南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 4: Syntax-Directed Translation

Yepang Liu

liuyp1@sustech.edu.cn

Why Do We Learn This Chapter?

- When implementing your compiler using Bison, you write semantic actions to construct parse trees, manage symbol tables, and perform type checking, etc.

- Did you ever have the following questions?
  - What are the theories behind the semantic actions?
  - What computations can be done?
  - What is the order of executing the code snippets? …

Outline

- Syntax-Directed Definitions
- Evaluation Orders for SDD's
- Applications of Syntax-Directed Translation (Lab)
- Syntax-Directed Translation Schemes
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)

A Brief Introduction

- **Syntax-directed translation (语法制导的翻译) is the process of language translation guided by context-free grammars**
  - Here, “language translation” is in a broad sense
    - Transforming infix expressions (中缀表达式) to postfix expressions (后缀表达式) is also viewed as “translation”
  - A language construct is typically made of smaller constructs
  - The semantics of a construct can be synthesized from its constituent constructs’ semantics
    - The type of the expression `x + y` is determined by the type of `x` and `y`, and the operator `+`
  - Or inherited from other constructs (e.g., siblings in the parse tree)
    - In “`int x`”, the type of `x` is determined by the type specifier to the left of `x`

Syntax-Directed Definitions

- A ***syntax-directed definition* (语法制导定义, SDD)** is a context-free grammar together with ***attributes*** and ***rules***
  - A set of **attributes (属性)** is associated with each grammar symbol*
    - Can be anything, e.g., data type of expressions, # instructions in the generated code
  - A **semantic rule (语义规则)** is associated with a production and describes how attributes are computed

$$E \to E_1 + T \qquad\qquad E.code = E_1.code \parallel T.code \parallel \text{'+'}$$

The attribute *code* represents the postfix notation of the construct

∥ is the operator for string concatenation

*Grammar symbols represent language constructs. Nonterminal nodes and subtrees rooted at these nodes correspond to productions.

Annotated Parse Tree

- An annotated parse tree for infix expression `9-5+2`
- The attribute t represents postfix notation

$expr \rightarrow expr + term$

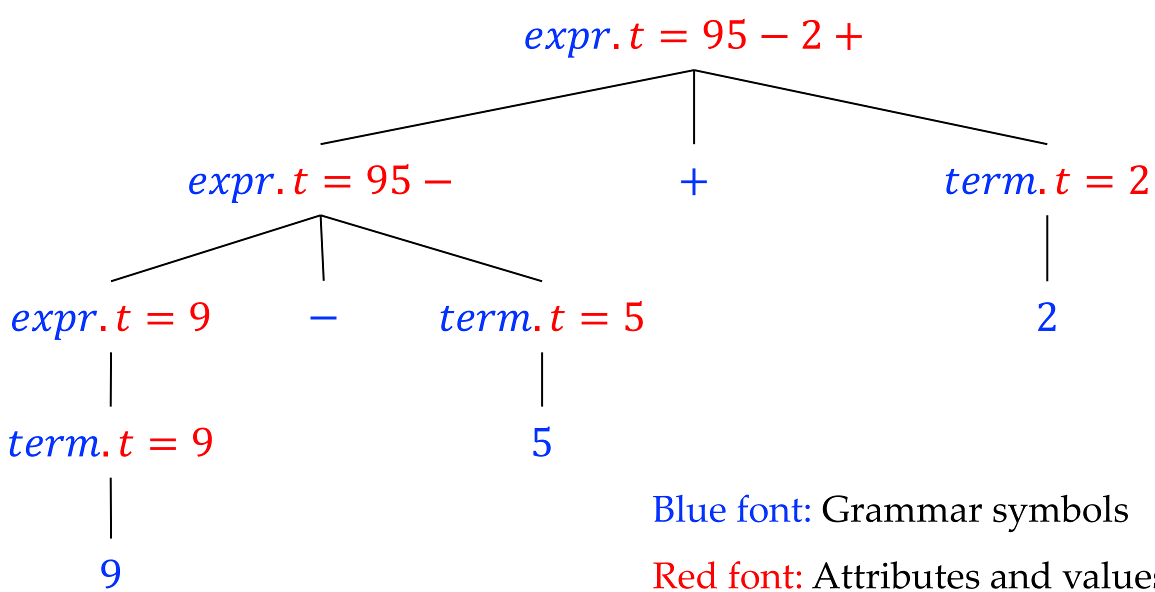
$expr \rightarrow expr - term$

$expr \rightarrow term$

$term \rightarrow 0$

…

$term \rightarrow 9$



Blue font: Grammar symbols

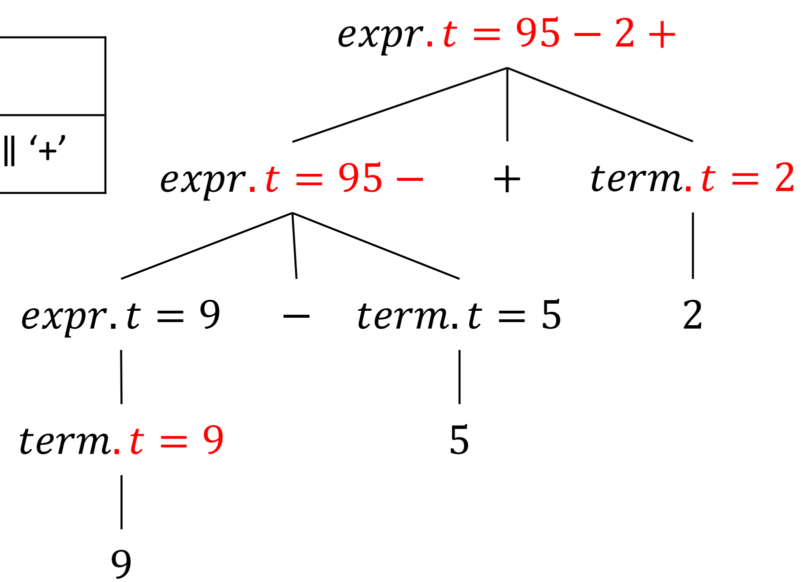
Red font: Attributes and values

Synthesized Attributes (合成属性)

- An attribute is said to be *synthesized* if its value at a parse-tree node N is only determined from attribute values at the children of N and at N itself (defined by a semantic rule associated with the production at N)

Production	$expr \rightarrow expr_1 + term$
Rule	$expr.t = expr_1.t \parallel term.t \parallel \text{‘+’}$

Production	$term \rightarrow 9$
Rule	$term.t = \text{‘9’}$

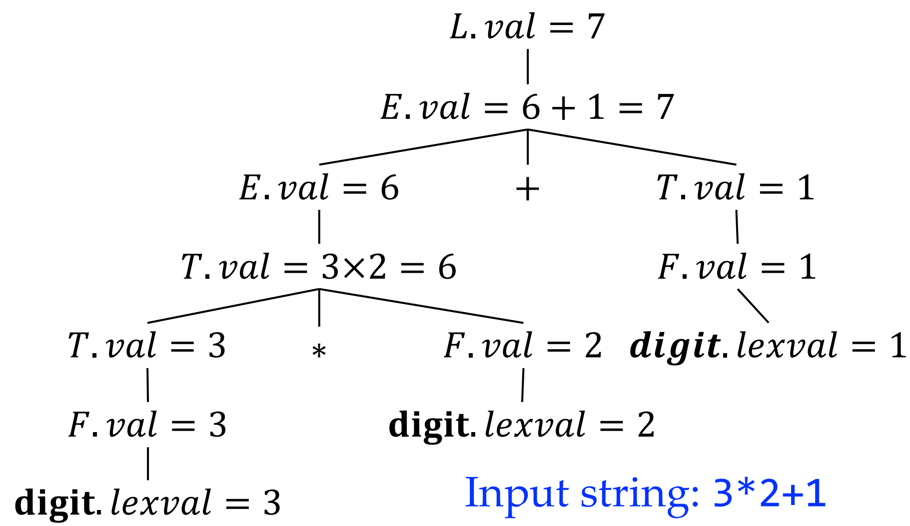


Terminals can also have synthesized attributes (lexical values), but there are no rules for computing the value of an attribute for a terminal.

