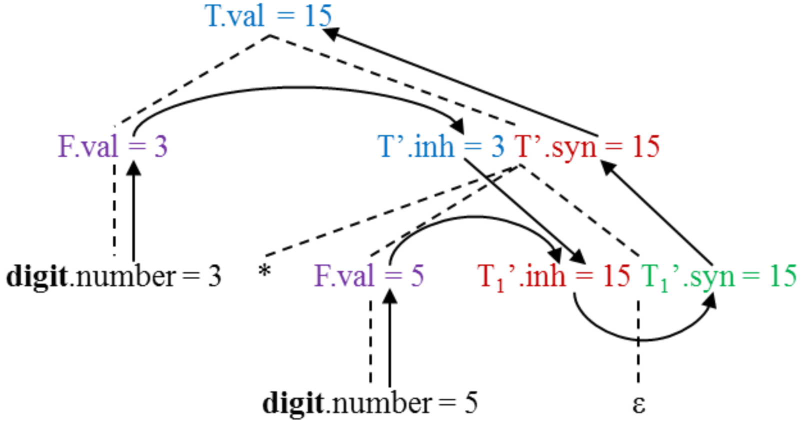
Syntax-Directed Translation



Syntax-Directed Definitions (SDD)

- Idea: attach attributes and rules to productions of a context-free grammar
  - attributes are associated to grammar symbols and either *synthesized* or *inherited*.
  - rules are associated to productions.

  - example: infix-to-postfix translator

Production			***Semantic Rule***
L	→	E **.**	$L.code = E.code$
E	→	$E_1 + T$	$E.code = E_1.code \mid\mid T.code \mid\mid$ '+'
E	→	T	$E.code = T.code$
T	→	$T_1 * F$	$T.code = T_1.code \mid\mid F.code \mid\mid$ '*'
T	→	F	$T.code = F.code$
F	→	**(** E **)**	$F.code = E.code$
F	→	**digit**	$F.code$ = **digit**.*number*

Syntax-Directed Definitions (SDD)

- Notation: Embedding semantic actions within production bodies
  - semantic actions are enclosed by curly braces
  - position of the semantic action in the production body determines the order in which the action is executed

Production — ***Semantic Rule***

$E \rightarrow E_1 + T$ — $E.code = E_1.code \,||\, T.code \,||\, \text{'+'}$

Syntax-Directed Translation Scheme

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

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

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

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Syntax-Directed Definitions (SDD)

- Example: expression evaluation using SDD

Syntax-Directed Translation Scheme

L	$\rightarrow$	E **.**	$\{ L.val = E.val \}$
E	$\rightarrow$	$E_1 + T$	$\{ E.val = E_1.val + T.val \}$
E	$\rightarrow$	T	$\{ E.val = T.val \}$
T	$\rightarrow$	$T_1 * F$	$\{T.val = T_1.val * F.val \}$
T	$\rightarrow$	F	$\{ T.val = F.val \}$
F	$\rightarrow$	**(** E **)**	$\{ F.val = E.val \}$
F	$\rightarrow$	**digit**	$\{ F.val =$ **digit**.*number* $\}$

Syntax-Directed Definitions (SDD)

- Example: AST generation using SDD

Syntax-Directed Translation Scheme

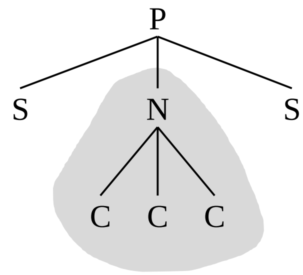
L	→	E **.**	{ **return** $E.node$; }
E	→	$E_1 + T$	{ **return new** BinOp('+', $E_1.node$, $T.node$); }
E	→	T	{ **return** $T.node$; }
T	→	$T_1 * F$	{ **return new** BinOp('*', $T_1.node$, $F.node$); }
T	→	F	{ **return** $F.node$; }
F	→	**(** E **)**	{ **return** $E.node$; }
F	→	**digit**	{ **return new** Number(**digit**.*number*); }

Syntax-Directed Definitions

- Why are SDDs useful?
  - on-the-fly code generation
    - rules are sequences of code
    - attributes designate memory addresses, registers
  - syntax tree generation
    - construct new class instances as a side-effect of executing a procedure in a recursive-descent parser. Return the node to the caller.

Synthesized vs. Inherited Attributes

- A *synthesized* attribute is defined only in terms of the attributes of the node itself and its children.

