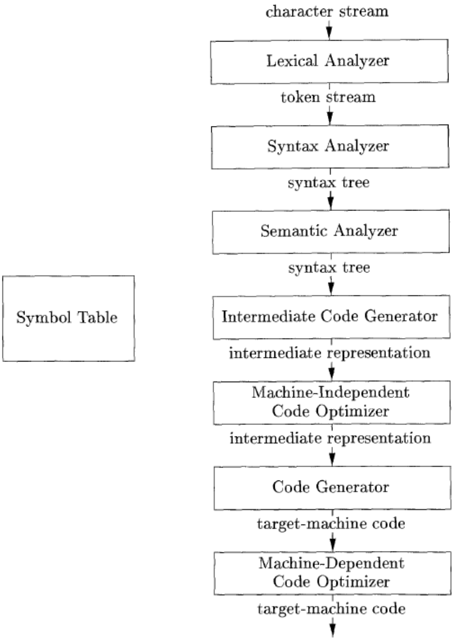
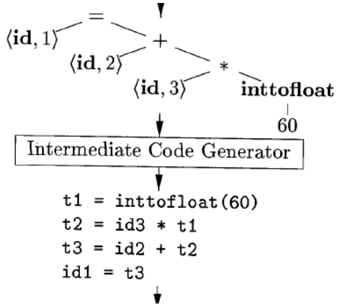
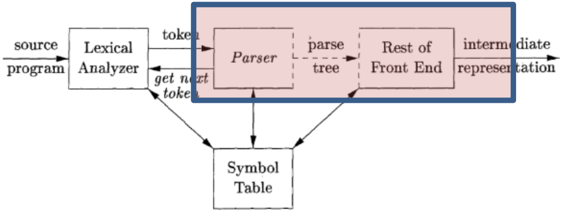
Syntax-Directed Translation

The Phases of a Compiler



Syntax-Directed Translation





- **Semantic analysis** and **translation** actions can be **interlinked** with parsing
- Implemented as a **single module**.

Syntax-Directed Translation

- **Translation** of languages **guided** by **context-free grammars**.

- Attach ***attributes*** to the **grammar symbol**

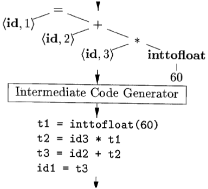
- **Syntax-directed definition** specifies the **values of attributes**
    - By associating **semantic rules** with the **grammar productions**

Syntax-Directed Translation

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

- If ***X*** is a **grammar symbol** and ***a*** is one of its **attributes**,
  - ***X.a*** denotes the value of the attribute *X.*

- Attributes may be
  - numbers, types, table references, or strings,
  - Strings may even be code in the intermediate language.



Attributes

Synthesized attribute:

- *Synthesized attribute* for a **nonterminal *A*** at a parse-tree node *N* is defined by
- **Semantic rule** associated with the **production at *N*.**
- The production must have ***A* as its head**.

- A synthesized attribute at node *N* is defined only in terms of attribute values at the **children of *N* and at *N* itself**.

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

Attributes

Inherited attribute:

- Inherited attribute for a **nonterminal B** at a parse-tree node N is defined by

- **Semantic rule** associated with the **production at the parent of N**

- Note that the production must have **B as a symbol in its body**.

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

$T \to F\, T'$	$T'.inh = F.val$
$T' \to *\, F\, T'_1$	$T'_1.inh = T'.inh \times F.val$

Attributes

- **Synthesized attribute** at node N to be **defined** in terms of **inherited attribute** values at node **N itself**.

$$T' \to \epsilon \qquad \Big| \quad T'.syn = T'.inh$$

- **Do not allow** an **inherited attribute** at node *N* to be defined in terms of attribute values at the **children of node *N***

- **Terminals** can have **synthesized attributes**, but not inherited attributes.
- **Attributes for terminals** have **lexical values** that are supplied by the **lexical analyzer**

$$F \to \mathbf{digit} \qquad \Big| \quad F.val = \mathbf{digit}.\text{lexval}$$

Example of SDD

Each of the Non-terminals has a **single synthesized attribute**, called ***val***

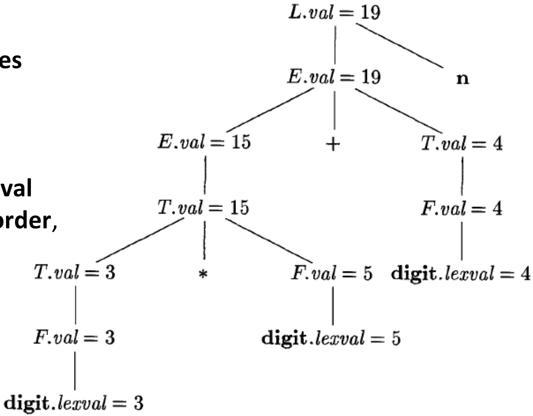
	Production	Semantic Rules
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)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

Annotated parse tree.