A Complete Example of SDD

- The SDD below helps compute the value of an expression *L*
- SDD's do not specify the evaluation order of attributes on a parse tree
  - Any order that computes an attribute *a* after all other attributes that *a* depends on is fine
  - Synthesized attributes have a nice property that they can be evaluated during a single bottom-up traversal of a parse tree (it is often unnecessary to explicitly create the tree)

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



Input string: 3*2+1

Inherited Attributes (继承属性)

- ***Inherited attributes*** have their value at a parse-tree node determined from attribute values at the node itself, its parent, and its siblings in the parse tree

We already have synthesized attributes. When are inherited attributes useful???

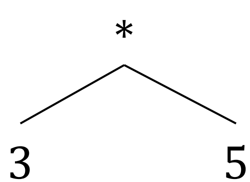
Top-Down Parse of 3*5

$T \rightarrow FT'$

$T' \rightarrow * \ FT'$

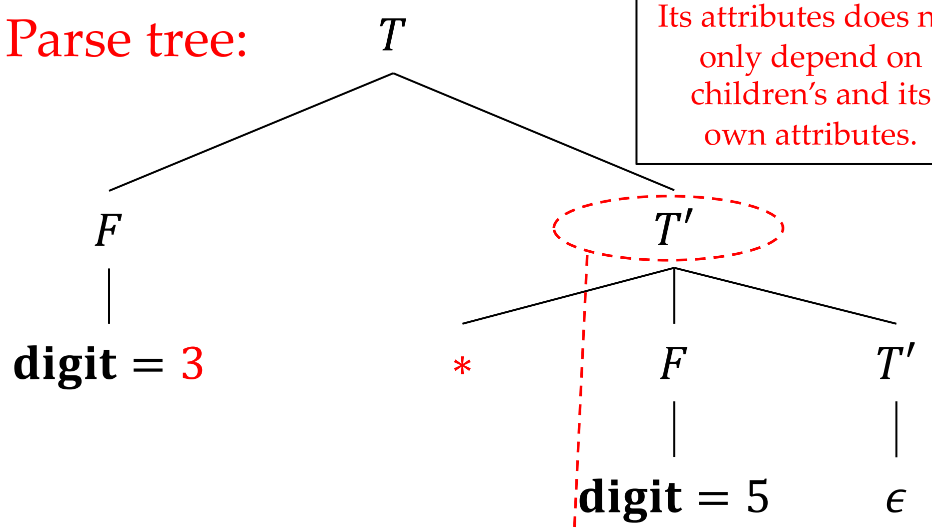
$T' \rightarrow \epsilon$

$F \rightarrow \textbf{digit}$



Abstract syntax tree

Parse tree:



Does not represent a valid subexpression;

Its attributes does not only depend on children's and its own attributes.

For some grammars, the structure of the parse tree does not **match** the abstract syntax of the code (3 and * are in different subtrees)

Not all non-terminals in a parse tree correspond to proper language constructs, e.g., T' above.

SDD with Inherited Attributes

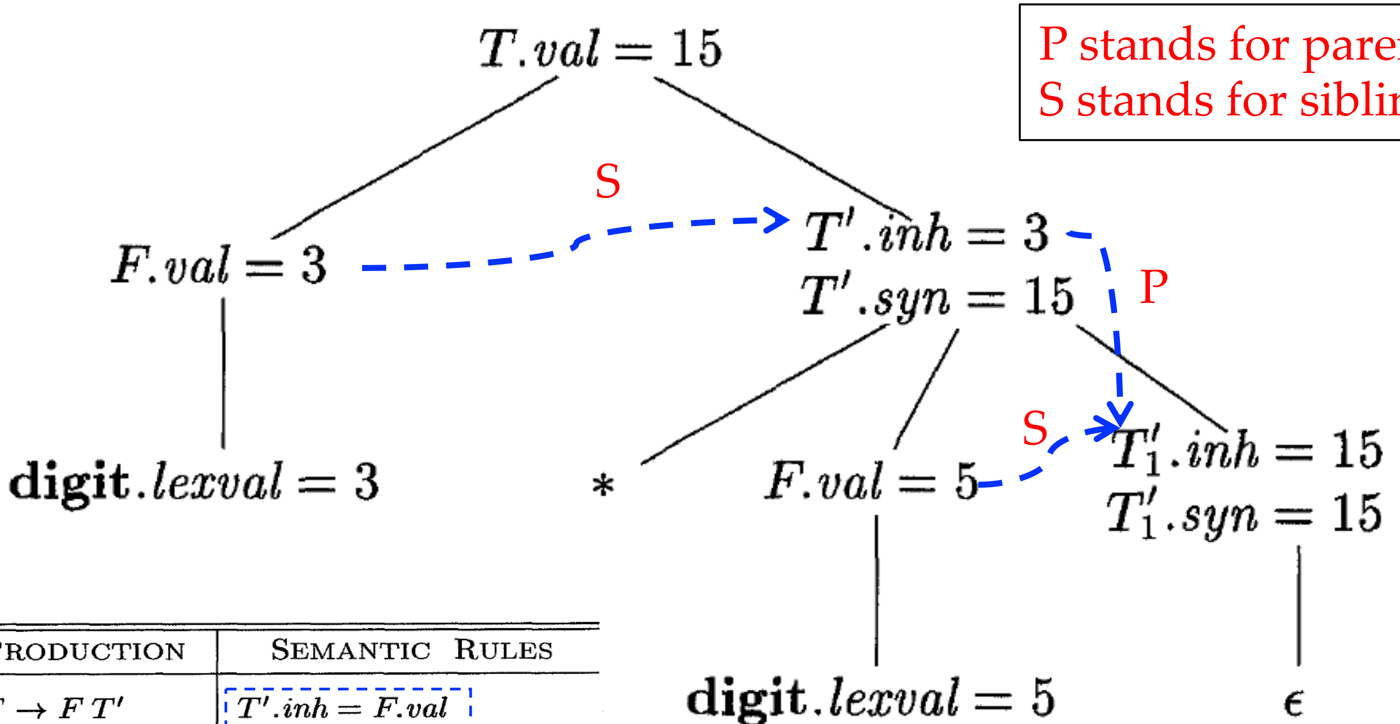
	Production	Semantic Rules
1)	$T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