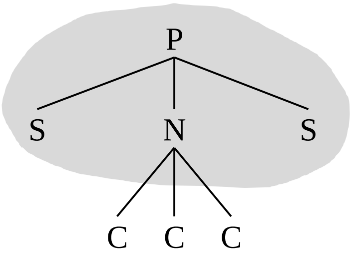


  - may be defined in terms of inherited attributes
  - terminal symbols can only have synthesized attributes
  - attribute is attached to the head of the production or a terminal

  - a grammar that has only synthesized attributes is called an S-attributed grammar

Synthesized vs. Inherited Attributes

- An *inherited* attribute is defined only in terms of the attributes of the node itself, its siblings and its parent.



  - inherited attributes may not be defined in terms of synthesized attributes
  - attribute is attached to a non-terminal in the body of a production

Evaluating SDDs w/ Annotated Parse Trees

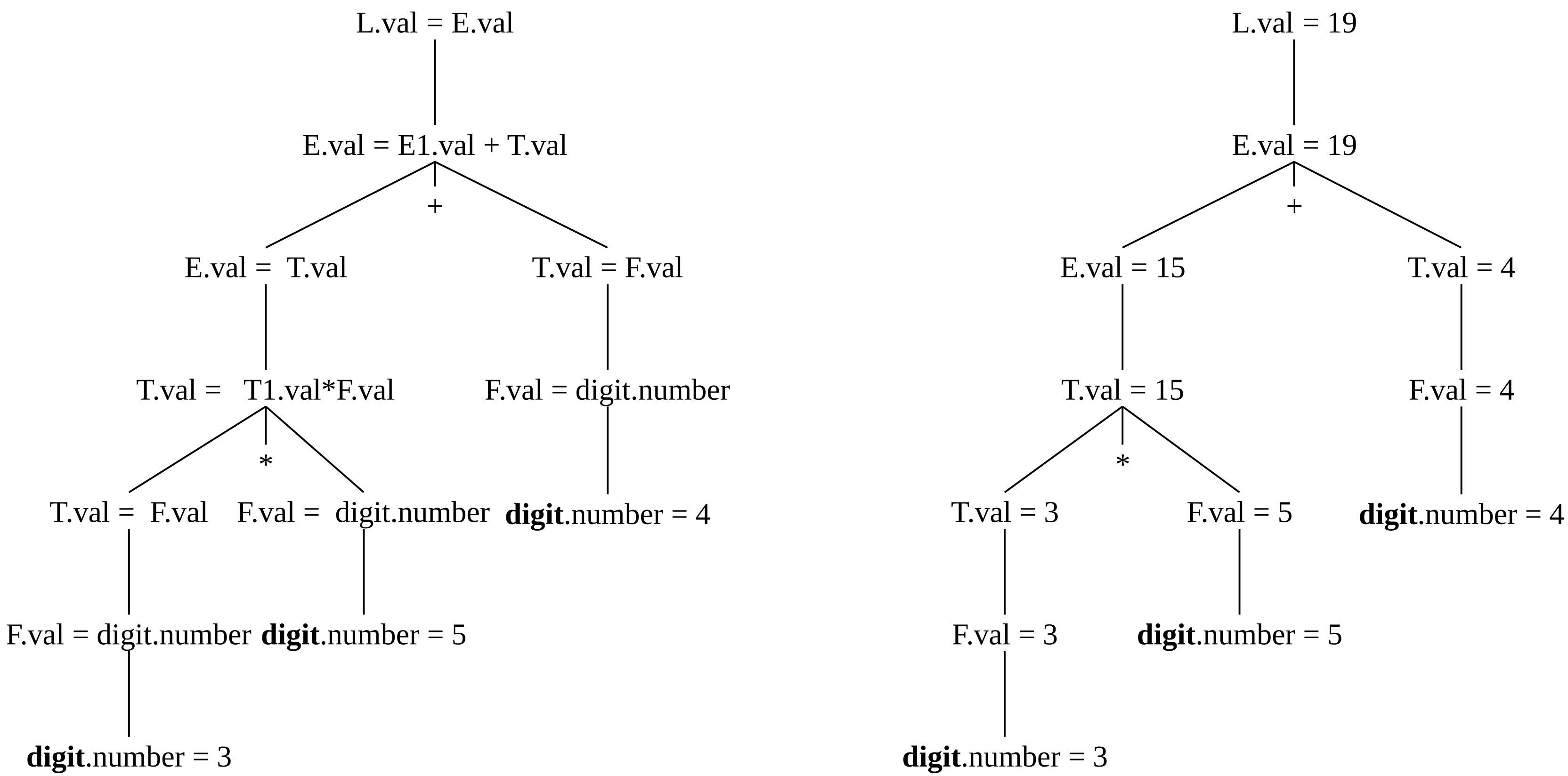
- An *annotated parse tree* is a visualization of syntax-directed definitions in the parse tree
  - attach attributes to nodes

