编译原理与设计

北京理工大学 计算机学院

语法制导翻译

- Formalisms for specifying translations for programming language constructs guided by context-free grammar
  - We associate **Attributes** to the grammar symbols representing the language constructs
  - Values for attributes are computed by **Semantic Rules** associated with grammar production
- Evaluation of semantic rules may
  - Insert information into the Symbol Table
  - Generate Code
  - Perform Semantic Check
  - Issue error message
  - …

语法制导翻译

- Two notations for attaching semantic rules
  - **Syntax Directed Definitions**. High-level specification hiding many implementation details (also called Attribute Grammars)
  - **Translation Schemes**. More implementation oriented: Indicate the order in which semantic rules are to be evaluated

SDD

$E \to E + T$	$E.code = E.code \| T.code \| '+'$
$E \to E - T$	$E.code = E.code \| T.code \| '-'$
$E \to T$	$E.code = T.code$
$T \to 0$	$T.code = '0'$
$T \to 1$	$T.code = '1'$
$\ldots$	
$T \to 9$	$T.code = '9'$

SDTScheme

$E \to E + T$	$\{print'+'\}$
$E \to E - T$	$\{print'-'\}$
$E \to T$	
$T \to 0$	$\{print'0'\}$
$T \to 1$	$\{print'1'\}$
$\ldots$	
$T \to 9$	$\{print'9'\}$

infix to postfix translation

语法制导翻译

- Syntax directed definition (SDD)
  - Specifies the values of attributes by associating semantic rules with the productions
  - a CFG along with attributes and rules, an attribute is associated with grammar symbols (attribute grammar), rules are associated with productions
  - is easier to read, easy for specification
- Syntax directed translation Scheme (SDT)
  - embeds program fragments (also called semantic actions) within production bodies
  - can be more efficient, easy for implementation

语法制导定义

- A generalization of context-free grammar
- Grammar symbols have an associated set of Attributes
- Productions are associated with Semantic Rules for computing the values of attribute
- generates Annotated Parse-Trees where each node is of type record with a field for each attributes

语法制导定义

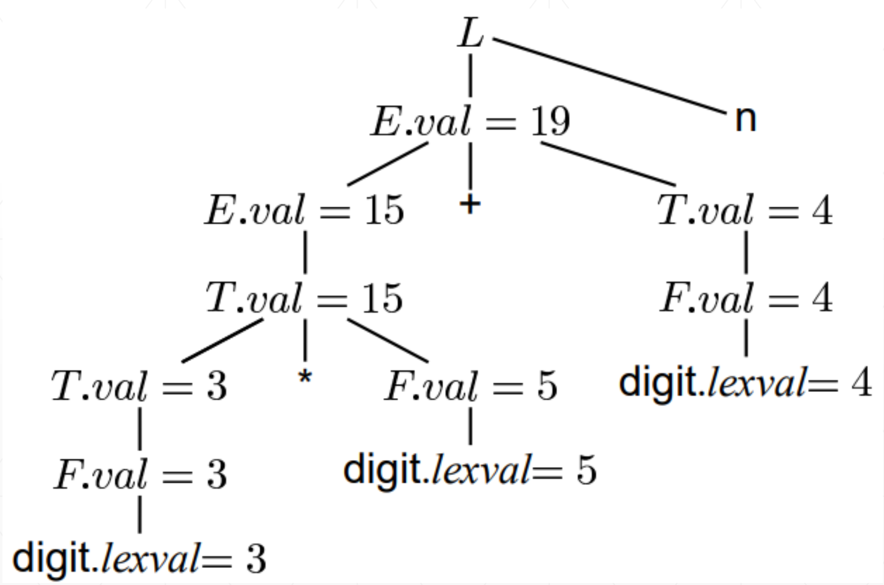
- **Synthesized Attributes**. Computed from the values of the attributes of the children node
- **Inherited Attributes**. Computed from the values of the attributes of both the siblings and the parent node

PRODUCTION	SEMANTIC RULE
$L \to E\mathrm{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathrm{digit}$	$F.val := \mathrm{digit}.lexval$