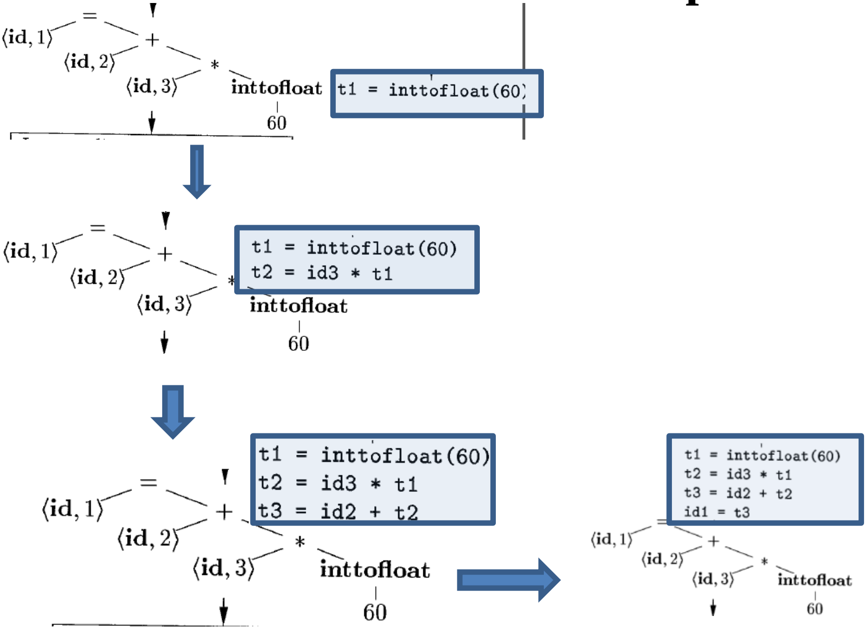
A **parse tree**, showing the **value(s) of its attribute(s)** is called an ***annotated*** *parse tree.*

Input string: **3 * 5 + 4 n**

- We show the resulting **values associated** with **each node**.
- Each of the nodes for the nonterminals has **attribute val** computed in a **bottom-up order**,



Annotation and Evaluation of parse tree

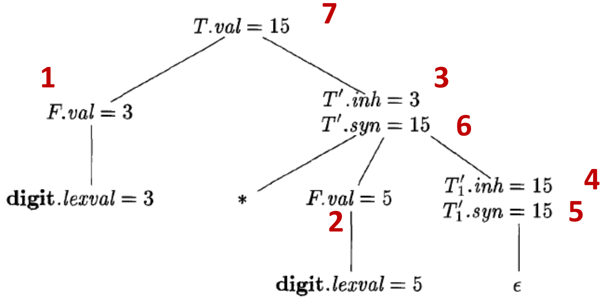


Annotated parse tree.

	Production	Semantic Rules
1)	$T \to F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\, F\, T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

val and **syn**: Synthesized
inh: Inherited

Annotated parse tree for 3 * 5



Evaluation Orders of SDD

- "**Dependency graphs**" are a useful tool for determining an **evaluation order** for the **attribute** instances in a given parse tree.
    - Depicts the **flow of information** among the attribute instances in a particular parse tree
    - **Directed edges**

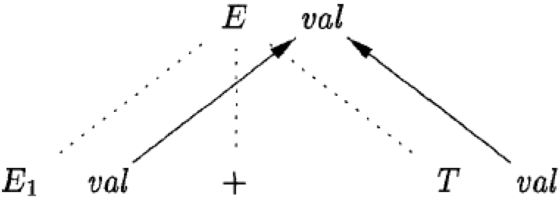
- For a **node A in parse tree** -> **node A in dependency graph**

A has a **synthesized** attribute **b**

Production	**Semantic Rule**
A->…X..	A.b=f(.., X.c, ..)