- Problems to solve:
  - construction of the parse tree?
    - easy, just attach the attributes/rules to the node
  - evaluation of the attributes (order)?
    - synthesized: bottom-up
    - synthesized + inherited: use dependency graphs

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Annotated Parse Tree w/ Synthesized Attributes

- Annotated parse tree for “3 * 5 + 4.” with the SDD from slide 4
  - only synthesized attributes → bottom-up evaluation



Annotated Parse Tree w/ Inherited Attributes

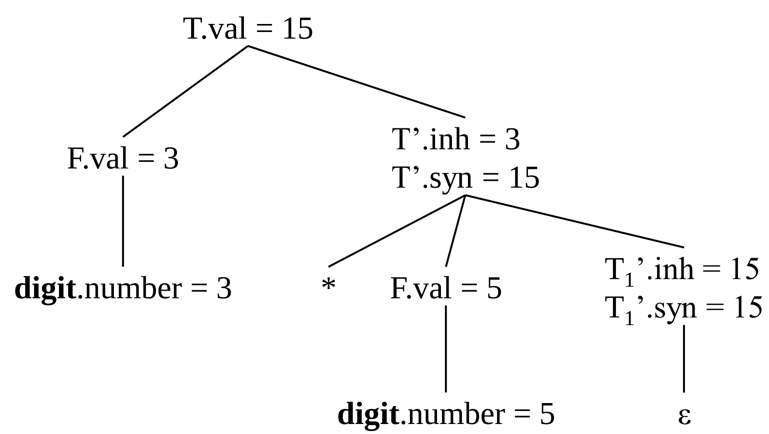
- Inherited attributes are useful when the structure of the grammar and the structure of the parse tree do not match
  - Elimination of left-recursion often leads to inherited attributes in SDDs:

Production			***Semantic Rule***
T	$\rightarrow$	$F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
T'	$\rightarrow$	$*\ F\ T_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
T'	$\rightarrow$	ε	$T'.syn = T'.inh$
F	$\rightarrow$	**digit**	$F.val =$ **digit**.*number*

Annotated Parse Tree w/ Inherited Attributes

- Annotated parse tree for “3 * 5”.

Production			***Semantic Rule***
T	→	$F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
T'	→	$*\ F\ T_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
T'	→	ε	$T'.syn = T'.inh$
F	→	**digit**	$F.val =$ **digit**.*number*



Evaluation Order for SDDs

- Problem: in which order should we evaluate the attributes?
- Idea: use a dependency graph
  - dependency graph defines a topological order
  - follow the order to evaluate the attributes

Dependency Graphs

- A dependency graph depicts the flow of information among the attribute instances in a particular parse tree

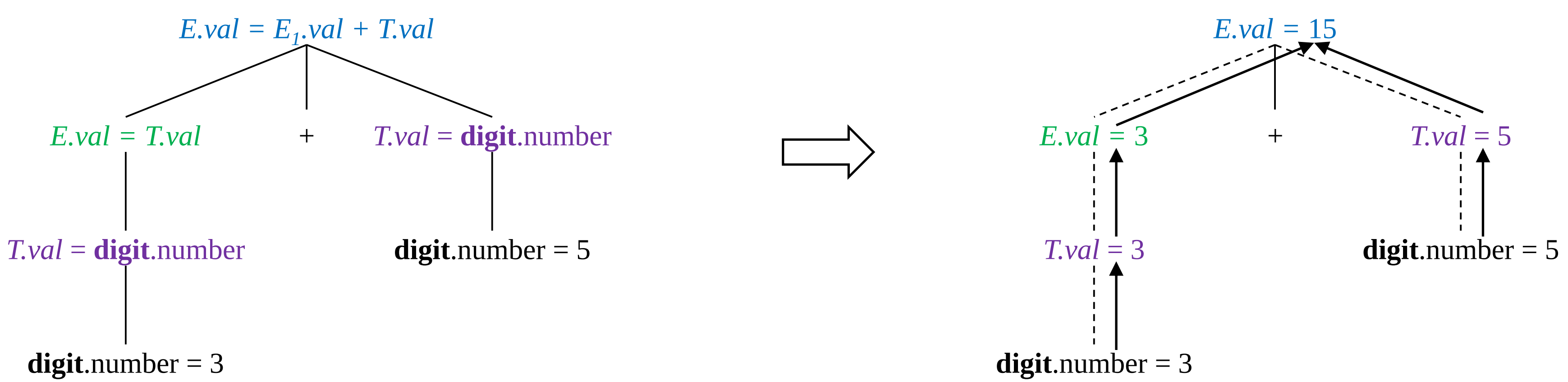
  - the dependency graph has a node for each attribute associated with X for every node labeled by grammar symbol X in the parse tree