The left operand of the operator $*$ is inherited by *T'* and kept for later computation when *T'* further gets replaced

The inherited attribute of *T'* is not defined by a rule associated with the production (2) or (3), whose head is *T'*

Annotated Parse Tree for 3*5

P stands for parent
S stands for sibling



	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$\boxed{T'.inh = F.val}$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T_1'$	$\boxed{T_1'.inh = T'.inh \times F.val}$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

value flow

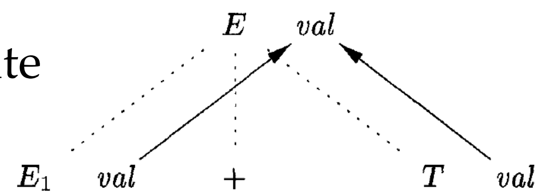
Outline

- Syntax-Directed Definitions
- Evaluation Orders for SDD's
- Applications of Syntax-Directed Translation (Lab)
- Syntax-Directed Translation Schemes
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)

Evaluation Orders for SDD's

- Given parse tree nodes N, M_1, M_2, …, M_k, if the attribute a of N is defined as $N.a = f(M_1.a_1, M_2.a_2, \dots, M_k.a_k)$, then in order to compute $N.a$, we must first compute $M_i.a_i$ $(1 \le i \le k)$

- *Dependency graphs* (依赖图) are a useful tool for determining evaluation orders
  - Depict the information flow among the attribute instances in a particular parse tree
  - Model the partial order among attribute instances (not every pair of elements has an order)

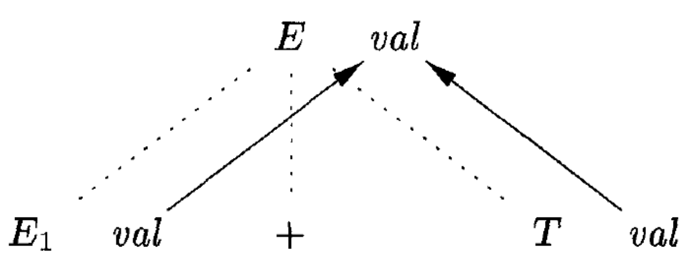


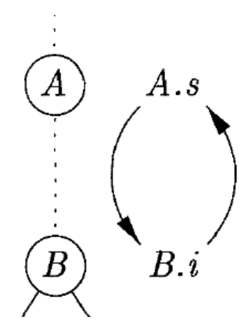
Dependency Graph

- An edge from one attribute instance (a_1) to another (a_2) means that the value of a_1 is needed to compute the value of a_2

- If there is any cycle in a dependency graph, we cannot find an order to compute the value of all attribute instances

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



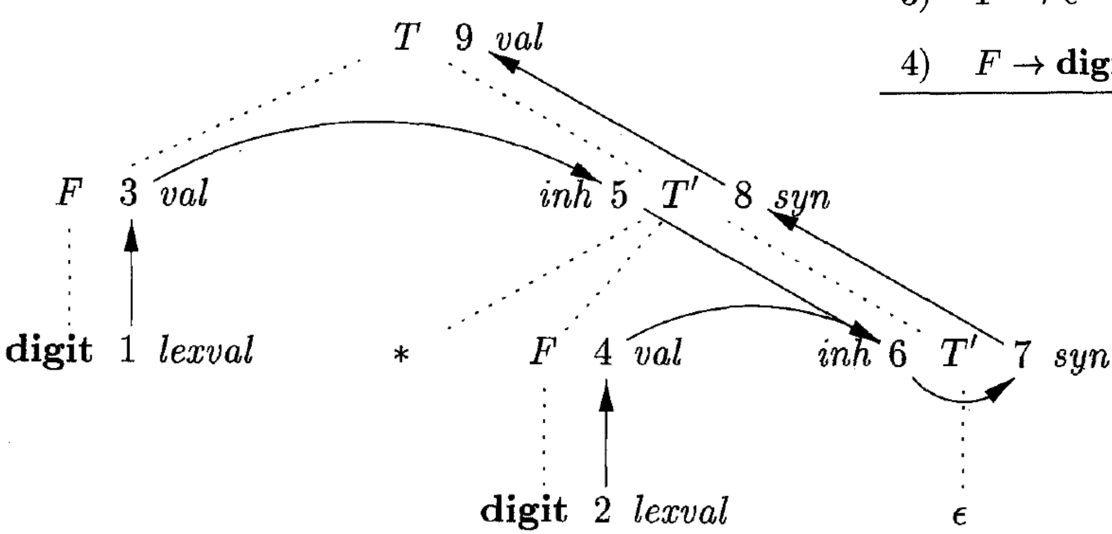


Dotted lines: parse tree edges **Solid lines:** depedency graph edges

Example: Parsing 3*2

Attribute values can be computed according to any *topological sort* (拓扑排序)* of the graph, e.g., 1, 2, 3, …, 9 in the example below

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



Dotted lines: parse tree edges

Solid lines: depedency graph edges

1–9: Evaluation order

* Topological sorting for a directed acyclic graph is a linear ordering of vertices such that for every directed edge $u \to v$, vertex u comes before v in the ordering. Topological sorting is not possible for graphs with cycles.

Ordering the Evaluation of Attributes

- Given an arbitrary SDD, it is hard to tell whether there exist any parse trees (annotated) whose dependency graphs have cycles (i.e., whether the SDD is computable)

- In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order*
  - ***S-attributed*** SDD's
  - ***L-attributed*** SDD's

* The dependency graphs for such SDDs are directed acyclic graphs

S-Attributed SDDs

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

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

Intuitively, there cannot be cycles in the dependency graph of any parse tree, since edges always go from children nodes to parent nodes, never the other way around.

S-Attributed SDDs Cont.

- When an SDD is S-attributed, we can evaluate its attributes in any bottom up order of the parse-tree nodes
  - e.g., postorder traversal (后序遍历) of the parse tree

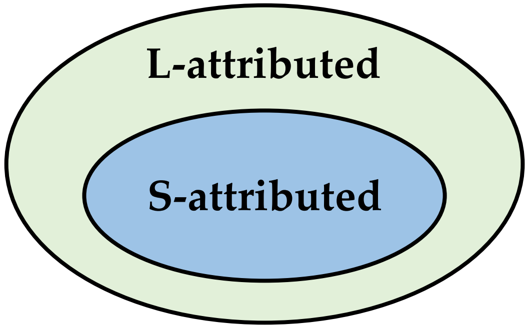
- So, S-attributed SDDs can be easily implemented during bottom-up parsing (the parsing process corresponds to a postorder traversal)

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

L-Attributed SDDs

- An SDD is *L-attributed* if for each production $A \rightarrow X_1 X_2 \dots X_n$, for each $j = 1 \dots n$, each inherited attribute of X_j depends on only:
  - the attributes of $X_1, \dots, X_{j-1}$ (either synthesized or inherited), or
  - the inherited attributes of A
- Or each attribute is synthesized

- Dependency-graph edges can go from left to right (on an annotated parse tree), but not right to left (hence the SDD is named "L-attributed")
- There cannot be cycles in the graph, as edges only go from left to right or from parents to children