- **Edge** from X.c to A.b
    - Edge from **child attribute** to **parent attribute**

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$



Evaluation Orders of SDD

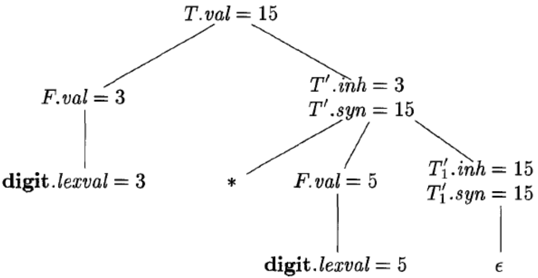
- "Dependency graphs" are a useful tool for determining an **evaluation order** for the attribute instances in a given parse tree.
  - Depicts the flow of information among the attribute instances in a particular parse tree
  - Directed edges
- For a **node A in parse tree** -> **node A in dependency graph**

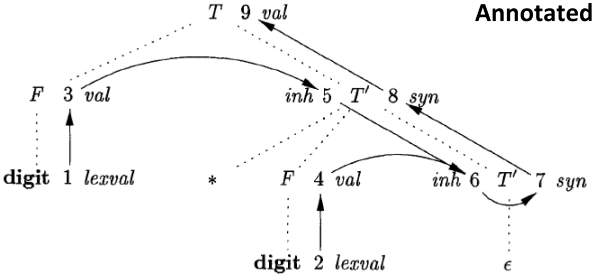
B has an inherited attribute **c**

Production	**Semantic Rule**
A->...B..X..	B.c=f(.., X.a, ..)

- **Edge** from **X.a to B.c**
  - Edge from **attribute a of X** (parent or sibling of B) to **attribute c of B** (body of the production)

	Production	Semantic Rules
1)	$T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$





Annotated parse tree for 3 * 5

Ordering the Evaluation of Attributes

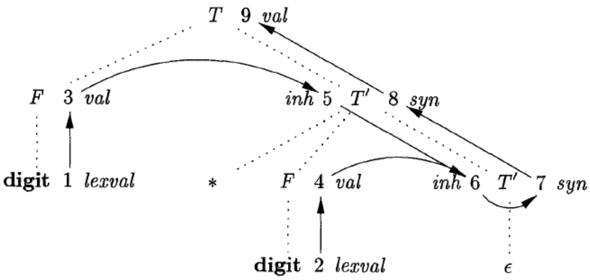
- The **dependency graph** characterizes the possible **evaluation orders**
  - In which we can **evaluate the attributes** at the various nodes of a parse tree.

- If the dependency graph has an **edge from node M to node N**,
  - Attribute corresponding to **M must be evaluated before** the attribute of N.

- If there is an edge of the dependency graph from **Ni to Nj, such that i < j**
  - the only allowable orders of evaluation are those sequences of nodes **N1, N2,... ,Nk**

- Embeds a directed graph into a linear order, and is called a **topological sort** of the graph

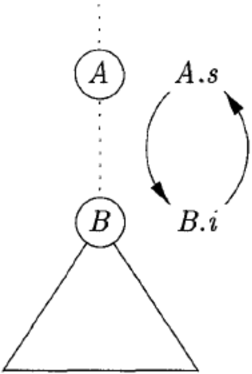
Topological Sort- Ordering the Evaluation



- One **topological sort** is the order in which the nodes have already been numbered: 1,2,... ,9.
- There are other topological sorts as well, such as 1,3,5,2,4,6,7,8,9.

Ordering the Evaluation – Cycles

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



These rules are circular; it is impossible to evaluate either *A.s* or *B.i*

Classes of SDD

(a) S-Attributed Definitions
(b) L-Attributed Definitions

Guarantee an evaluation order

S-Attributed SDD

An SDD is *S-attributed* if **every attribute is synthesized**.

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

L-Attributed SDD

- The idea behind L-attributed SDD class is that,
  - **Between the attributes** associated with a **production body**, **dependency-graph edges can go from left to right**,
  - But not from right to left (hence "L-attributed")

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \to X_1 X_2 \cdots X_n$, 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.

