



NEW YORK UNIVERSITY

G22.2130-001

Compiler Construction

# Lecture 8: Syntax-Directed Translation

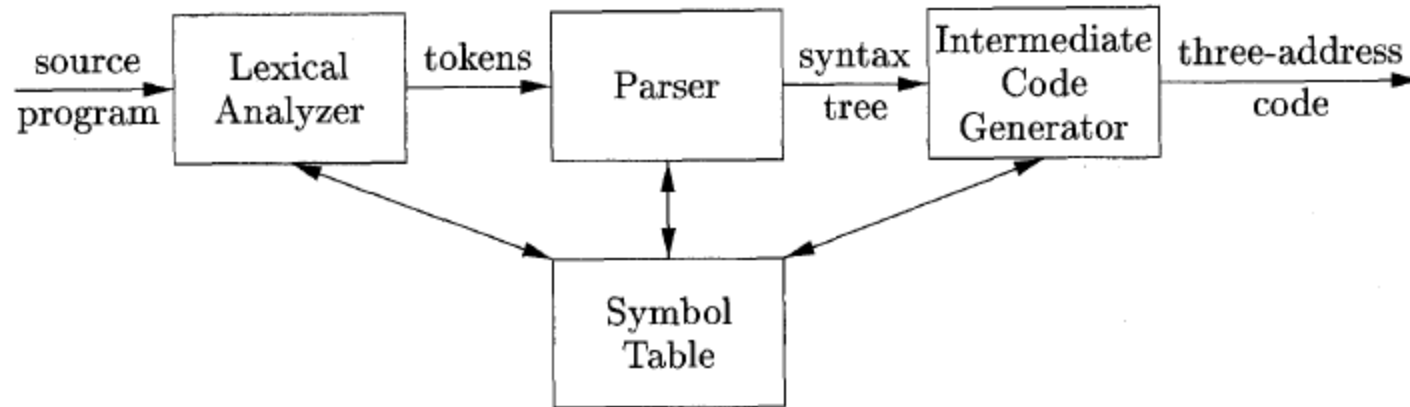
Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