L-Attributed SDD Example

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

Left sibling's attribute

- Parent's inherited attribute
- Left sibling's attribute

Synthesized attributes: val, syn, lexval. Inherited attributes: inh

Attribute Evaluation for L-Attributed SDD's

Input: A node n in a parse tree T

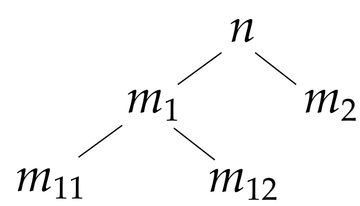
Output: An evaluation order for the attributes of the subtree rooted at n

Imagine the directions of the dependency graph edges, then it is easy to understand why DFS works here. Is the evaluation order a topological sort of parse tree nodes?

procedure depth_first(n)[*]
begin
    for every child m of n from left to right **do begin**
        evaluate the inherited attributes of m;
        depth_first(m); // here m's synthesized attributes will be evaluated
    end
    evaluate the synthesized attributes of n;
end

[*] The inherited attributes of x (non-root) are computed before calling depth_first(x), as indicated by the for body

Example




```
procedure depth_first(n)*
begin
    for every child m of n from left to right do begin
        evaluate the inherited attributes of m;
        depth_first(m);
    end
    evaluate the synthesized attributes of n;
end
```

depth_first(n)

1. Compute m_1's inherited attribute

depth_first(m_1)

2. Compute m_{11}'s inherited attribute

depth_first(m_{11})

3. Compute m_{11}'s synthesized attribute
4. Compute m_{12}'s inherited attribute

depth_first(m_{12})

5. Compute m_{12}'s synthesized attribute
6. Compute m_1's synthesized attribute
7. Compute m_2's inherited attribute

depth_first(m_2)

8. Compute m_2's synthesized attribute
9. Compute n's synthesized attribute

Attribute Evaluation for L-Attributed Definitions

Input: A node n in a parse tree T

Output: An evaluation order for the attributes of the subtree rooted at n

When evaluating the inherited attributes of a node, the attributes of nodes to its left have been evaluated

Guarantee

```
procedure depth_first(n)*
begin
    for every child m of n from left to right do begin
        evaluate the inherited attributes of m;
        depth_first(m); // here m's synthesized attributes will be evaluated
    end
    evaluate the synthesized attributes of n;
end
```

* The inherited attributes of x (non-root) are computed before calling depth_first(x), as indicated by the for body

Attribute Evaluation for L-Attributed Definitions

Input: A node n in a parse tree T

Output: An evaluation order for the attributes of the subtree rooted at n

When evaluating the inherited attributes of a node (m), the inherited attributes of its parent node (n) have been evaluated

Guarantee

```
procedure depth_first(n)*
begin
    for every child m of n from left to right do begin
        evaluate the inherited attributes of m;
        depth_first(m); // here m's synthesized attributes will be evaluated
    end
    evaluate the synthesized attributes of n;
end
```

* The inherited attributes of n are computed before calling depth_first(n), as indicated by the for body

Attribute Evaluation for L-Attributed Definitions

Input: A node n in a parse tree T

Output: An evaluation order for the attributes of the subtree rooted at n

```
procedure depth_first(n)*
begin
    for every child m of n from left to right do begin
        evaluate the inherited attributes of m;
        depth_first(m); // here m's synthesized attributes will be evaluated
    end
    evaluate the synthesized attributes of n;
end
```

Can be implemented in top-down parsing (will be introduced later)

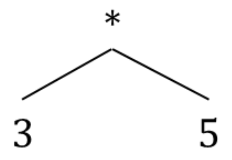
* The inherited attributes of n are computed before calling depth_first(n), as indicated by the for body

Outline

- Syntax-Directed Definitions
- Evaluation Orders for SDD's
- Applications of Syntax-Directed Translation (Lab)
  - Constructing Syntax tree
  - The Structure of a Type
- Syntax-Directed Translation Schemes
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)

Construction of Syntax Tree

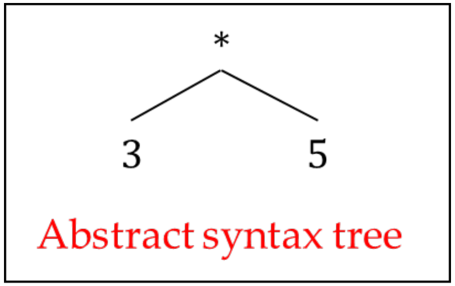
- Abstract syntax tree (or syntax tree for short) revisited:
  - Each interior node N represents a construct (corresponding to an operator)
  - The children of N represent the meaningful components of the construct represented by N (corresponding to operands)

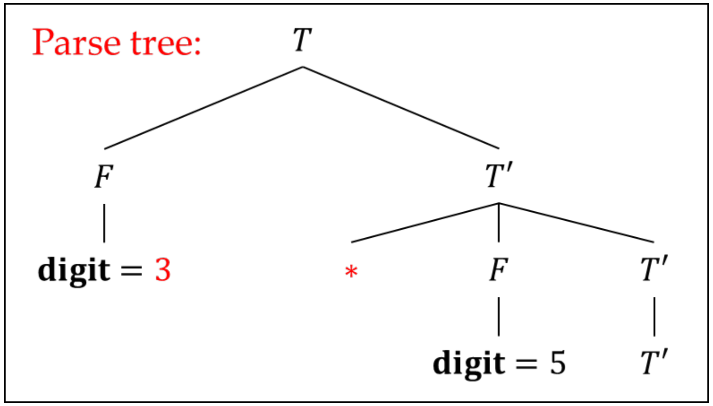


Abstract syntax tree

Construction of Syntax Tree

- Syntax tree vs. parse tree
  - In a syntax tree, interior nodes represent programming constructs, while in a parse tree, interior nodes represent nonterminals*
  - A parse tree is also called a *concrete syntax tree*, and the underlying grammar is called a *concrete syntax* for the language



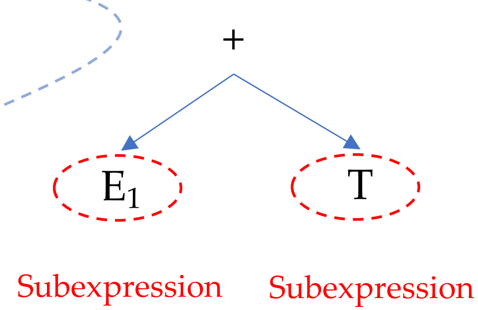


*Not all nonterminals represent programming constructs, e.g., those introduced to eliminate left recursions (*T'* in the earlier L-attributed SDD example)

Construction of Syntax Tree

- An S-attributed SDD for building syntax trees for simple expressions
  - Each node of the syntax tree is implemented as an **object** with a field *op*, representing the label of the node, and some additional fields
    - **Leaf node:** one additional field holding the lexical value
    - **Interior node:** # additional fields = # of children

	Production	Semantic Rules
1)	$E \to E_1 + T$	$\boxed{E.node = \textbf{new}\ Node('+', E_1.node, T.node)}$
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)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

