry* , a lexical value that points to a symbol-table object
  - *L.inh*, the type being assigned to every identifier on the list

Semantic Rules with Controlled Side Effects

- Dependency graph for the input string **float id_1,id_2,id_3**



Applications of Syntax-Directed Translation

- Syntax-directed translation techniques will be applied to type checking and intermediate code generation
- Main application is the construction of syntax trees
  - Syntax tree is an intermediate representation, a common form of SDD turns its input string into a tree
  - To complete translation into intermediate code, compiler then walks the syntax tree, using another set of rules
- Consider two SDD's for constructing syntax trees for expressions
  - An S-attributed definition suitable for use during bottom-up parsing
  - L-attributed definition suitable for use during top-down parsing

Construction of Syntax Trees

- Each node in a syntax tree represents a construct
  - The children of the node represent meaningful components of the construct
  - For instance, a syntax-tree node representing an expression E+T has a label + and two children representing sub expressions E and T
- Implement the nodes of a syntax tree by objects with a suitable number of fields
  - Each object will have an *op* field that is the label of the node
  - The objects will have additional fields as follows:
    - If 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 node is an interior node, there are 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,….,c_k) creates an object with first filed *op* and *k* additional fields for the *k* children

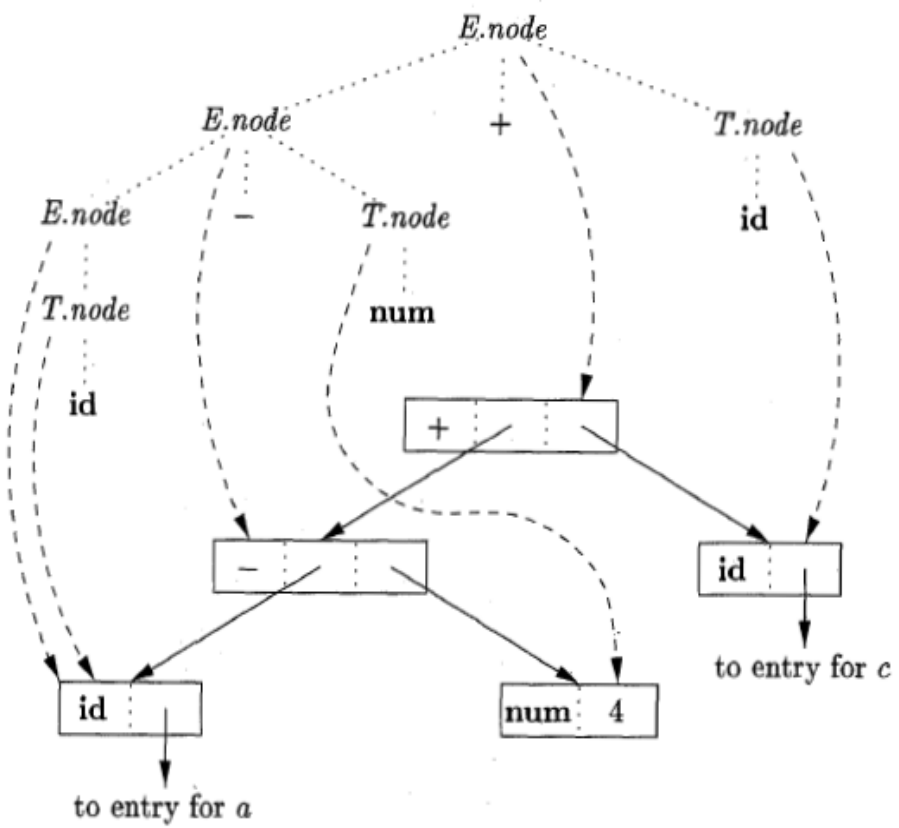
Construction of Syntax Trees

- **Example 1**: S-attributed definition for constructing syntax trees for simple expressions