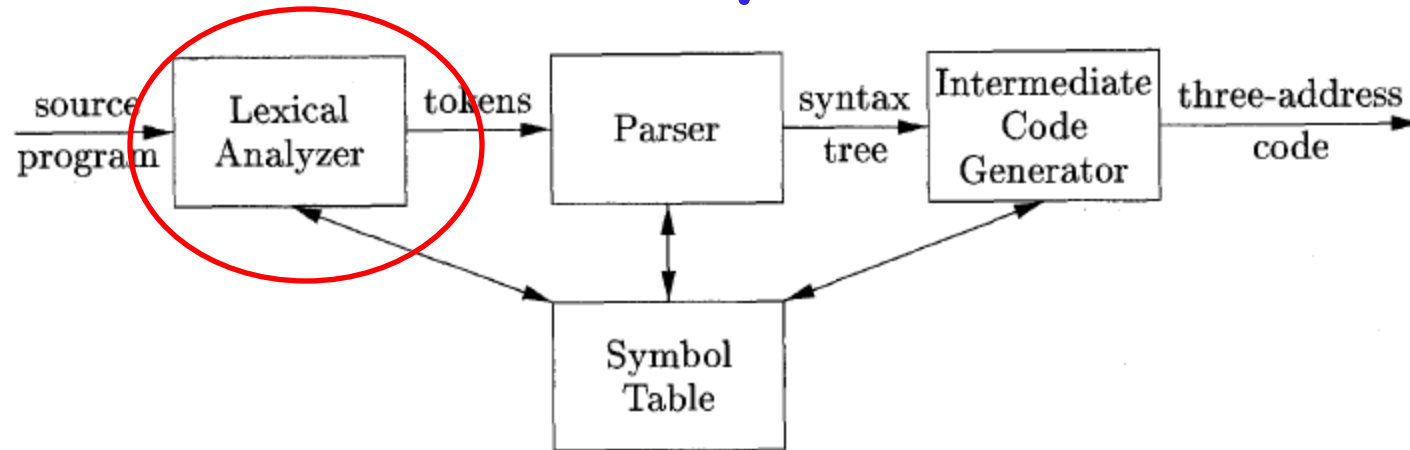


Copyright © Randy Glasbergen. www.glasbergen.com

# A Step-Back



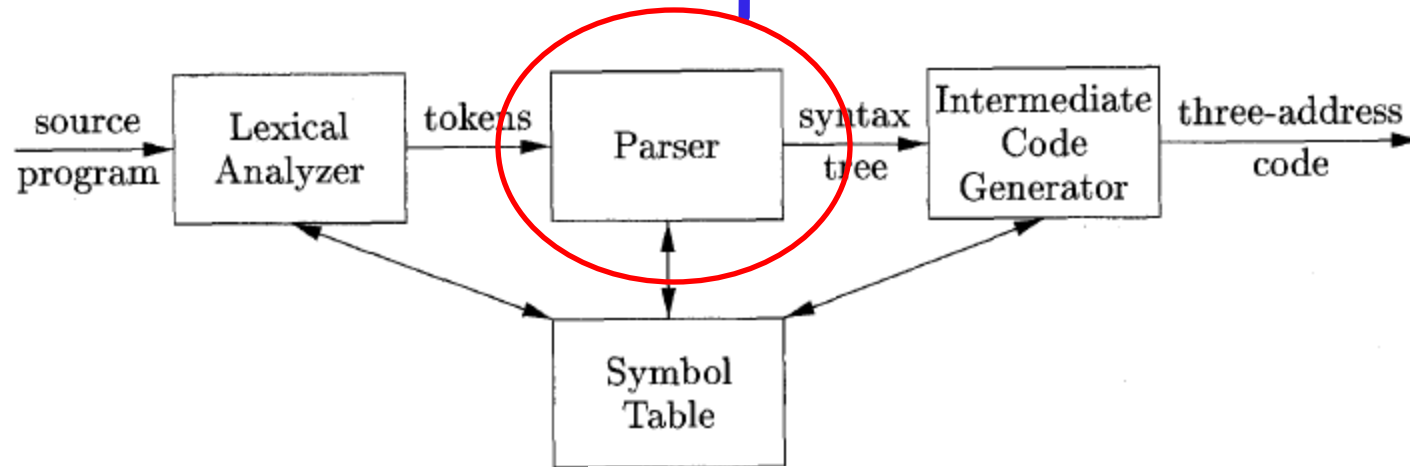
# A Step-Back



## Chapter 3

- Strings
- Regular expressions
- Tokens
- Transition diagrams
- Finite Automata

# A Step-Back



## Chapter 4

- Grammars
- Derivations
- Parse-trees
- Top-down parsing (LL)
- Bottom-up parsing (LR, SLR, LALR)

# Now What?

# We Need Some Tools

- To help in semantic analysis
- To help in intermediate code generation
- Two such tools
  - Semantic rules (Syntax-Directed Definitions)

PRODUCTION

$E \rightarrow E_1 + T$

SEMANTIC RULE

$E.code = E_1.code \parallel T.code \parallel '+'$

- Semantic actions (Syntax Directed Translations)