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

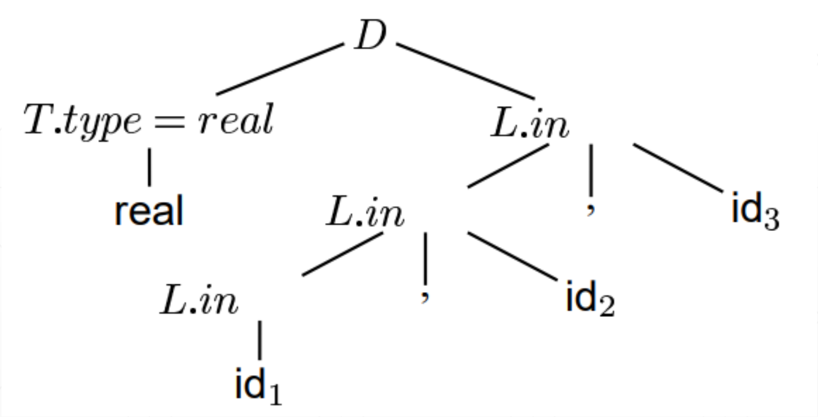
语法制导定义

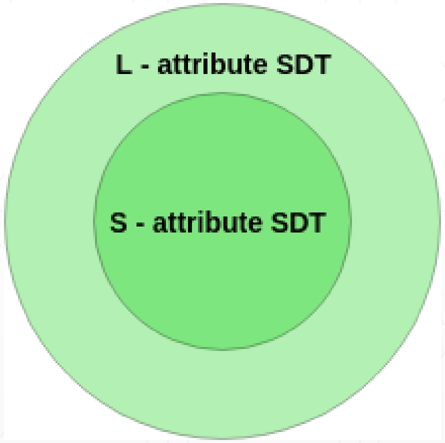
- **S-Attributed Definitions**. A Syntax Directed Definition that uses only synthesized attributes
  - evaluated by a bottom-up, or post order, traversal of the parse-tree



语法制导定义

- **L-Attributed Definitions**. uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only
  - Inherited attributes can be evaluated by pre order





语法制导定义实现/模式

- Attributed Evaluation

  - Unrestricted definition
    - No restrictions on attribute dependency
    - Perform a topological sort on the attribute dependency graph and evaluate in topological order
    - Expensive to evaluate
  - S-attributed definition
    - Synthesized attributes only
    - Attributes may be evaluated bottom-up
    - Evaluated very efficiently
  - L-attributed definition
    - Permits both synthesized and some inherited attributes
      Inherited attributes restricted to left siblings
    - Attributes may be evaluated depth-first, left to right

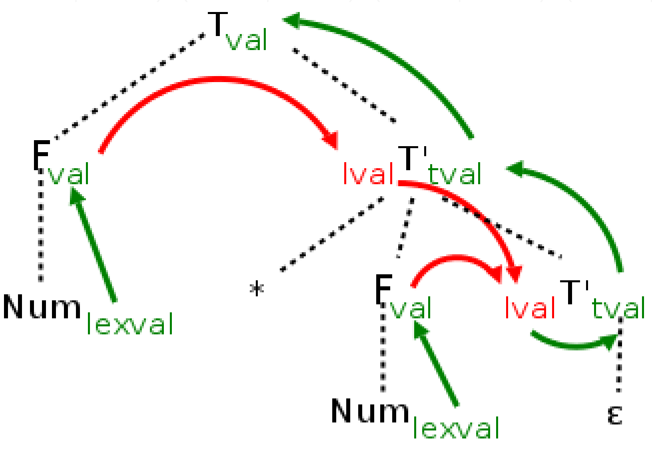
语法制导定义实现/模式

- Dependency Graphs
  - Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes
  - Dependency Graphs are the most general procedure to evaluate syntax directed translations with both synthesized and inherited attribute
  - shows the interdependencies among the attributes of the various nodes of a parse tree
    - There is a node for each attribute
    - If attribute **b** depends on an attribute **c** there is a link from the node for **c** to the node for ***b* (*b*←*c*)**

语法制导定义实现/模式

- **Dependency Graph**
  - Attribute at the head of an arrow depends on the on the one at the tail
  - Must evaluate the head attribute **after** evaluating the tail attribute

Production	Semantic Rules	Type
T → F T'	T'.lval = F.val	Inherited
	T.val = T'.tval	Synthesized
T' → * F T_1'	T'_1.lval = T'.lval * F.val	Inherited
	T'.tval = T'_1.tval	Synthesized
T' → ε	T'.tval = T'.lval	Synthesized
F → num	F.val = num.lexval	Synthesized



语法制导定义实现/模式

- **Dependency Graph** evaluation order
  - The evaluation order of semantic rules depends from a Topological Sort derived from the dependency graph
  - Any topological sort of a dependency graph gives a valid order to evaluate the semantic rules
- Topological sort
  - Choose a node having no incoming edges
  - Delete the node and all outgoing edges.
  - Repeat

语法制导定义实现/模式

- Evaluation of S-Attributed Definitions
  - The parser keeps the values of the synthesized attributes in its stack.
  - Whenever a reduction $A \rightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.
  - Thus, a translator for an S-Attributed Definition can be implemented by extending the stack of an LR-Parser.

