Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1402-1403

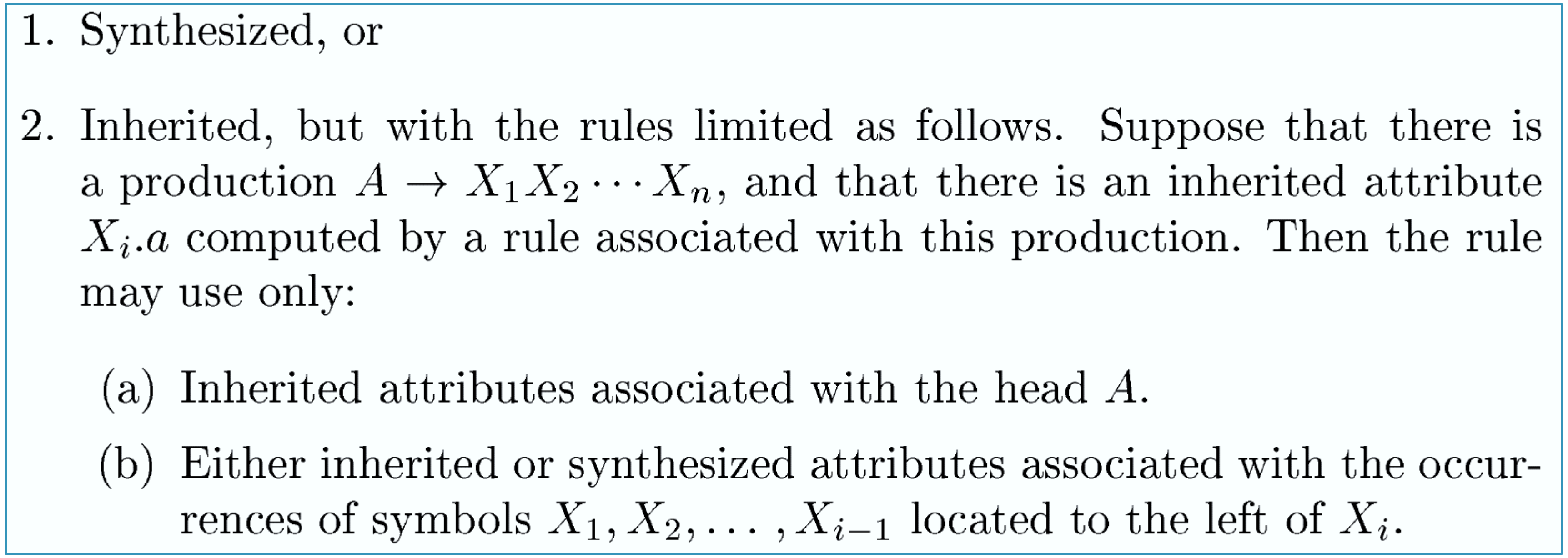
S-Attributed Definitions

- In practice, translations can be implemented using classes of SDDs that guarantee an evaluation order, since they do not permit dependency graphs with 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 of the parse tree

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

L-Attributed Definitions

- The idea behind this 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
- **An SDD is L-attributed if every attribute is**



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

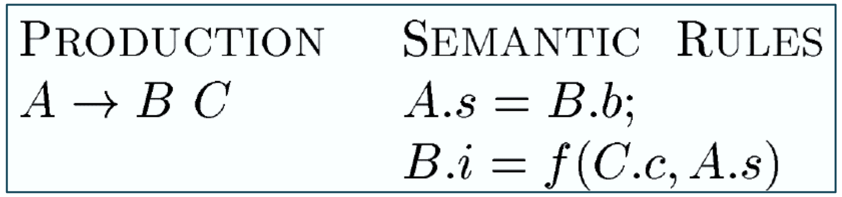
L-Attributed Definitions

- **Example**
  - This SDD is L-attributed

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

L-Attributed Definitions

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



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

SDD

- **Exercise:** Design an S-attributed SDD to compute $S.val$, the decimal-number value of an input string. For example, the translation of string 101.101 should be the decimal number 5.625

	Production	**Semantic rules**
1)	$S \rightarrow L_1 . L_2$	$S.val \ = L_1.val + L_2.val/L_2.f$
2)	$S \rightarrow L$	$S.val \ = L.val$
3)	$L \rightarrow L_1 B$	$L.val \ = L_1.val * 2 + B.val$
		$L.f \ = L_1.f * 2$
4)	$L \rightarrow B$	$L.val \ = B.val$
		$L.f \ = 2$
5)	$B \rightarrow 0$	$B.val = 0$
6)	$B \rightarrow 1$	$B.val = 1$

$$\boxed{\begin{array}{l} S \rightarrow L \ . \ L \mid L \\ L \rightarrow L \ B \mid B \\ B \rightarrow 0 \mid 1 \end{array}}$$

Syntax-Directed Translation Schemes

- Syntax-directed translation schemes are a complementary notation to syntax-directed definitions
- A syntax-directed translation scheme (SDT) is a context-free grammar with program fragments embedded within production bodies
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal

Postfix Translation Schemes

- By far 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 and is executed along with the reduction of the body to the head of that production
- SDTs with all actions at the right ends of the production bodies are called **postfix SDTs**

Postfix Translation Schemes

- **Example**

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

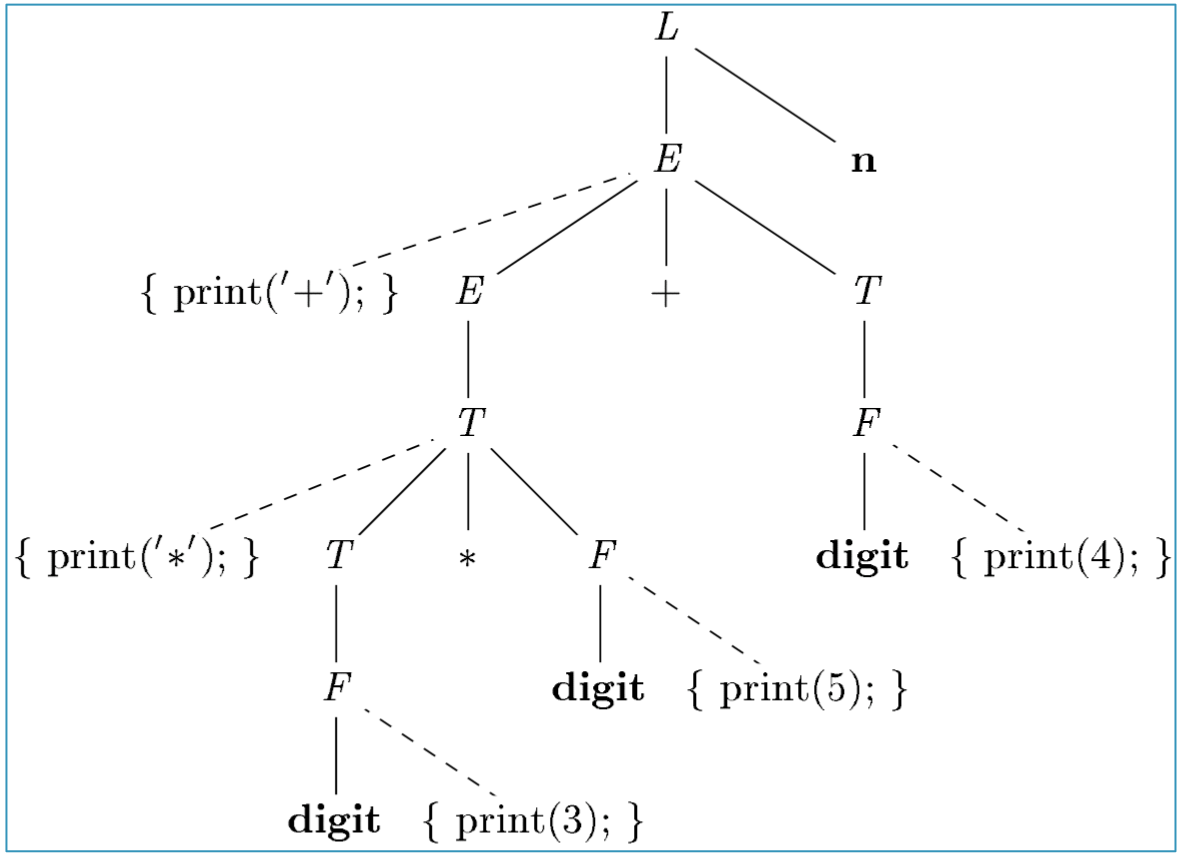
SDTs With Actions Inside Productions

- 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
- **Example**

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

SDTs With Actions Inside Productions

- Any SDT can be implemented as follows:
  1. Ignoring the actions, parse the input and produce a parse tree as a result
  2. Then, examine each interior node N, say one for production $A \rightarrow \alpha$. Add additional children to N for the actions in α
  3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action
- **Example:** the parse tree for expression 3 * 5 + 4
  - + * 3 5 4



Translation of Expressions

- **Goal**
  - Translation of expressions and statements into three-address code
- **Example**
  - Attributes ***S.code*** and ***E.code*** denote the three-address code for ***S*** and ***E***, respectively
  - Attribute ***E.addr*** denotes the address that will hold the value of ***E***

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \parallel gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \parallel gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Translation of Expressions

- **Example**
  - The SDD in the previous slide translates the assignment statement $\boldsymbol{a} = \boldsymbol{b} + -\boldsymbol{c}$; into the following three-address code sequence

```
t₁ = minus c
t₂ = b + t₁
a = t₂
```

Incremental Translation

- Code attributes can be long strings, so they are usually generated incrementally
- In the incremental approach, gen not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
 & & & \\
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
 & & & \\
 & | & -\ E_1 & \{\ E.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr);\ \} \\
 & & & \\
 & | & (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\
 & & & \\
 & | & \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \}
\end{array}
$$