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

# Syntax-Directed Definitions

- Context-free grammar
- With attributes and rules to calculate the attributes

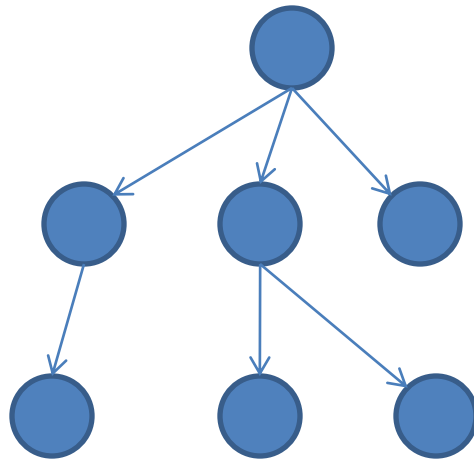
PRODUCTION

$E \rightarrow E_1 + T$

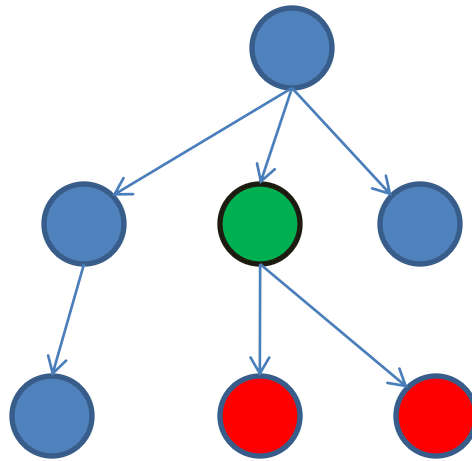
SEMANTIC RULE

$E.code = E_1.code \parallel T.code \parallel '+'$

# Two Types of Attributes



# Two Types of Attributes



SDD involving only synthesized attributes is called ***S-attributed***

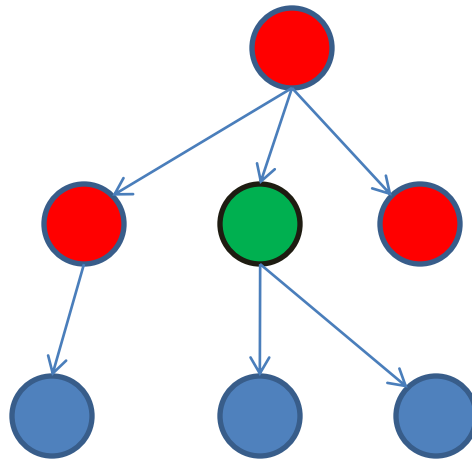
## Synthesized Attributes

Attribute of the node is defined in terms of:

- Attribute values at children of the node
- Attribute value at node itself



# Two Types of Attributes



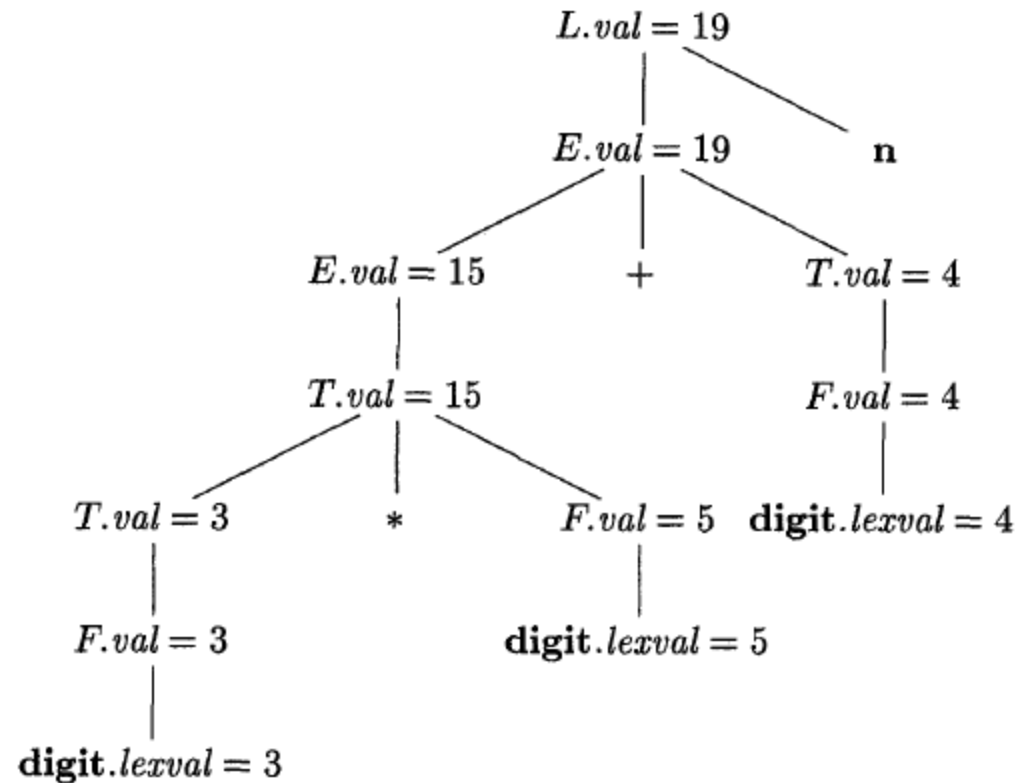
## Inherited Attributes

Attribute of the node is defined in terms of:

- Attribute values at parent of the node
- Attribute values at siblings
- Attribute value at node itself

$$3*5+4n$$

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



A parse tree showing the values of its attributes is called ***annotated parse tree***.