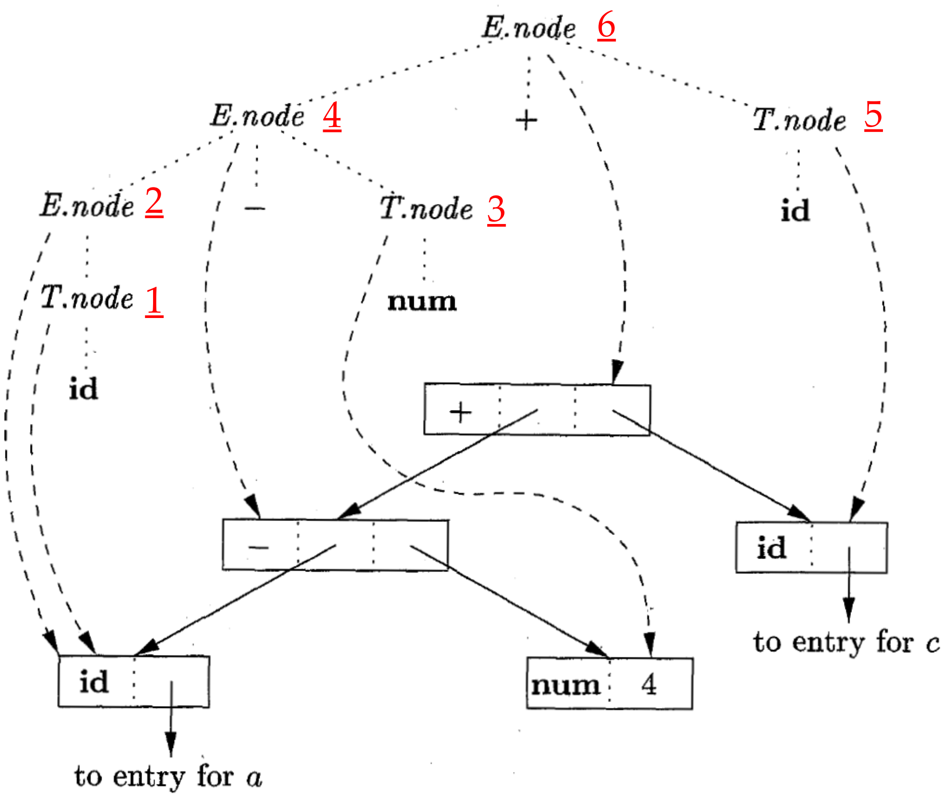


Construction of Syntax Tree

Input expression: $a - 4 + c$

Steps (object creations only; bottom-up evaluation):

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



Parse tree edge

Pointer to the node in syntax tree

Syntax tree edge

1- 5: Evaluation order of attributes

Outline

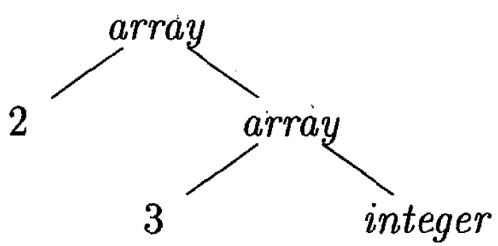
- Syntax-Directed Definitions
- Evaluation Orders for SDD's
- Applications of Syntax-Directed Translation (Lab)
- Syntax-Directed Translation Schemes
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)

- Constructing Syntax tree
- The Structure of a Type

Computing the Structure of a Type

```
int[2][3] a = ...;
```

What is the type of `a`?



elements   Element type

array($\boxed{2}$, $\boxed{\textit{array}(\ \boxed{3}, \boxed{\textit{integer}}\ )}$)

That is: array of 2 arrays of 3 integers

Computing the Structure of a Type

PRODUCTION
$T \rightarrow B\ C$
$B \rightarrow$ **int**
$B \rightarrow$ **float**
$C \rightarrow [\ \textbf{num}\ ]\ C_1$
$C \rightarrow \epsilon$

The grammar generates type specifiers:

- `int`
- `float`

Basic types

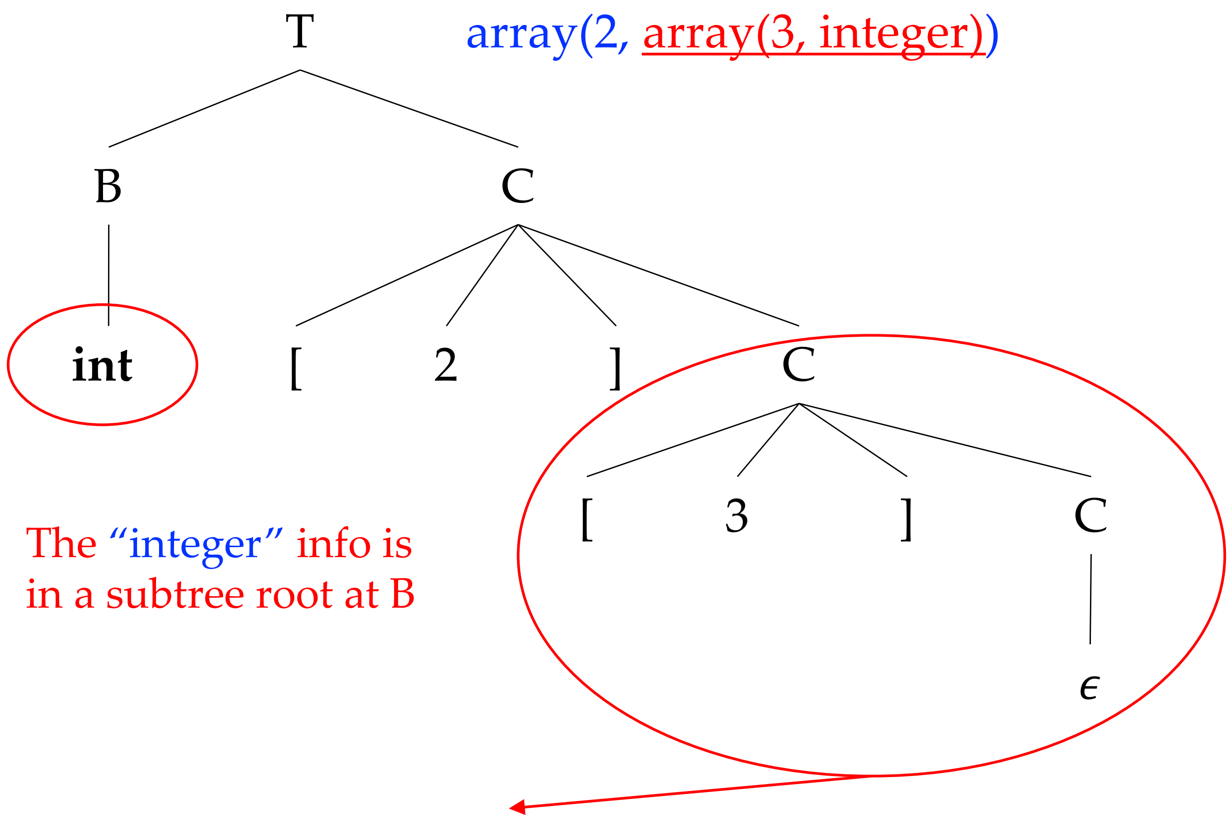
- `int[2]`
- `int[2][3]`
- `int[4][5][6]`
- `...`

Array types

Computing the Structure of a Type

- int[2][3]

PRODUCTION
$T \rightarrow B\ C$
$B \rightarrow$ **int**
$B \rightarrow$ **float**
$C \rightarrow [\ \textbf{num}\ ]\ C_1$
$C \rightarrow \epsilon$



How can we obtain the type expression array(3, integer) from this subtree?