  - for each synthesized attribute A.b defined in terms of the value of X.c, the dependency graph has an edge from X.c to A.b
    (note that the node representing X is a child of the node representing A)

  - for each inherited attribute B.c defined in terms of the value of X.a, the dependency graph has an edge from X.a to B.c
    (note that the node representing X can be a parent or a sibling of the node representing B)

Evaluation Order of Annotated Parse Trees

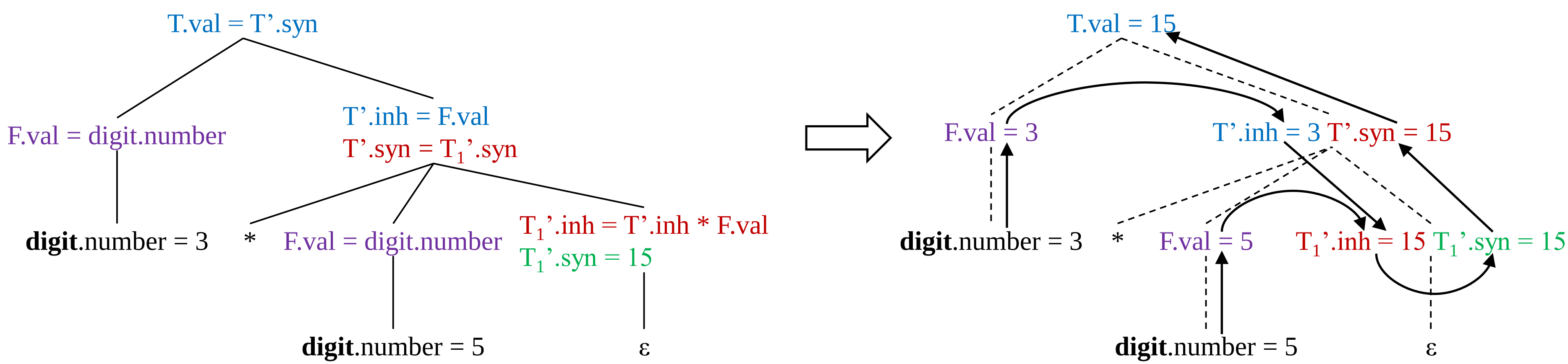
- SDD with synthesized attributes for “3 * 5”



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

Evaluation Order of Annotated Parse Trees

- SDD with inherited attributes for "3 * 5"



Production			***Semantic Rule***
T	→	*F T'*	*T'.inh = F.val* *T.val = T'.syn*
T'	→	$* F T_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
T'	→	ε	*T'.syn = T'.inh*
F	→	**digit**	*F.val =* ***digit****.number*

Evaluation Order of Dependency Graphs

- Evaluation order
  - if the dependency graph has an edge from node M to N, then the attribute corresponding to M must be evaluated before N
  - that is, the only valid order of evaluations are sequences N_1, N_2, …, N_k such that if there is an edge in the dependency graph from N_i to N_j, then $i < j$ (topological sort)
  - if there is a cycle, there is exists no topological sort (and the graph cannot be evaluated); if there are no cycles, at least one topological sort exists.
  - in general, it is hard to tell whether there exist any parse trees whose dependency graph have cycles.

S-Attributed Definitions

- S-attributed definitions describes the class of SDDs that only have synthesized attributes
  - there always exists a topological order
  - evaluate by postorder graph traversal (bottom up)

```
postorder(Node n)
{
  for (each child c of n, from the left) postorder(c);
  evaluate attributes of n;
}
```

    - this order corresponds exactly to the order in which an LR parser reduces a production to its head

L-Attributed Definitions

- L-attributed Definitions
  - class of SDDs with synthesized and inherited attributes, but dependency edges only go from "left-to-right"
    Each attribute must be either
    - synthesized or
    - inherited but with the following restrictions:
      for production $A \rightarrow X_1X_2 \ldots X_n$ and inherited attribute $X_i.a$ computed by a rule associated with this production, the rule may use only
      - inherited attributes associated with the head A
      - 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.
      - inherited or synthesized attributes associated with the occurrence of X_i itself but only so that there are no cycles formed by the attributes of X_i
  - i.e., L-attributed definitions have no cycles and can thus always be evaluated
  - a top-down parser for a grammar with eliminated left-recursion leads to an L-attributed SDD

L-Attributed Definitions

- Example

Production			*Semantic Rule*
T	$\rightarrow$	$F\ T'$	$T'.inh = F.val$
T'	$\rightarrow$	$*\ F\ T_1'$	$T_1'.inh = T'.inh * F.val$

Production			*Semantic Rule*
A	$\rightarrow$	BC	$A.syn = B.b$ $B.inh = f\,(C.c, A.syn)$

Rules with Controlled Side Effects

- Attribute grammars
  - no side effects
  - allow any evaluation order consistent with the dependency graph

- In practice, we need side effects → control side effects in SDDs
  - permit incidental side effects that do not constrain attribute evaluation i.e., permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a correct translation

Production			***Semantic Rule***
L	→	*E.*	*print (E.val)*
...			