# Example

Give the annotated parse tree of  $(3+4)*(5+6)n$

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

# When Are Inherited Attributes Useful?

PRODUCTION
1) $T \rightarrow F T'$
2) $T' \rightarrow * F T'_1$
3) $T' \rightarrow \epsilon$
4) $F \rightarrow \mathbf{digit}$

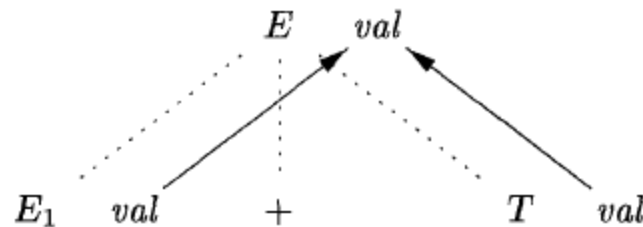
# Example

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

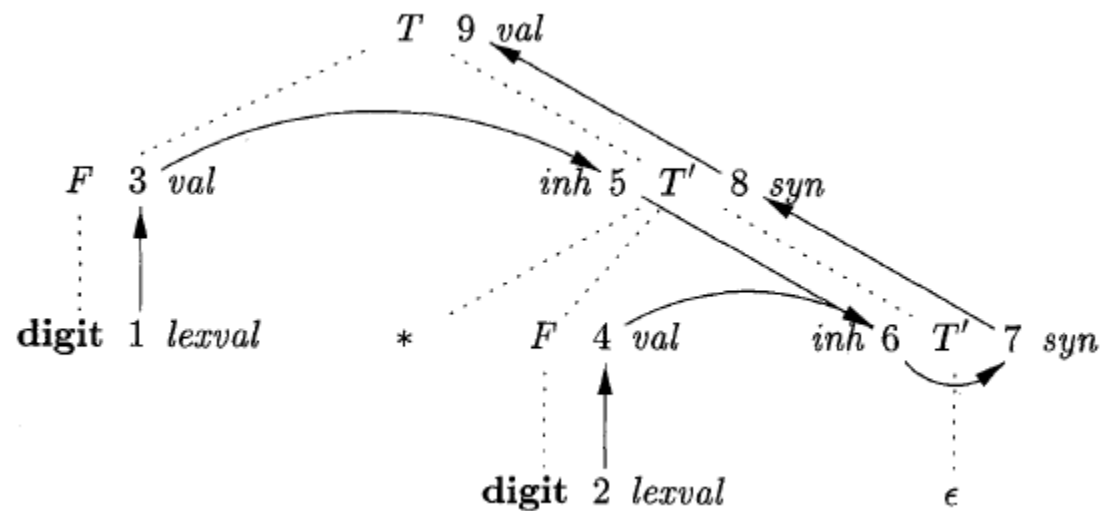
Give annotated parse-trees for:  
**int a, b, c**

# Evaluation Orders of SDDs

- Annotated parse tree shows attribute values
- **Dependency graph** helps us determine how those values are computed



# Topological Order



# S-Attributed Definitions

- Every attribute is synthesized
- We can evaluate its attribute in any bottom-up order of the nodes of the parse tree  
(e.g. postorder traversal -> LR parser).