Computing the Structure of a Type

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

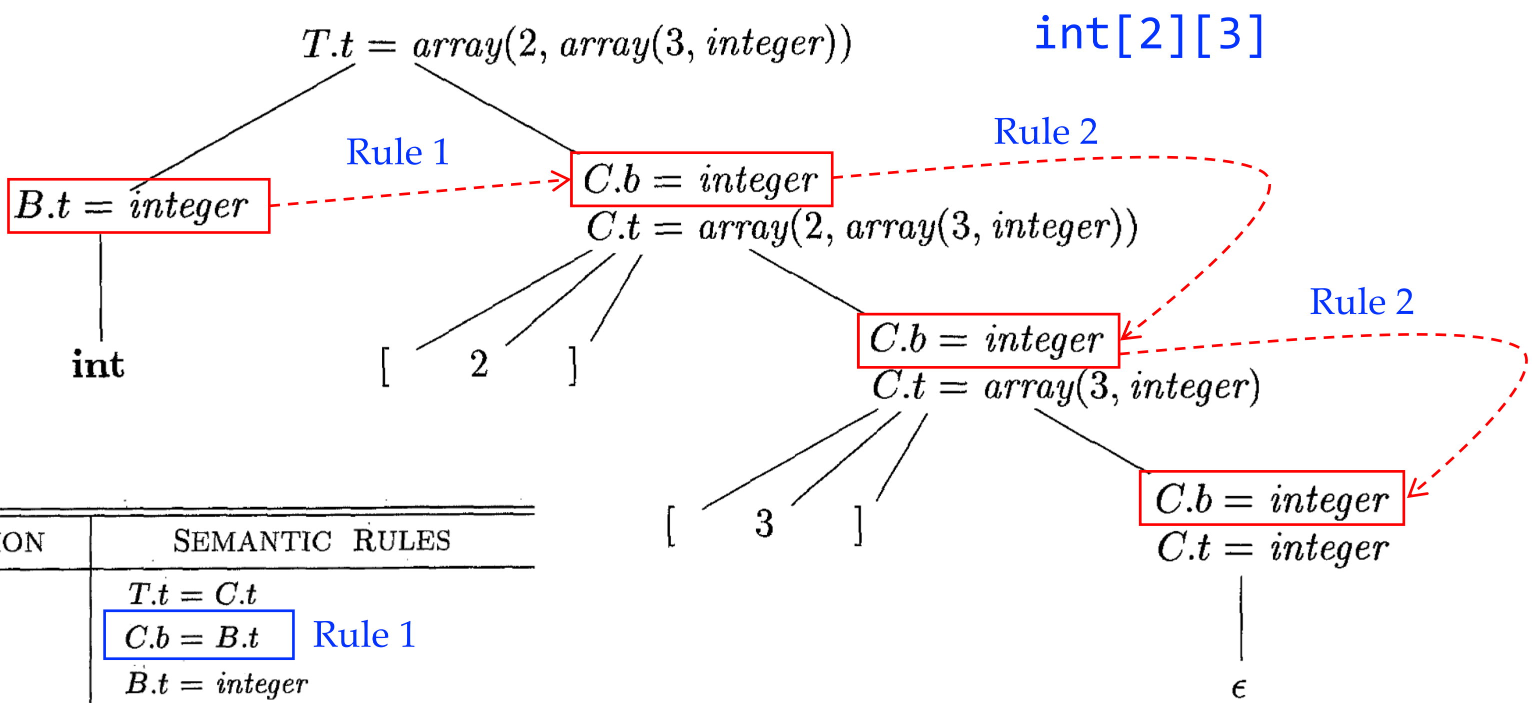
L-attributed SDD

Synthesized attribute *t* represents a type

Inherited attribute *b* passes the basic type down the parse tree

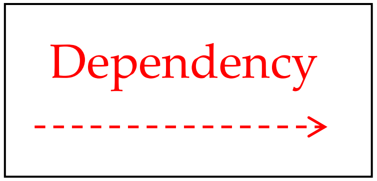
Computing the Structure of a Type

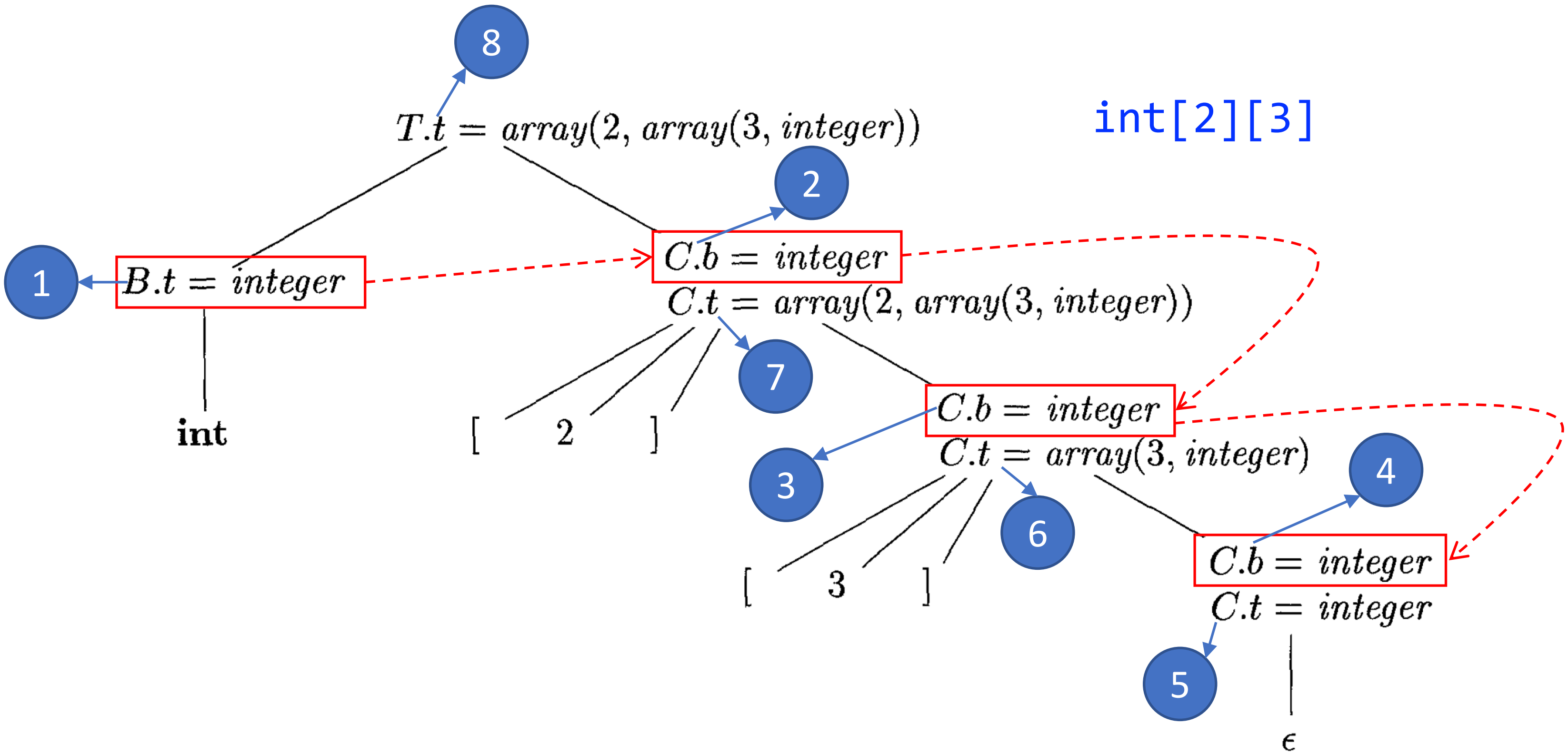
Dependency



PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $\boxed{C.b = B.t}$ Rule 1
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [$ **num** $]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $\boxed{C_1.b = C.b}$ Rule 2
$C \rightarrow \epsilon$	$C.t = C.b$