L-Attributed SDD

	Production	Semantic Rules
1)	$T \to F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\, F\, T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

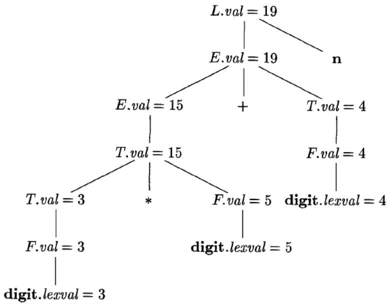
Production	Semantic Rules
$A \to B\ C$	$A.s = B.b;$ $B.i = f(C.c, A.s)$

Side Effects

- Print a result,
- Enter the type of an identifier into a symbol table.

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

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

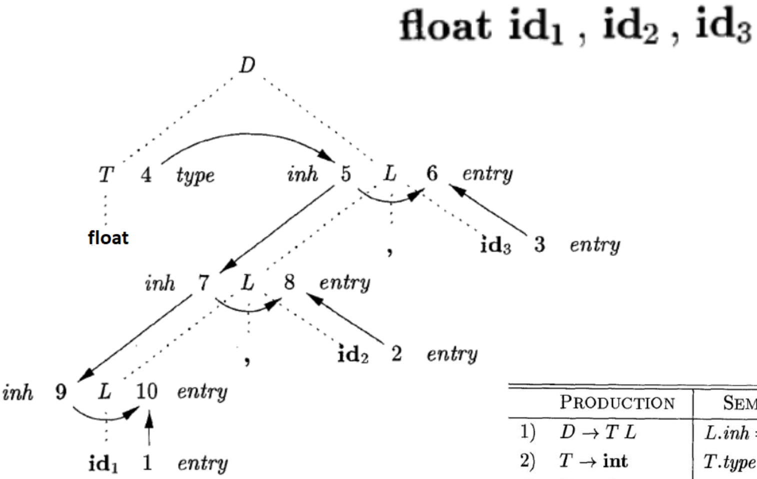


Side Effects – examples

- The SDD takes a **simple declaration** D consisting of a **basic type T** followed by a **list L of identifiers**.
- T can be **int or float**.
- For each identifier on the list, the **type is entered into the symbol-table** entry for the identifier.

	PRODUCTION	SEMANTIC RULES	
1)	$D \rightarrow T\ L$	$L.inh = T.type$	**The type is passed to the attribute L.inh**
2)	$T \rightarrow \textbf{int}$	$T.type = \text{integer}$	**Evaluate the synthesized attribute T.type, giving it the appropriate value, integer or float.**
3)	$T \rightarrow \textbf{float}$	$T.type = \text{float}$	
4)	$L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$	**Passes L.inh down the parse tree**
		$addType(\textbf{id}.entry, L.inh)$	**Function addType() properly installs the type L.inh as the type of the identifier.**
5)	$L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$	

Side Effects



	Production	Semantic Rules
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to \mathbf{int}$	$T.type = \text{integer}$
3)	$T \to \mathbf{float}$	$T.type = \text{float}$
4)	$L \to L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \to \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

Declaration statement

Representing data types: **Type Expressions**

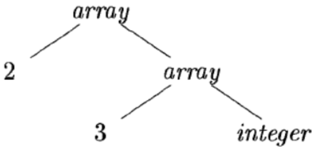
Types have structure, which we shall represent using type expressions.

- A **type expression is either a basic type** (*boolean, char, integer, float,* and *void*)

  or
- is formed by **applying an operator** called a **type constructor** to a type expression.
- A **type expression** can be formed by applying the **array type constructor** to a **number and a type expression**.

Declaration statement

- The array type int [2] [3] can be read as "array of 2 arrays of 3 integers each"
- Can be represented as a **type expression** array(2, array(3, integer)).
- This type is represented by the tree.



- The **operator array** takes **two parameters**, a number and a type.
  - Here the **type expression** can be formed by applying the **array type constructor** to a number and a type expression.

Declaration statement Example SDD

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

Type Expressions

- Nonterminal T generates either a **basic type** or an **array type**.
- Nonterminal B generates one of the basic types int and float.
- T generates a basic type when C derives €.
- Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.