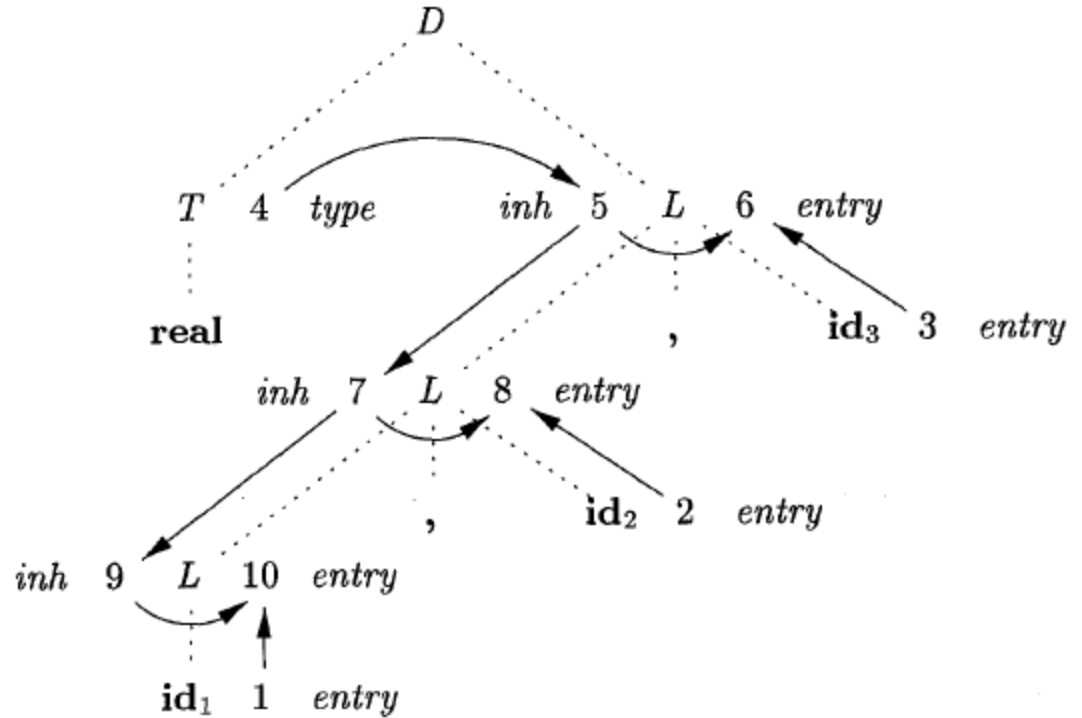


# L-Attributed Definitions

- Dependency graph edges can only go from left to right
  - i.e. use attributes from above or from the left

## Example

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



# Syntax-Directed Translations

- Context-free grammar
- Can implement SDDs
- Program fragments embedded within production bodies
  - called semantic rules
  - Can appear anywhere within the production body
- Steps are usually as follows
  - Build parse tree
  - perform actions as you traverse left-to-right, depth-first (preorder)

# Example

$L$	$\rightarrow$	$E \text{ n}$	$\{ \text{print}(E.val); \}$
$E$	$\rightarrow$	$E_1 + T$	$\{ E.val = E_1.val + T.val; \}$
$E$	$\rightarrow$	$T$	$\{ E.val = T.val; \}$
$T$	$\rightarrow$	$T_1 * F$	$\{ T.val = T_1.val \times F.val; \}$
$T$	$\rightarrow$	$F$	$\{ T.val = F.val; \}$
$F$	$\rightarrow$	$( E )$	$\{ F.val = E.val; \}$
$F$	$\rightarrow$	<b>digit</b>	$\{ F.val = \text{digit.lexval}; \}$

# Implementing L-Attributed SDDs

- L-attributed definitions can be used in many translation applications
- Several methods of implementation
  - Build parse tree and annotate
  - Build parse tree, add actions, execute in preorder
  - Recursive descent

# Recursive Descent

- Function  $A$  for each nonterminal  $A$
- Arguments of  $A$  are inherited attributes of nonterminal  $A$
- Return value of  $A$  is the collection of synthesized attributes of  $A$

# Example

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

For that rule we want to generate labels:

L1: C

L2: S1

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

$L1 = \mathit{new}();$

$L2 = \mathit{new}();$

$S_1.\mathit{next} = L1;$

$C.\mathit{false} = S.\mathit{next};$

$C.\mathit{true} = L2;$

$S.\mathit{code} = \mathbf{label} \parallel L1 \parallel C.\mathit{code} \parallel \mathbf{label} \parallel L2 \parallel S_1.\mathit{code}$

# Example

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

For that rule we want to generate labels:

L1: C

L2: S1

```
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();  
        L2 = new();  
        Ccode = C(next, L2);  
        check ')' is next on the input, and advance;  
        Scode = S(L1);  
        return("label" || L1 || Ccode || "label" || L2 || Scode);  
    }  
    else /* other statement types */  
}
```



# Example

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

For that rule we want to generate labels:

L1: C

L2: S1

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

```
void S(label next) {
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        check ')' is next on the input, and advance;
        print("label", L2);
        S(L1);
    }
    else /* other statement types */
}
```

# Reading

- Skim: 5.3, 5.4.3, 5.4.4, 5.4.5, 5.5.3, and 5.5.4
- Read the rest