  - constrain the allowable evaluation orders so that the same translation is produced for any allowable order
    - implemented by adding implicit edges to the dependency graph

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Rules with Controlled Side Effects

- Example: variable declarations

Production			***Semantic Rule***
D	→	$T\ L$	$L.inh = T.type$
T	→	**int**	$T.type$ = integer
T	→	**float**	$T.type$ = float
L	→	L_1, **id**	$L_1.inh = L.inh$ addType(**id**.*entry*, *L*.*inh*)
L	→	**id**	addType(**id**.*entry*, *L*.*inh*)

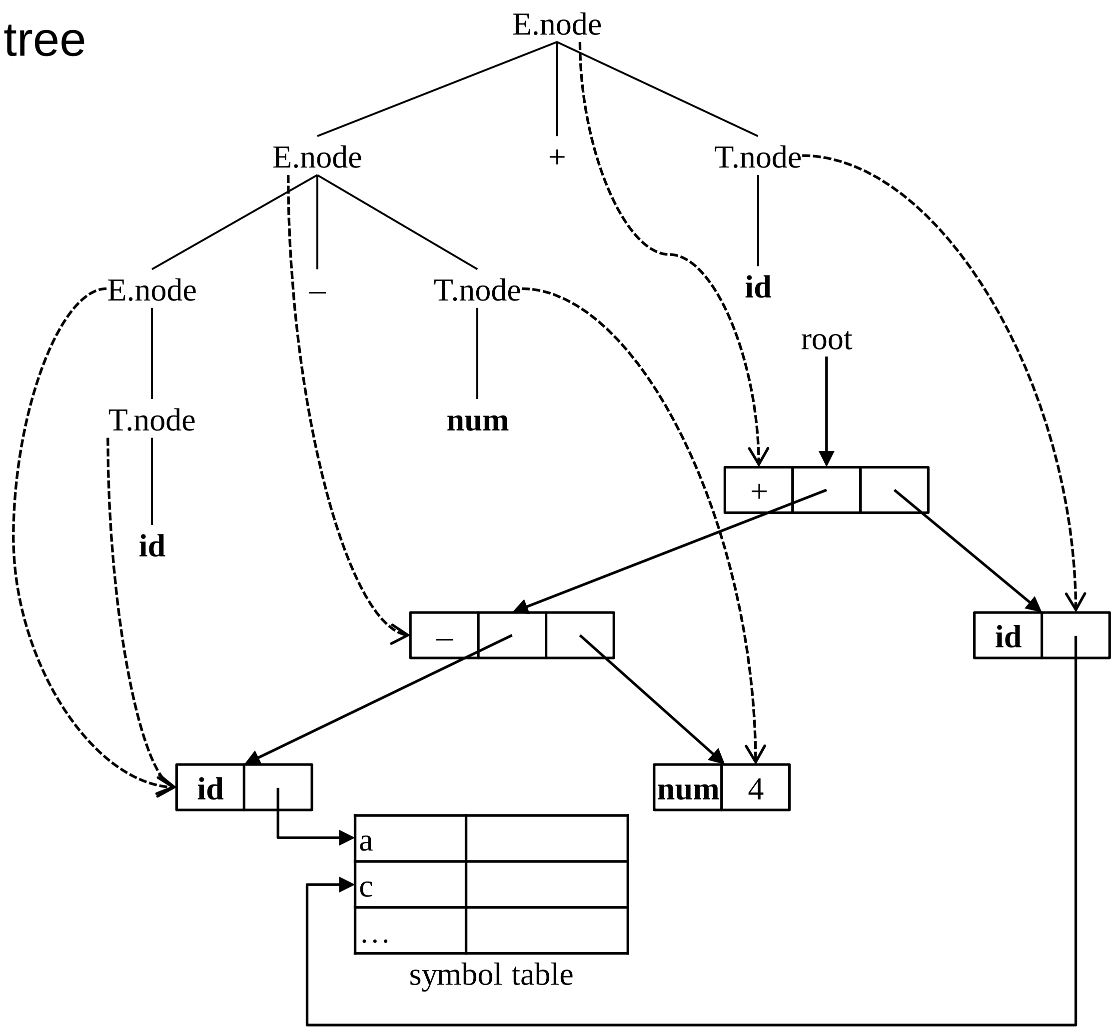
Using SDT to Construct Syntax Trees

- During a bottom-up parse or a postorder traversal of the parse tree
  - S-attributed definition
    - actions comprise creating objects for nodes in the syntax tree