Computing the Structure of a Type





1 … 8 : evaluation order (according to the algorithm on #22)

Outline

- Syntax-Directed Definitions
- Evaluation Orders for SDD's
- Applications of Syntax-Directed Translation (Lab)
- Syntax-Directed Translation Schemes
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)

Syntax-Directed Translation Schemes

- SDD's tell us what to do (high-level specifications) in the translation, but not how to do
- Syntax-directed translation schemes (SDT's, 语法制导的翻译方案) specify more details on how to do the translation
- An SDT* is a context-free grammar with semantic actions (program fragments) embedded within production bodies
  - Differ from the semantic rules in SDD's
  - Semantic actions can appear anywhere within a production body

*In this course, we are only interested in SDT's for computing L-attributed SDD's.

An Example SDT (1)

- The SDT below implements a simple calculator

SDT

L	$\rightarrow$	E **n**	$\{$ 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	$\rightarrow$	**digit**	$\{ F.val =$ **digit**$.lexval; \}$

Semantic actions:

Real code in {}

SDD

$L \rightarrow E$ **n**	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

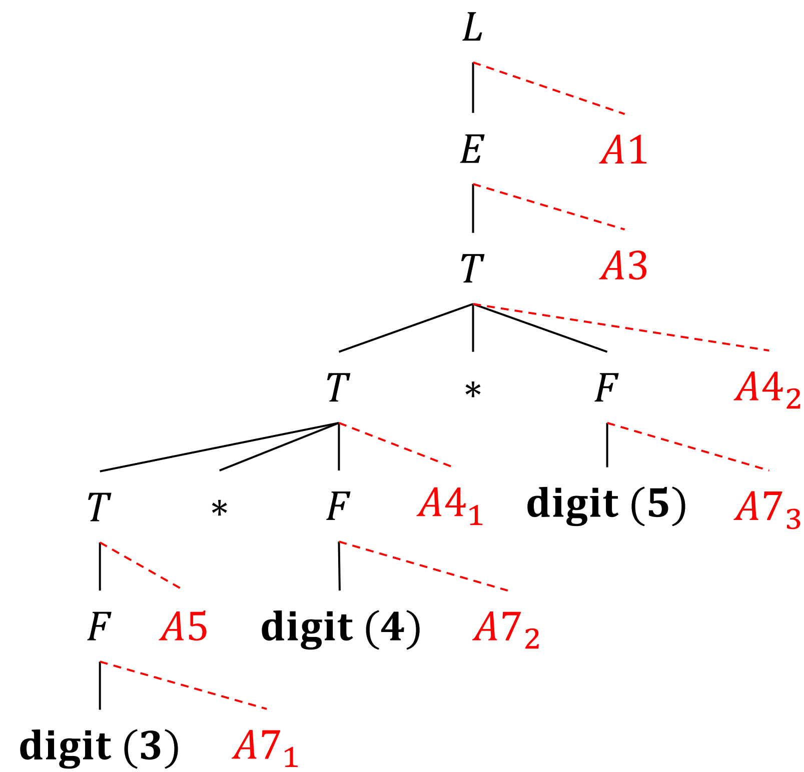
Semantic rules:

Mathematical definitions

An Example SDT (2)

- Parse and calculate 3*4*5

$$
\begin{array}{llll}
L & \rightarrow & E \; \mathbf{n} & \{ \; \text{print}(E.val); \; \} \; \text{A1} \\
E & \rightarrow & E_1 + T & \{ \; E.val = E_1.val + T.val; \; \} \; \text{A2} \\
E & \rightarrow & T & \{ \; E.val = T.val; \; \} \; \text{A3} \\
T & \rightarrow & T_1 * F & \{ \; T.val = T_1.val \times F.val; \; \} \; \text{A4} \\
T & \rightarrow & F & \{ \; T.val = F.val; \; \} \; \text{A5} \\
F & \rightarrow & (\; E \;) & \{ \; F.val = E.val; \; \} \; \text{A6} \\
F & \rightarrow & \mathbf{digit} & \{ \; F.val = \mathbf{digit}.lexval; \; \} \; \text{A7}
\end{array}
$$



Order of actions:

$A7_1$, $A5$, $A7_2$, $A4_1$, $A7_3$, $A4_2$, $A3$, $A1$

All SDT's (for computing L-attributed SDD's) can be implemented by: 1) first building the parse tree, 2) treating semantic actions as "virtual" parse-tree nodes, and 3) performing preorder traversal

SDT's With Actions Inside Productions

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

- The action a should be done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal)
  - If the parse is bottom-up, we perform the action a as soon as X appears on the top of the parsing stack
  - If the parse is top-down, we perform the action a before attempting to expand Y (if Y is a nonterminal) or check for Y on the input (if Y is a terminal)

SDT's Implementable During Parsing

- In practice, SDT's are often implemented during parsing, without first building a parse tree
- Not all SDT's can be implemented during parsing*
  - Even if the underlying grammar is parsable by a method (e.g., LL, LR), after introducing semantic actions, the parsing method may become inapplicable
- Determine if an SDT can be implemented during parsing
  - Introduce distinct *marker nonterminals* to replace each embedded action; each marker M has only one production $M \rightarrow \epsilon$
  - If the grammar with marker nonterminals can be parsed by a given method, then the SDT can be implemented during parsing

*Note that all SDT's for computing L-attributed SDD's can be implemented after building the parse tree, as earlier mentioned

A Problematic SDT

- This SDT translates infix expression to prefix expressions

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

$$
\begin{array}{ll}
L \rightarrow E & \\
E \rightarrow M_1 E + T & M_1 \rightarrow \epsilon \\
E \rightarrow T & \\
T \rightarrow M_2 T * F & M_2 \rightarrow \epsilon \\
T \rightarrow F & \\
F \rightarrow (E) & \\
F \rightarrow \mathbf{digit}\ M_3 & M_3 \rightarrow \epsilon
\end{array}
$$

It is impossible to build parsing tables without conflicts using top-down or bottom-up parsing methods. This SDT cannot be implemented during parsing. It can be implemented after parsing according to earlier mentioned steps.