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

Construction of Syntax Trees

- Construction of syntax-tree for the input a – 4 + c



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

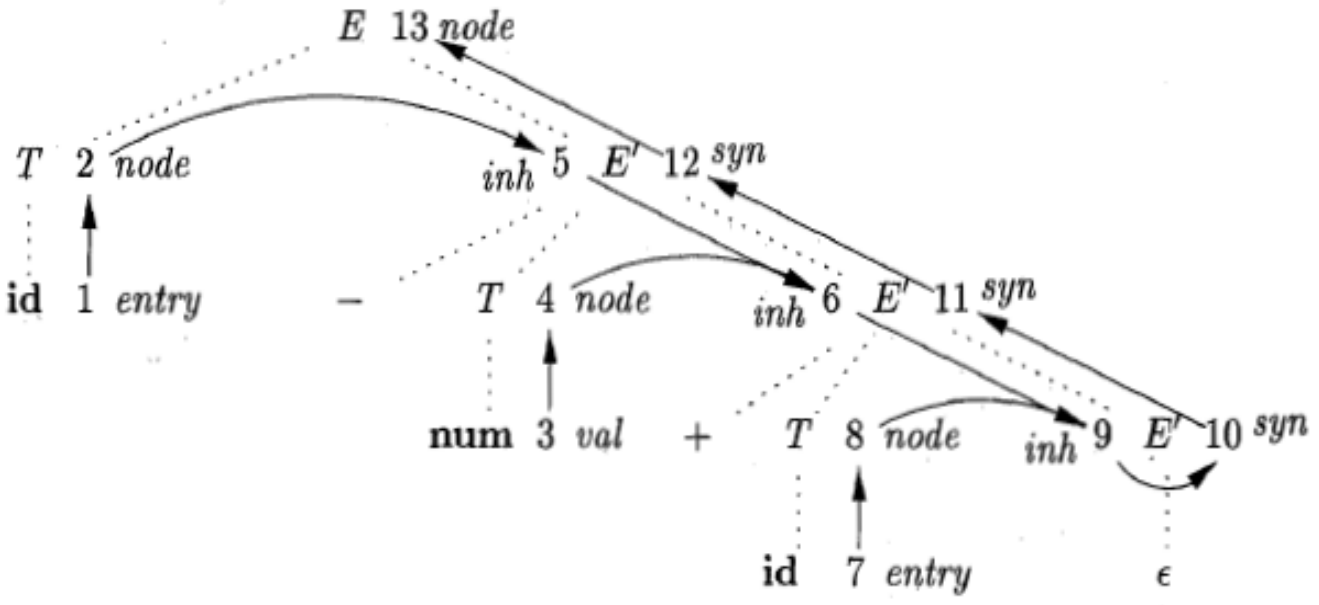
Construction of Syntax Trees

- **Example 2**: L-attributed definition for constructing syntax trees during top-down parsing

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

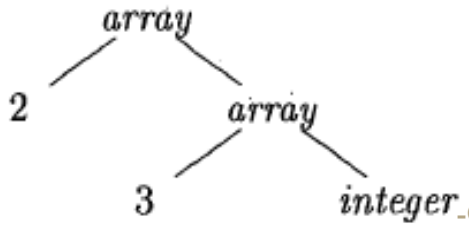
Construction of Syntax Trees

- Dependency graph for a – 4 + c



The Structure of a Type

- Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input
  - Attributes can then be used to carry information from one part of the parse tree to another
- **Example:** In C, the type **int** [2][3] can be read as, "array of 2 arrays of 3 integers.
  - Corresponding type expression is *array*(*2, array*(*3,integer*))
  - Operator *array* takes two parameters: a number and a type
  - If types are represented by trees, then this operator returns a tree node labeled *array* with two children for a number and a type