- Synthesized attributes are computed just before each reduction:
  - Before the reduction $A \rightarrow XYZ$ is made, the attribute for A is computed: $A.a := f(val[top], val[top-1], val[top-2])$.

state	*val*
Z	$Z.x$
Y	$Y.x$
X	$X.x$
$\ldots$	$\ldots$

语法制导定义实现/模式

- Evaluation of S-Attributed Definitions

- **Example.** Consider the S-attributed definitions for the arithmetic expressions. To evaluate attributes the parser executes the following code

PRODUCTION	CODE
$L \to E\mathsf{n}$	$print(val[top-1])$
$E \to E_1 + T$	$val[ntop] := val[top] + val[top-2]$
$E \to T$	
$T \to T_1 * F$	$val[ntop] := val[top] * val[top-2]$
$T \to F$	
$F \to (E)$	$val[ntop] := val[top-1]$
$F \to \mathsf{digit}$	

- The variable *ntop* is set to the *new top of the stack*. After a reduction is done *top* is set to *ntop*.
  - When a reduction $A \to \alpha$ is done with $\mid \alpha \mid = r$, then $ntop = top - r + 1$.

语法制导定义实现/模式

- Evaluation of L-Attributed Definitions

- **L-Attributed Definitions** contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.
- **Definition.** A syntax directed definition is *L-Attributed* if each *inherited attribute* of X_j in a production $A \to X_1 \ldots X_j \ldots X_n$, depends only on:
  1. The attributes of the symbols to the **left** (this is what L in *L-Attributed* stands for) of X_j, i.e., $X_1 X_2 \ldots X_{j-1}$, and
  2. The inherited attributes of A.
- **Note.** An S-Attributed definition is also L-Attributed since the restrictions only apply to inherited attributes.

语法制导定义实现/模式

- Evaluation of L-Attributed Definitions

  - L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.
  - The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

  Algorithm L-Eval(n: Node). *Input:* Parse-Tree node from an L-Attribute Definition. *Output:* Attribute evaluation.

```
Begin
    For each child m of n, from left-to-right Do Begin;
        evaluate inherited attributes of m;
        L-Eval(m)
    End;
    evaluate synthesized attributes of n
End.
```

语法制导定义实现/模式

- Evaluation of L-Attributed Definitions

Grammar rule:

$$while\_stmt \rightarrow \texttt{while} \ \texttt{(} \ expr \ \texttt{)} \ stmt$$

L-attributed grammar rule:

while_stmt → `while (`
```
        { fprintf(fout, "__while_top%u:\n", label++); }
        expr )
        { fprintf(fout, "lw $a1 %s\n", temp);
          fprintf(fout, "beqz $a1 __end_while%u\n", label); }
        stmt
        { fprintf(fout, "j __while_top%u\n", label);
          fprintf(fout, "__end_while%u:\n", label); }
```

AST 构建

- AST = Abstract Syntax Tree

```
Parse Tree
==========

       exp
        |
      term
       /|\
   term * factor
    |     /|\
    |    / | \
factor ( exp )
  |       /|\
  3    exp + term
        |      |
      term   factor
        |      |
      factor   2
        |
        4
```

```
Abstract Syntax Tree
====================

  *
 / \
3   +
   / \
  4   2
```

AST 构建

- AST = Abstract Syntax Tree

```
Input
=====

{
   x = 0;
   while (x<10) {
      x = x+1;
   }
   y = x*2;
}
```

```
                Parse Tree
                ==========

          ____ methodBody _________
         /       /          \        \
        {   declList          stmtList }
               |            /          \
            epsilon    stmtList          stmt___
                     /       \          /  |  \   \
               stmtList     stmt      ID  = exp  ;
              /          |        \   (y)  /  |  \
        stmtList        stmt      ...   exp  *  term
           |           / | | \           |       |
        epsilon      ID  = exp ;        term   factor
                     (x)    |             |       |
                        INTLITERAL     factor    INT
                           (0)            |      (2)
                                          ID
                                          (x)
```

```
          AST
          ===

          methodBody
         /          \
 declList            stmtList
                   /    |    \
           assign  while   assign
           /    \    ...   /      \
         ID     INT      ID        *
         (x)    (0)      (y)      / \
                                 ID  INT
                                 (x) (2)
```