Declaration statement Example SDD

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

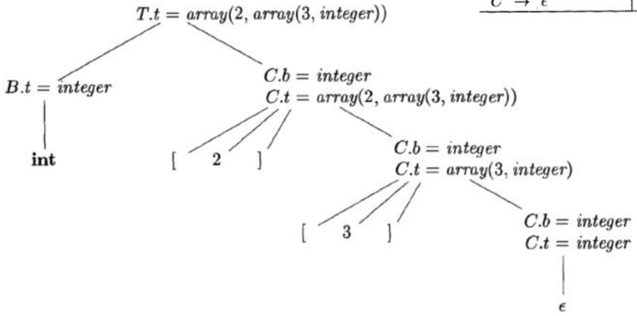
Type Expressions

- The nonterminals *B* and *T* have a synthesized attribute *t* representing a type.
- The nonterminal *C* has two attributes: an inherited attribute *b* and a synthesized attribute *t*.

Declaration statement Example SDD

input string 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**	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

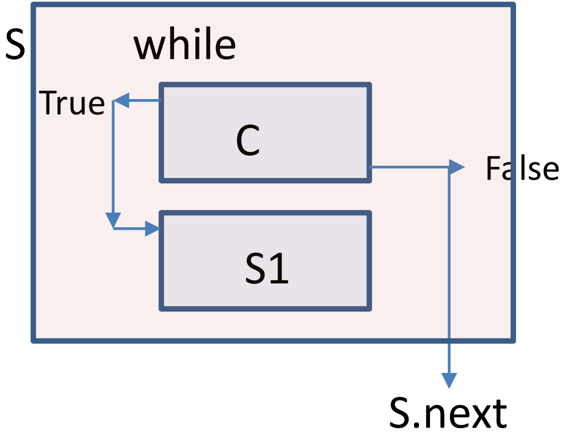


- The nonterminal *C* has two attributes: an inherited attribute *b* and a synthesized attribute *t.*
- The inherited *b* attributes pass a basic type down the tree, and the synthesized *t* attributes accumulate the result.

While() statement – Translation

Synthesized attribute: code (S)
Inherited attribute: next (S),
true (C), false (C)

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

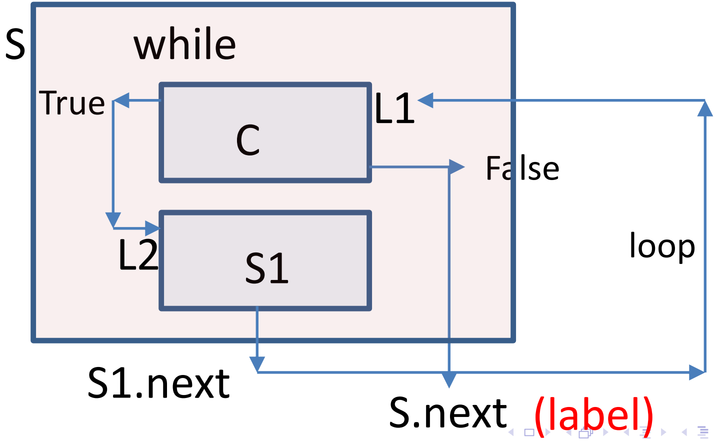


While() statement – Translation

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

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

SDD



Syntax-Directed Translation Schemes

- **Syntax-directed translation schemes** are a **complementary** notation to syntax directed definitions.
- All of the applications of syntax-directed definitions can be implemented using syntax-directed translation schemes.
- Syntax-directed translation scheme (SDT) is a **context free grammar** with **program fragments embedded** within **production bodies**.
- The program fragments are called **semantic actions** and can appear at any position within a production body.
- During parsing, an **action** in a production body **is executed** as soon as all the **grammar symbols to the left of the action have been matched with input**.

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. Thus, if we have a production $B \to X\ \{a\}\ Y$, the action a is done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal). More precisely,

- 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).

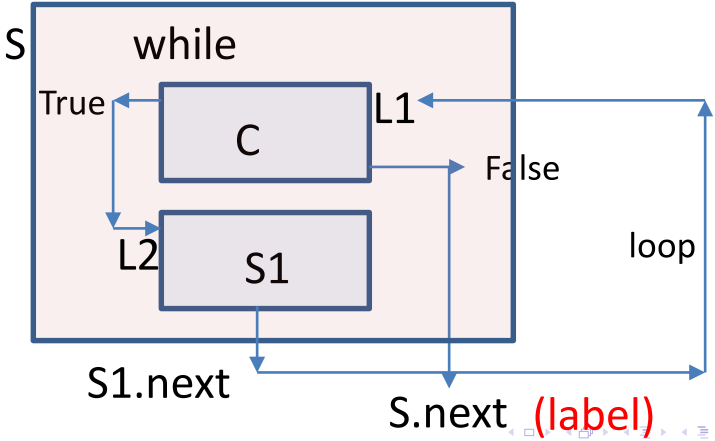
Syntax-Directed Translation Schemes

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

- 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
    - 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.**

SDT's With Actions Inside Productions

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



Application of SDD – Syntax tree construction

- **Each node** in a syntax tree represents **a construct**; the children of the node represent the meaningful components of the construct.