The Structure of a Type

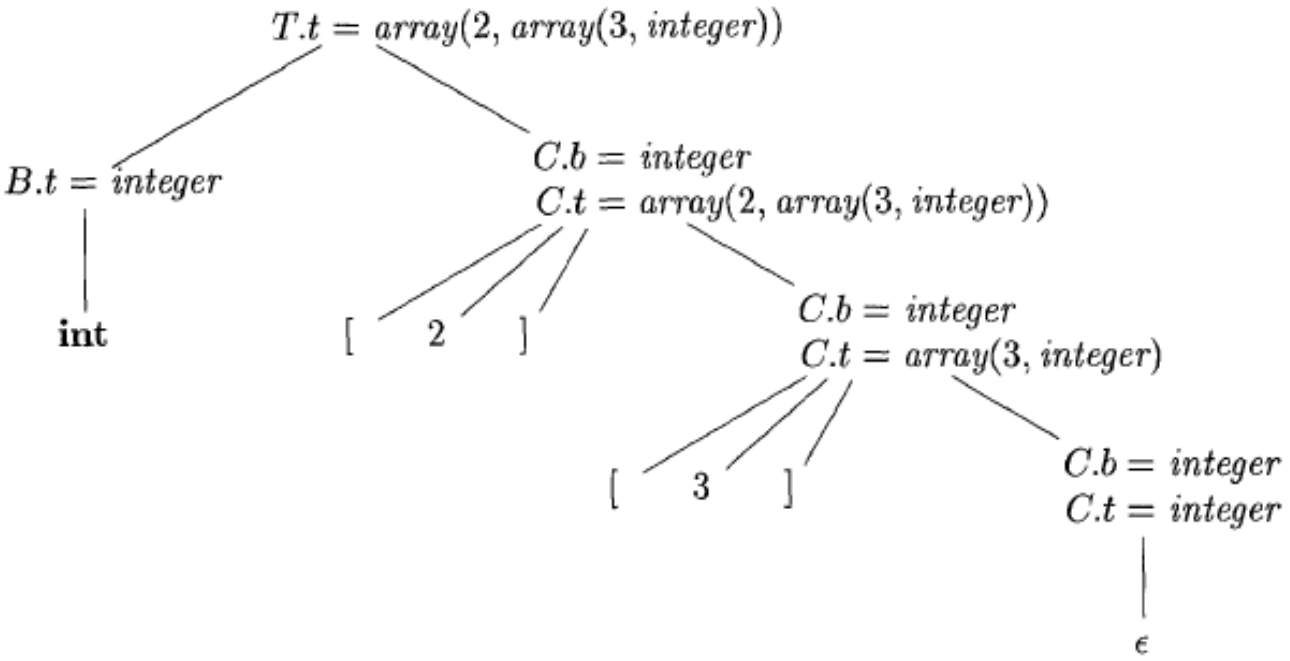
- An SDD with nonterminal T that generates either a basic type or an array type

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

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

The Structure of a Type

- An annotated parse tree for the input string **int** [2][3]



Syntax-Directed Translation Schemes

- A syntax-directed translation scheme (SDT) is a CFG with program fragments embedded within production bodies
  - The program fragments are called semantic actions and can appear at any position within the production body
  - An SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order
- Focus on the use on SDT's to implement two classes of SDD's:
  - Underlying grammar is LR-parsable, and the SDD is S-attributed
  - Underlying grammar is LL-parsable and the SDD is L-attributed
- 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, $M \rightarrow \varepsilon$

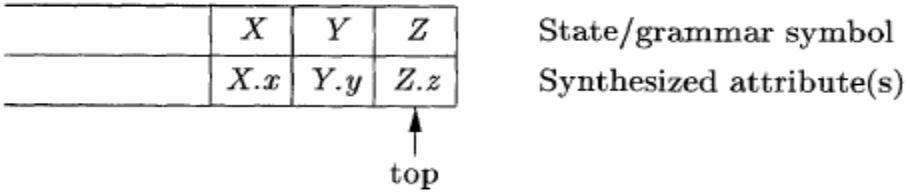
Postfix Translation Schemes

- Simplest SDD implementation occurs when the grammar is parsed bottom-up and the SDD is S-attributed
- 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*

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

Parser-Stack Implementation of Postfix SDT's

- Postfix SDT's can be implemented during LR parsing by executing the actions when the reductions occur
  - The attributes can be put on a stack in a place where they can be found during the reduction
  - Best plan is to place the attributes along with the grammar symbols in records on the stack itself



  - Here parser stack contains records with a field for a grammar symbol, and below it a field for an attribute

Parser-Stack Implementation of Postfix SDT's

- We can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack
- If attributes are all synthesized, and actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head
- If we reduce a production such as , $A \rightarrow XYZ$, then we have all the attributes of X, Y and Z at known positions on the stack
  - After the action, A and its attributes are at the top of the stack, in the position of the record for X