Production			***Semantic Rule***
E	$\rightarrow$	$E_1 + T$	$E.node$ = new Node('+', $E_1.node$, $T.node$)
E	$\rightarrow$	$E_1 - T$	$E.node$ = new Node('– ', $E_1.node$, $T.node$)
E	$\rightarrow$	T	$E.node = T.node$
T	$\rightarrow$	**(** E **)**	$T.node = E.node$
T	$\rightarrow$	**id**	$T.node$ = new Leaf(**id**, **id**.*entry*)
T	$\rightarrow$	**num**	$T.node$ = new Leaf(**num**, **num**.*val*)

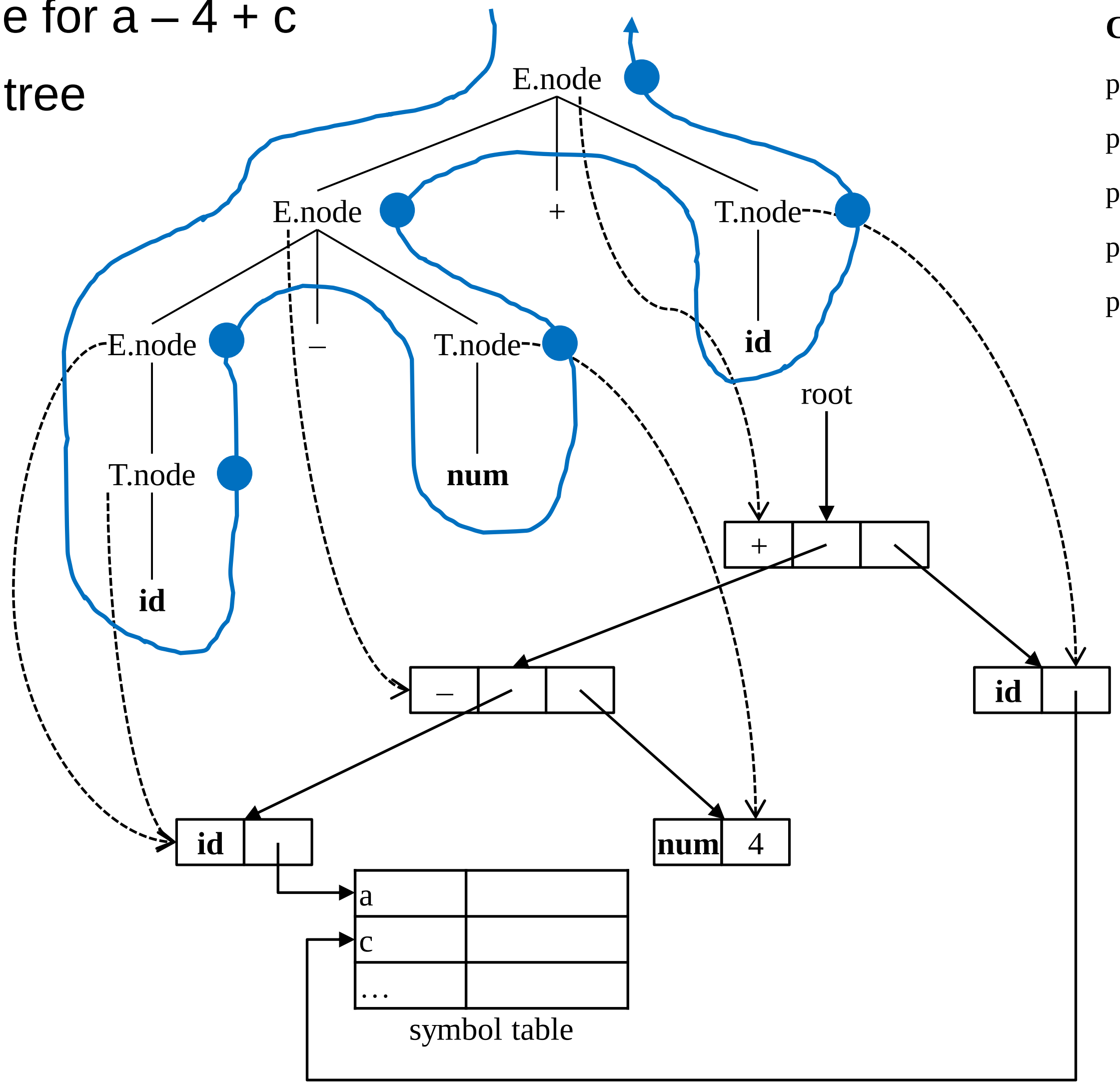
Using SDTs to Construct Syntax Trees

- Syntax tree for a – 4 + c
  - parse tree



Using SDTs to Construct Syntax Trees

- Syntax tree for a – 4 + c
  - parse tree



Construction Steps

p_1 = **new** Leaf(**id**, entry-a)

p_2 = **new** Leaf(**num**, 4)

p_3 = **new** Node(**'-'**, p_1, p_2)

p_4 = **new** Leaf(**id**, entry-c)

p_5 = **new** Node(**'-'**, p_3, p_4)