AST 构建

- Use semantic actions to build the AST
  - **LL parsing: extend procedures for nonterminals**
  - **Example:**

$$\boxed{\begin{array}{l} S \rightarrow ES' \\ S' \rightarrow \varepsilon \mid + S \\ E \rightarrow num \mid (S) \end{array}}$$

```
void parse_S() {
  switch (token) {
    case num: case '(':
      parse_E();
      parse_S'();
      return;
    default:
      throw new ParseError();
  }
}
```

```
Expr parse_S() {
  switch (token) {
    case num: case '(':
      Expr left = parse_E();
      Expr right = parse_S'();
      if (right == null) return left;
      else return new Add(left, right);
    default: throw new ParseError();
  }
}
```

AST 构建

- LR parsing
  - Need to add code for explicit AST construction
- AST construction mechanism for LR Parsing
  - With each symbol X on stack, also store AST sub-tree for X on stack
  - When parser performs reduce operation for $A \rightarrow \beta$, create AST subtree for A from AST fragments on stack for β, pop $|\beta|$ subtrees from stack, push subtree for β.

AST 构建

- LR parsing

$$\boxed{E \rightarrow \text{num} \mid (E) \mid E+E \mid E*E}$$

Parser stack stores value of each symbol

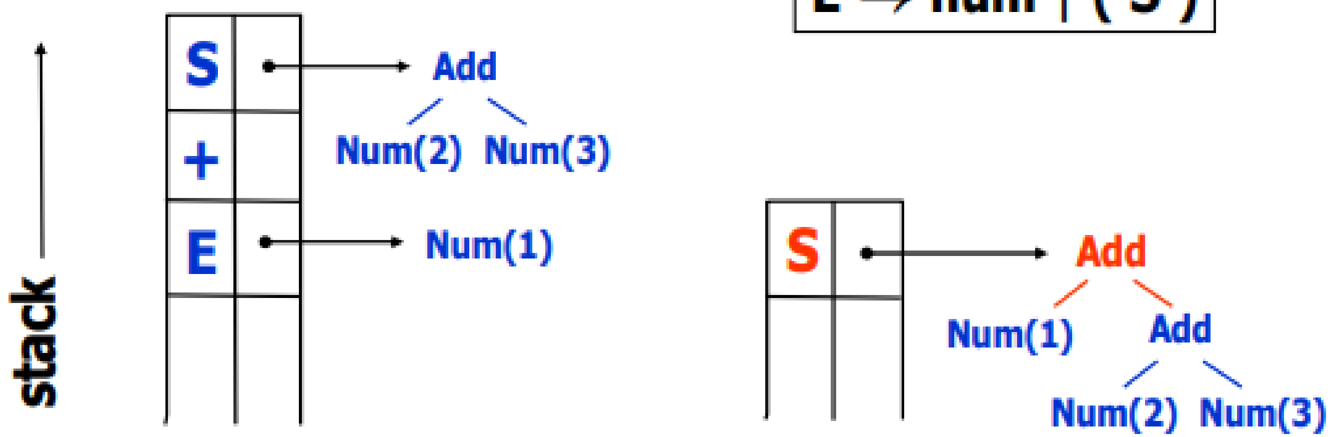
		(1+2)*3	
(1		+2)*3	
(E	Num(1)	+2)*3	RESULT=new Num(1)
(E+2		)*3	
(E+E	Num(2)	)*3	RESULT=new Num(2)
(E	Add(,)	)*3	RESULT=new Add(e1,e2)
(E)		*3	
E		*3	RESULT=e