Parser-Stack Implementation of Postfix SDT's

- Implementing the desk calculator in a bottom-up parsing stack

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

SDT's with Actions Inside Productions

- An action may be placed at any position within the body of a production
  - It is performed immediately after all symbols to its left are processed
  - So, if we have a production $B \rightarrow X \{a\} Y$, the 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 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 parse is top-down, we perform a just before we attempt to expand this occurrence of Y (if Y is nonterminal) or check for Y on input(if Y is terminal)
- SDT's that can be implemented during parsing include postfix SDT's and a class of SDT's that implement L-attributed definition

SDT's with Actions Inside Productions

- **Example :** Desk-calculator that prints the prefix form of an expression, rather than evaluating the expression

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

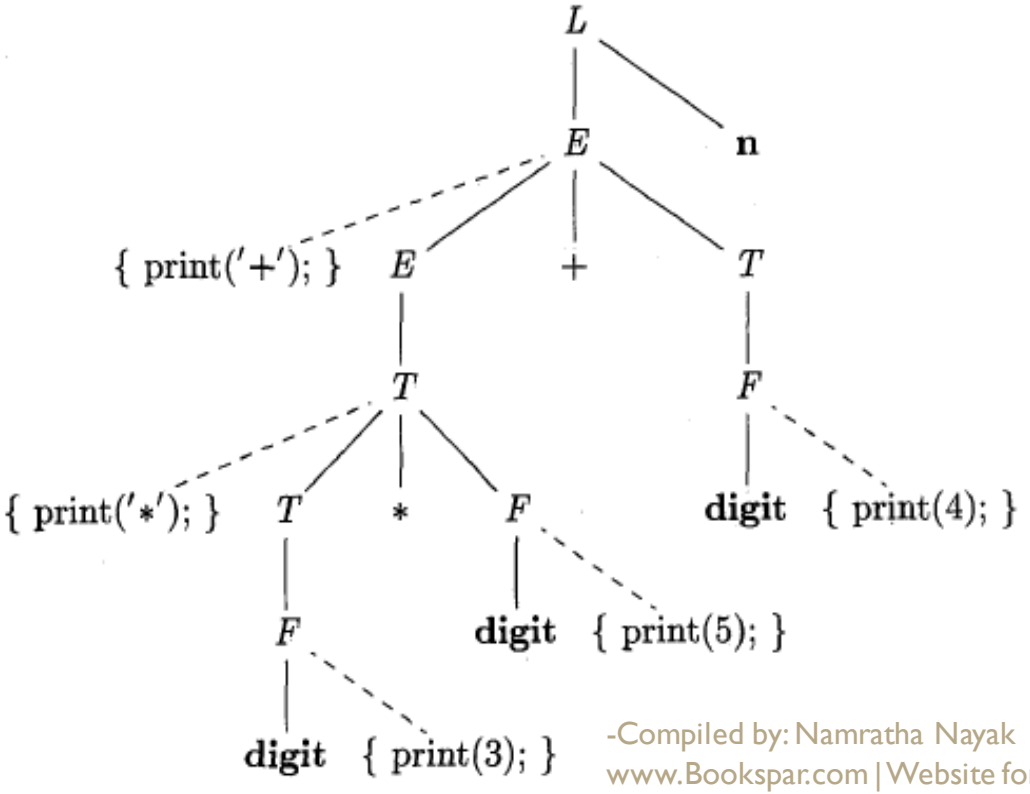
  - It is impossible to implement this SDT during either top-down or bottom-up parsing
  - Since the parser would have to perform critical actions like printing instances of * or +, long before it knows whether these symbols will appear in its input

SDT's with Actions Inside Productions

- Any SDT can be implemented as follows:
  - Ignore the actions, parse the input and produce a parse tree as result
  - Then, examine each interior node N, say one for production $A \rightarrow \alpha$. 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
  - Perform a preorder traversal of the tree and as soon as a node labeled by an action is visited, perform that action

SDT's with Actions Inside Productions

- Parse tree for expression 3 * 5 + 4 with actions inserted
- If we visit the nodes in preorder, we get the prefix form of the expression : + * 3 5 4



Eliminating Left Recursion from SDT's

- When grammar is part of the SDT, we need to worry about how the actions are handled
- Consider the simple case, where we only care about the order in which the actions in an SDT are performed
  - For example, if each action prints only strings, we care only about the order in which the strings are printed
- The following principles can guide us:
  - When transforming the grammar, treat the actions as if they were terminal symbols
  - The principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string
  - The actions are executed in the same order in any left-to-right parse, top-down or bottom-up