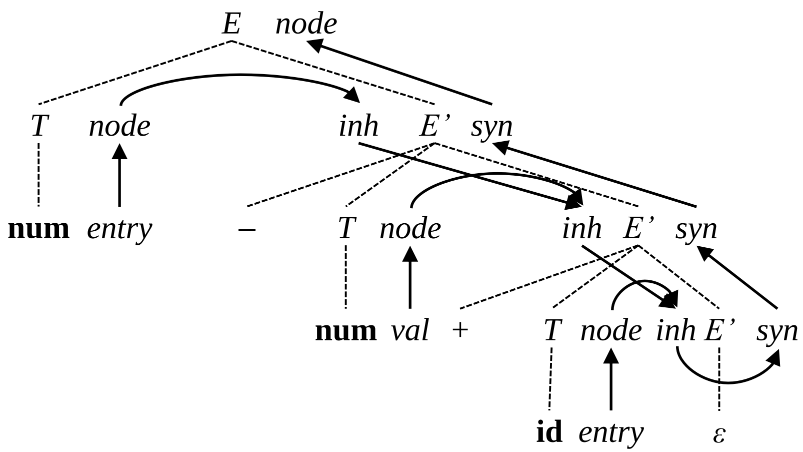
Using SDTs to Construct Syntax Trees

- L-attributed definition for a top-down parser

Production	***Semantic Rule***
$E \rightarrow T\,E'$	$E.node = E'.syn$ $E'.inh = T.node$
$E' \rightarrow +\,T\,E_1'$	$E_1'.inh =$ **new** Node('+', $E'.inh$, $T.node$) $E'.syn = E1'.syn$
$E' \rightarrow -\,T\,E_1'$	$E_1'.inh =$ **new** Node('–', $E'.inh$, $T.node$) $E'.syn = E1'.syn$
$E' \rightarrow \varepsilon$	$E'.syn = E'.inh$
$T \rightarrow (\,E\,)$	$T.node = E.node$
$T \rightarrow$ **id**	$T.node =$ **new** Leaf(**id**, **id**.entry)
$T \rightarrow$ **num**	$T.node =$ **new** Leaf(**num**, **num**.val)

Using SDTs to Construct Syntax Trees

- Syntax tree for a – 4 + c
  - parse tree and dependency graph



Production			***Semantic Rule***
E	→	*T E'*	*E.node = E'.syn* *E'.inh = T.node*
E'	→	+ *T E$_1$'*	*E$_1$'.inh* = **new** Node('+', *E'.inh*, *T.node*) *E'.syn = E1'.syn*
E'	→	– *T E$_1$'*	*E$_1$'.inh* = **new** Node('–', *E'.inh*, *T.node*) *E'.syn = E1'.syn*
E'	→	ε	*E'.syn = E'.inh*
T	→	(*E*)	*T.node = E.node*
T	→	**id**	*T.node* = **new** Leaf(**id, id**.entry)
T	→	**num**	*T.node* = **new** Leaf(**num, num**.val)

The Structure of a Type

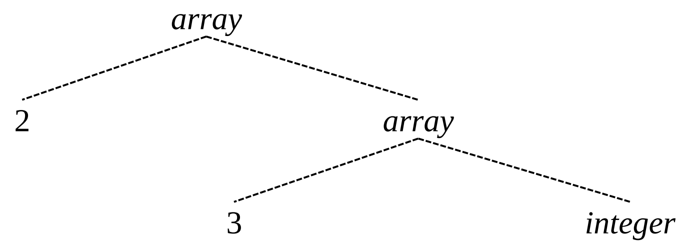
- Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input

- In C

  ```
  int [2][3]
  ```

  actually means

  "array of 2 arrays of 3 integers", or array(2, array(3, integer))



The Structure of a Type

- Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input
- In C

```
int [2][3]
```

actually means

"array of 2 arrays of 3 integers", or array(2, array(3, integer))