- A syntax-tree node representing an expression E1 + E2 has label + and two children representing the subexpressions E1 and E2

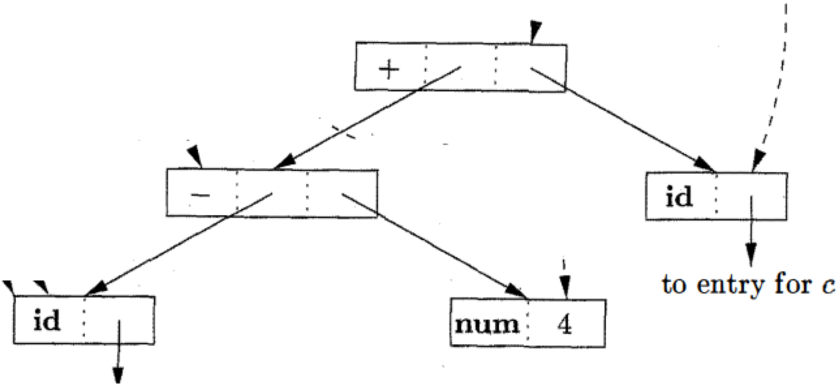
We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

- If 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 $c_1, \ldots, c_k$.

Application of SDD – Syntax tree construction

Syntax tree for $a - 4 + c$



Application of SDD – Syntax tree construction

- Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.

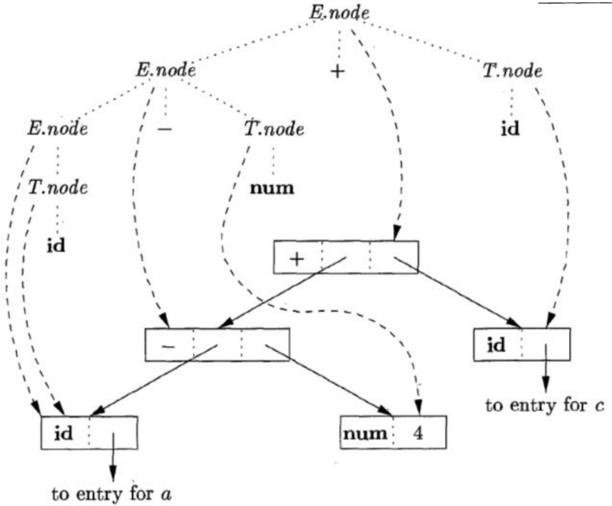
- A syntax-tree node representing an expression E1 + E2 has label + and two children representing the subexpressions E1 and E2

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

Application of SDD – Syntax tree construction

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

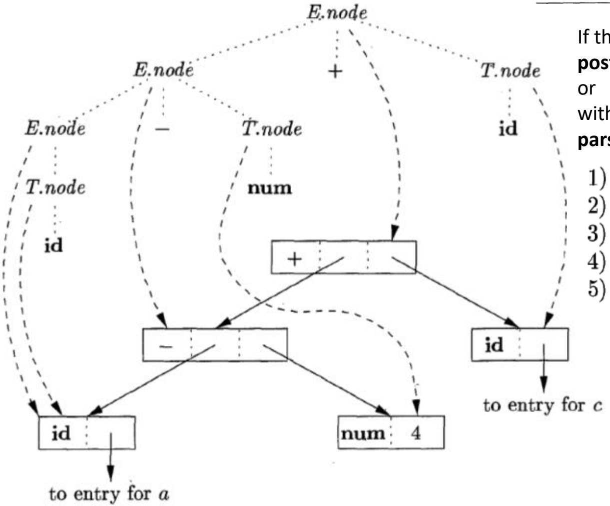
Syntax tree for $a - 4 + c$



Application of SDD – Syntax tree construction

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

Syntax tree for $a - 4 + c$



If the rules are **evaluated** during a **postorder traversal** of the parse tree, or
with reductions during a **bottom-up parse**, then the sequence of steps

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