Eliminating Left Recursion from SDT's

- Consider the following E-productions from an SDT for translating infix expressions into postfix notation

$$\begin{aligned} E &\rightarrow E_1 + T \quad \{ \text{print}('+'); \} \\ E &\rightarrow T \end{aligned}$$

- Apply standard transformation to E, the remainder of the left-recursive production

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

  - And β, the body of the other production is T
- If we introduce R for the remainder of E, we get

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

Eliminating Left Recursion from SDT's

- When the actions of an SDD compute attributes rather than merely printing the output, must be more careful about how we eliminate left-recursion from a grammar
  - If the SDD is S-attributed, then we always construct an SDT by placing attribute-computing actions at appropriate positions in the new productions
- Suppose that the two productions are

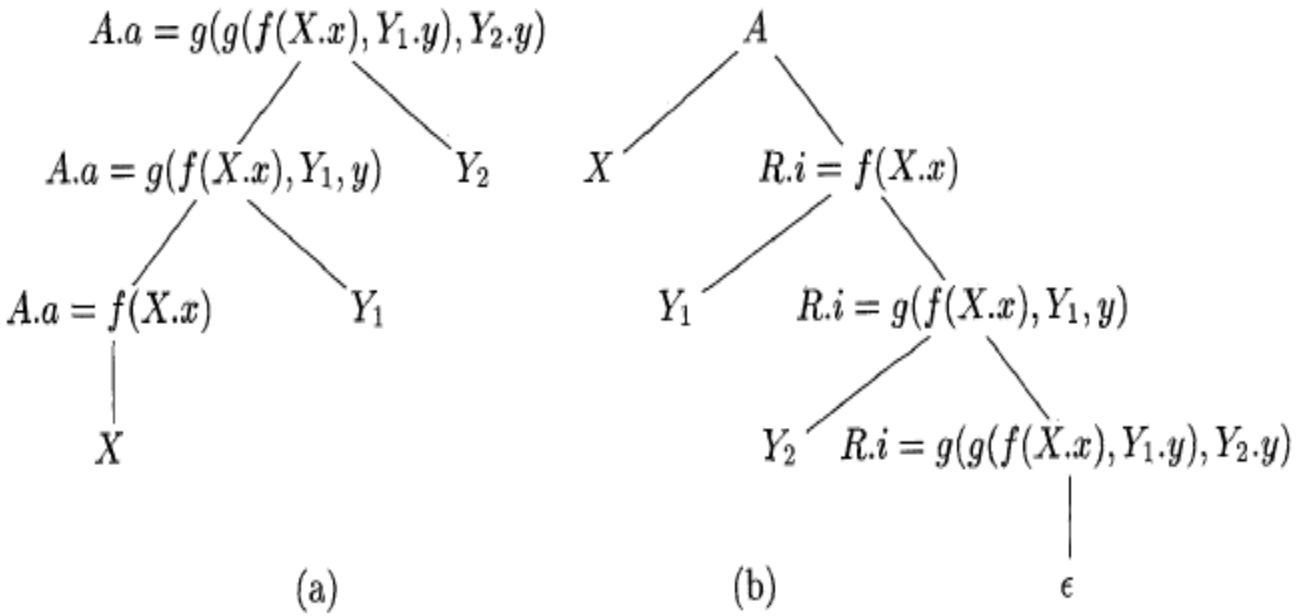
$$\begin{aligned} A &\rightarrow A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\ A &\rightarrow X\ \{A.a = f(X.x)\} \end{aligned}$$

  - Here, *A.a* is synthesized attribute of the left-recursive nonterminal *A*, and *X* and *Y* are symbols with synthesized attributes *X.x* and *Y.y*
  - The schema has an arbitrary function *g* computing *A.a* in the recursive production and an arbitrary function *f* computing *A.a*
  - In each case, *f* and *g* take as arguments whatever attributes they are allowed to access if the SDD is S-attributed

Eliminating Left Recursion from SDT's

- Turn the grammar into

$$A \rightarrow X\ R$$
$$R \rightarrow Y\ R \mid \epsilon$$



Eliminating Left Recursion from SDT's

- SDT to accomplish the translation in (b)