Outline

- Syntax-Directed Definitions
- Evaluation Orders for SDD's
- Applications of Syntax-Directed Translation (Lab)
- Syntax-Directed Translation Schemes
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)

Uses of SDT's

- We can use SDT's to implement two important classes of SDD's:
  - The underlying grammar is LR, and the SDD is S-attributed
  - The underlying grammar is LL, and the SDD is L-attributed

Postfix Translation Schemes

- If the grammar of an SDD is LR, and the SDD is S-attributed, then we can construct a *postfix SDT* (后缀SDT) to implement the SDD in bottom-up parsing
  - Semantic actions always appear at the end of productions (hence "postfix")

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

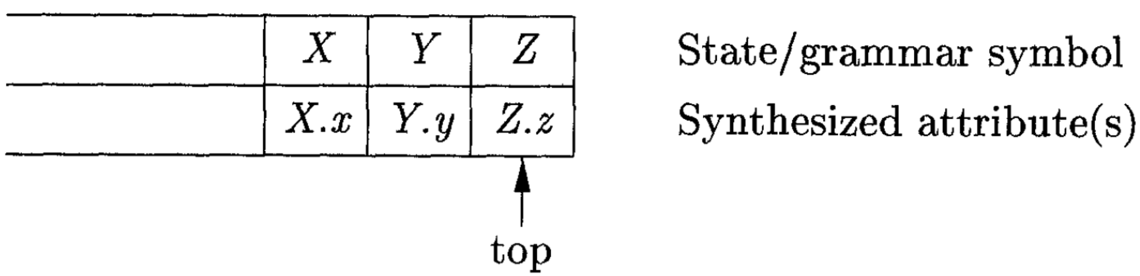
			SDT
L	$\to$	E **n**	$\{$ print$(E.val)$; $\}$
E	$\to$	$E_1 + T$	$\{ E.val = E_1.val + T.val; \}$
E	$\to$	T	$\{ E.val = T.val; \}$
T	$\to$	$T_1 * F$	$\{ T.val = T_1.val \times F.val; \}$
T	$\to$	F	$\{ T.val = F.val; \}$
F	$\to$	(E)	$\{ F.val = E.val; \}$
F	$\to$	**digit**	$\{ F.val =$ **digit**$.lexval;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

This is possible because in bottom-up parsing, before reducing to a production head, the grammar symbols in the production body have been visited and their synthesized attributes have been computed (both non-terminals and terminals).

Parser-Stack Implementation of Postfix SDT's

- Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur

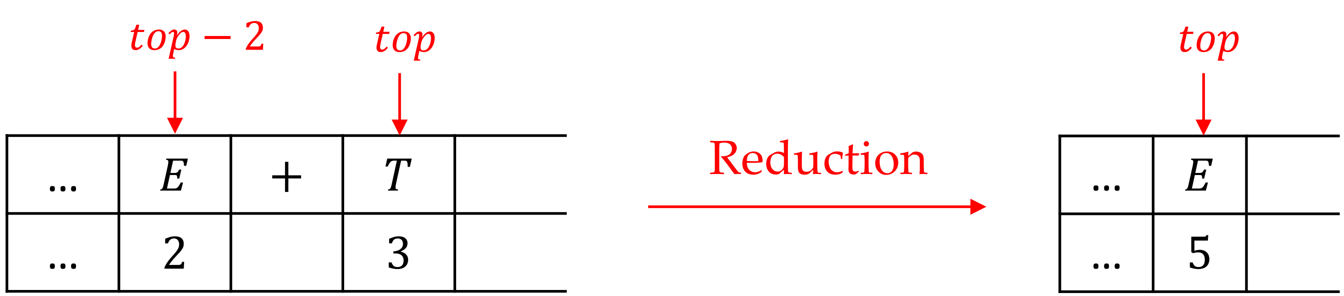
- The synthesized attributes can be placed along with the grammar symbols on the stack



If we do reduction using $A \to XYZ$, then the attributes of A can be calculated based on the attributes of X, Y, and Z, which are already on the stack.

The Calculator Example

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



Uses of SDT's

- We can use SDT's to implement two important classes of SDD's:
  - The underlying grammar is LR, and the SDD is S-attributed
  - The underlying grammar is LL, and the SDD is L-attributed

SDT's for L-Attributed SDD's

- L-attributed SDD's can be implemented during top-down parsing, if the underlying grammar is LL

- The way of turning an L-attributed SDD into an SDT is to place semantic actions at appropriate positions in the concerned production $A \rightarrow X_1 X_2 \dots X_n$
  - Embed the action that computes the inherited attributes for a nonterminal X_i immediately before X_i in the production body
  - Place the actions that compute a synthesized attribute for the production head at the end of the production body

An L-Attributed SDD

- The SDD generates labels for the `while` loop

L1

```
while (condition) {
    doSomething();
}
```

L2

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

Inherited attributes: $S.next$, $C.true$, $C.false$

Synthesized attribute: $S.code$

* There will be jump instructions with the labels as targets in $C.code$ and $S_1.code$.

Turning into an SDT

- Semantic actions:
  a) $L1 = new(); L2 = new();$
  b) $C.false = S.next;\ C.true = L2;$
  c) $S_1.next = L1;$
  d) $S.code = \cdots;$
- According to the rules of action placement:
  - b) should be placed before C, c) should be placed before S_1, and d) should be placed at the end of the production body
  - a) can be placed at the very beginning; there is no constraint

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

Outline

- Syntax-Directed Definitions
- Evaluation Orders for SDD's
- Applications of Syntax-Directed Translation (Lab)
- Syntax-Directed Translation Schemes
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)

Translation During Recursive-Descent Parsing

- Many translation applications can be addressed using L-attributed SDD’s. It is possible to extend a recursive-descent parser to implement L-attributed SDD’s.
  - A recursive-decent parser has a function *A* for each nonterminal *A*

```
void A() {
        Choose an A-production, A → X1 X2 ··· Xk;
        for ( i = 1 to k ) {
                if ( Xi is a nonterminal )
                        call procedure Xi();
                else if ( Xi equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

Translation During Recursive-Descent Parsing

- Extend a recursive-descent parser to implement L-attributed SDD's as follows:
  - A recursive-decent parser has a function *A* for each nonterminal *A*
  - Use the arguments of function *A* to **pass** *A*'s inherited attributes so that children nodes on the parse tree can use the attributes
  - **Return** the synthesized attributes of *A* when the function *A* completes so that parent node on the parse three can use the attributes

- With the above extension, in the body of the function *A*, we need to both parse and handle attributes

The While-Loop Example

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

Save attributes in local variables

Pass inherited attributes
(the label of the statement after while)

```
string S(label next) {
    string Scode, Ccode; /* local variables holding code fragments */
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();     C.false    C.true
        L2 = new();
        Ccode = C(next, L2);
        check ')' is next on the input, and advance;
        Scode = S(L1);  <- S_1.next (the label of the condition evaluating statement)
        return("label" || L1 || Ccode || "label" || L2 || Scode);
    }
    else /* other statement types */
}
```

Pass inherited attributes when further handling other nonterminals

Compute synthesized attributes and return

We mainly put code that handles attributes here, the code is not complete.

Reading Tasks

- Chapter 5 of the dragon book
  - 5.1 Syntax-Directed Definitions
  - 5.2 Evaluation Orders for SDD’s (5.2.1-5.2.4)
  - 5.3 Applications of Syntax-Directed Translation
  - 5.4 Syntax-Directed Translation Schemes
  - 5.5 Implementing L-Attributed SDD’s (5.5.1)