AST 构建

- **Example**

$S \rightarrow E+S \mid S$

$E \rightarrow num \mid (\ S\ )$



Before reduction

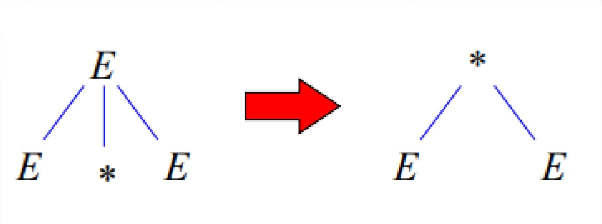
$S \rightarrow E+S$

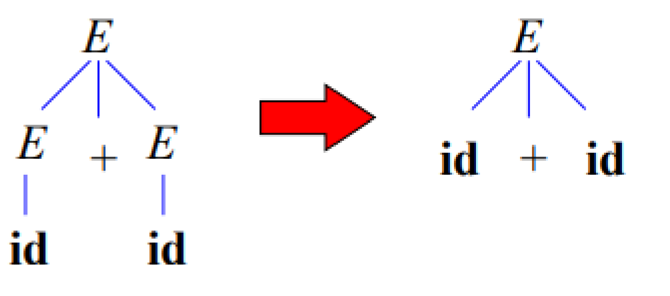
After reduction

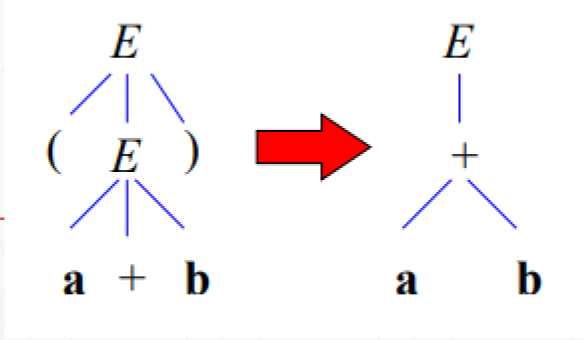
$S \rightarrow E+S$

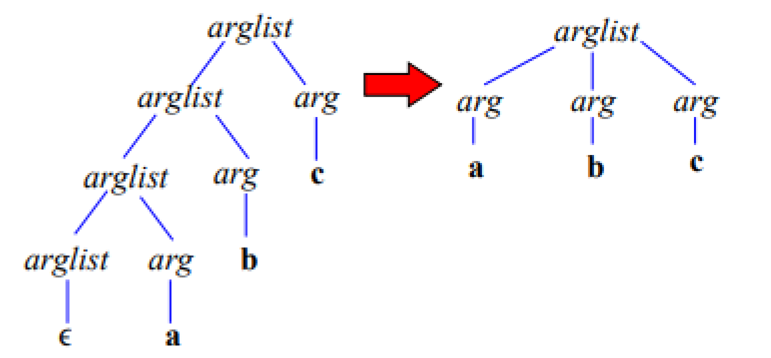
AST 构建

- Convert a concrete syntax tree to abstract syntax tree
  - Operators are promoted from leaves to internal nodes
  - Chains of single productions are collapsed
  - Syntactic details like parentheses, semi-colons, and commas are omitted
  - Subtree lists are flattened





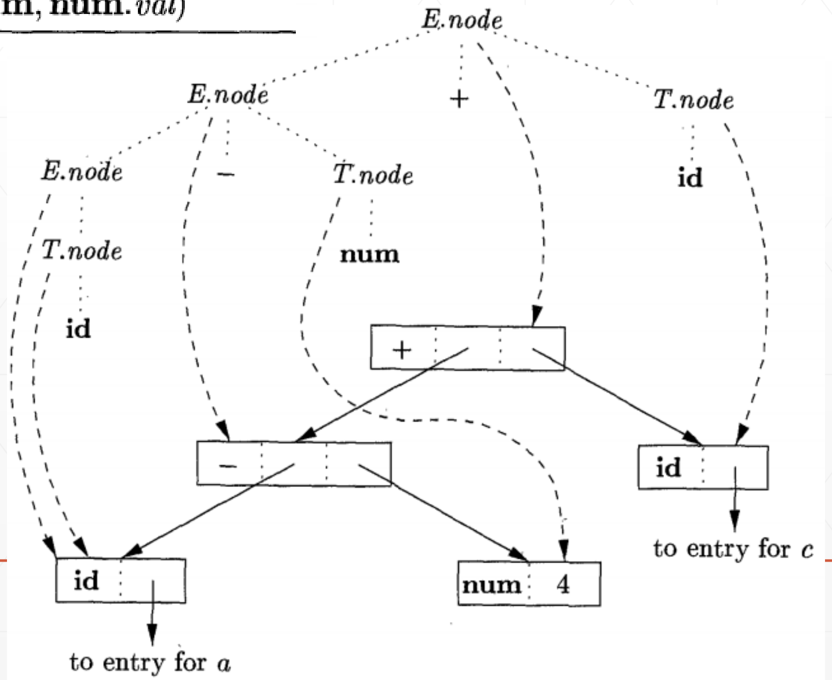




AST 构建

- Build the AST for Synthesized Attributes

	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)	$T \to \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$



应用: AST 构建

- Build the AST for Synthesized Attributes

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