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

  - The inherited attribute *R.i* is evaluated immediately before a use of *R* in the body, while the synthesized attributes *A.a* and *R.s* are evaluated at the ends of the productions
  - Thus, whatever values are needed to compute these attributes will be available from what has been computed to the left

SDT's for L-Atrributed Definitions

- Rules for turning an L-atrributed SDD into an SDT are :
  - Embed the action that computes the inherited attributes for a nonterminal *A* immediately before the occurrence of *A* in the body of the production
  - Place the actions that compute a synthesized attribute for the head of a production at the end of the body of the production
- These principles are illustrated with the help of two examples
  - First involves typesetting – illustrates how the techniques of compiling can be used in language processing for applications
  - Second is about the generation of intermediate code for a typical programming language construct : a form of while-statement

SDT's for L-Atrributed Definitions

- **Example 1**: Motivated by languages for typesetting mathematical formulas
  - Concentrate on capability to define subscripts
  - A simple grammar for boxes (elements of text bounded by a rectangle) is:

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

  - A box can be either:
    - Two boxes, juxtaposed, with the first, B_1, to the left of the other B_2
    - A box and a subscript box. The second box appears in a smaller size, lower, and to the right of the first box
    - A parenthesized box, for grouping boxes and subscripts
    - A text string, that is, any string of characters
  - The grammar is ambiguous, but can be used to parse bottom-up if we make subscripting and juxtaposition right associative with **sub** taking precedence

SDT's for L-Atrributed Definitions

- Constructing larger boxes from smaller ones

- Values associated with the vertical geometry of boxes are:
  - *Point size* used to set text within a box. Inherited attribute *B.ps* represents the point size of box *B*.
  - Each box has a *baseline*, which is the vertical position that corresponds to the bottom of lines of text.
  - A box has a *height*, which is the distance from the top of the box to the baseline. Synthesized attribute *B.ht* gives the height of box *B*.
  - A box has a *depth*, which is the distance from the baseline to the bottom of the box. Synthesized attribute *B.dp* gives the depth of box *B*.

SDT's for L-Atrributed Definitions

- SDT for typesetting boxes

	PRODUCTION	ACTIONS
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)	$B \rightarrow B_1$ **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)	$B \rightarrow$ **text**	$\{\ B.ht = getHt(B.ps, \textbf{text}.lexval);$ $B.dp = getDp(B.ps, \textbf{text}.lexval);\ \}$

SDT's for L-Atrributed Definitions

- SDD for computing point sizes, heights and depths

	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)	$B \rightarrow B_1$ **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)	$B \rightarrow$ **text**	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

SDT's for L-Atrributed Definitions

- **Example 2:** A simple while-statement and the generation of intermediate code
  - For this we need only one production

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

    - *S* is the nonterminal that generates all kinds of statements
    - C is the conditional expression – a boolean expression that evaluates to true or false
  - We generate explicit instructions of the form **label** *L*, where *L* is an identifier, to indicate that *L* is the label of the instruction that follows
  - Meaning of the while-statement is that the conditional *C* is evaluated
    - If true, control goes to the beginning of the code for S_1
    - If false, control goes to the code that follows the while-statement's code

SDT's for L-Atrributed Definitions

- The following attributes are used to generate proper intermediate code
  - Inherited attribute *S.next* labels the beginning of the code that must be executed after *S* is finished
  - Synthesized attribute *S.code* is the sequence of intermediate-code steps that implements a statement *S* and ends with a jump to *S.next*
  - Inherited attribute *C.true* labels the beginning of the code that must be executed if *C* is true
  - Inherited attribute *C.false* labels the beginning of the code that must be executed if *C* is false
  - 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*

SDT's for L-Atrributed Definitions

- SDD that computes the attributes for the while-statement

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

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

  - Function *new* generates labels
  - Variables $L1$ and $L2$ hold labels that we need in the code
    - $L1$ is the beginning of the code for the while-statement and S_1 jumps here after it finishes
    - $L2$ is the beginning of the code for S_1, and it becomes the value of $C.true$
  - $C.false$ is set to $S.next$, because when the condition is false, we execute whatever code must follow the code for S
  - We use || as the symbol for concatenation of intermediate-code fragments

SDT's for L-Attributed Definitions

- SDT for while-statements

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