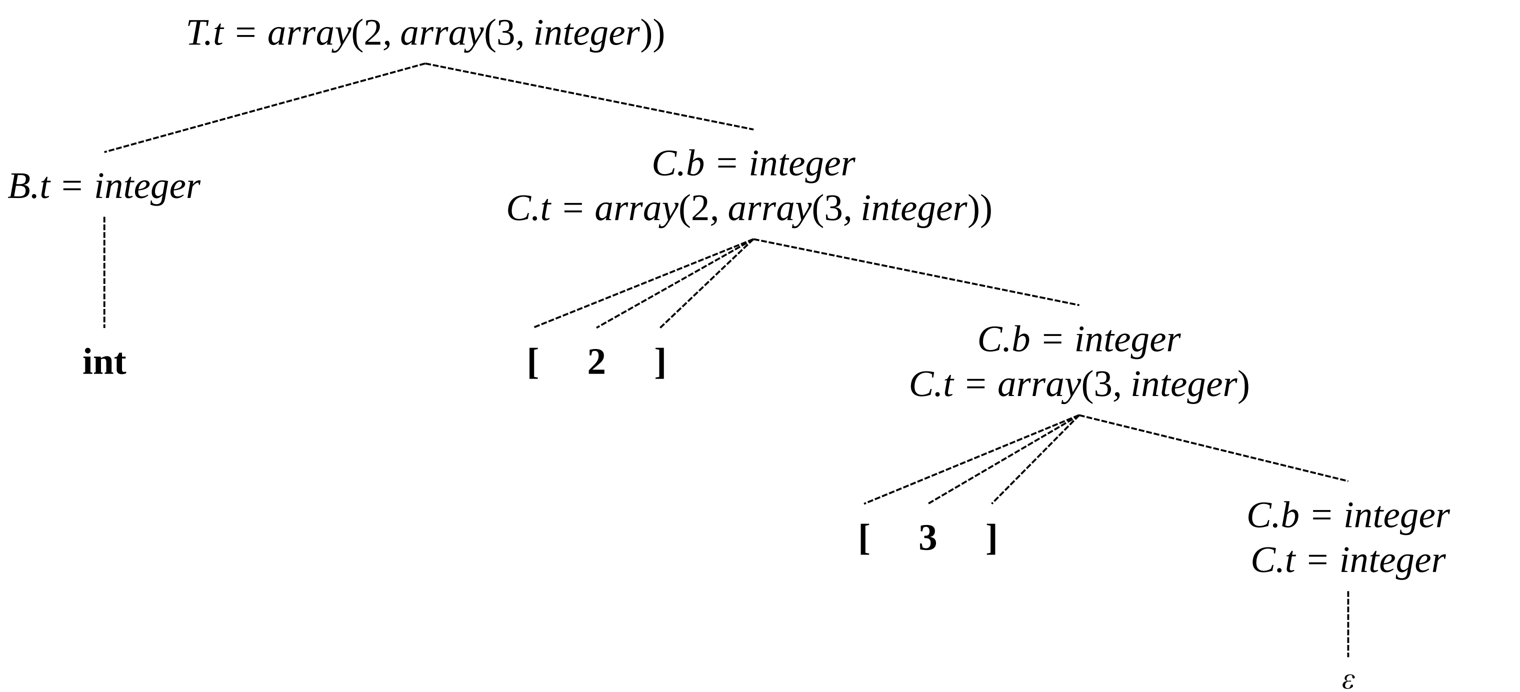
Production			***Semantic Rule***
T	→	BC	$T.t = C.t$ $C.b = B.t$
B	→	**int**	$B.t = \mathit{integer}$
B	→	**float**	$B.t = \mathit{float}$
C	→	[**num**] C_1	$C.t = \text{array}(\textbf{num}.\text{val}, C_1.t)$ $C_1.b = C.b$
C	→	ε	$C.t = C.b$

The Structure of a Type

- C array types

```
int [2][3]
```

Production			***Semantic Rule***
T	→	BC	$T.t = C.t$ $C.b = B.t$
B	→	**int**	$B.t = integer$
B	→	**float**	$B.t = float$
C	→	[**num**] C_1	$C.t =$ array(**num**.val, $C_1.t$) $C_1.b = C.b$
C	→	ε	$C.t = C.b$



CSE 컴퓨터공학부 Department of Computer Science & Engineering

Syntax-Directed Translation (SDT) Schemes

- Complementary notation to syntax-directed definitions
  - used to implement SDDs

- Semantic actions (program fragments) are embedded in the bodies of the production rules

- Execution (implementation) of an SDT:
  - build parse tree
  - perform actions left-to-right, depth-first (preorder traversal)

Postfix Translation Schemes for SDT

- Simplest method when the grammar can be parsed bottom-up and the SDD is S-attributed
  - semantic actions placed at the right end of the production body
  - actions are executed along with the reduction (i.e., when the body is reduced to the head)
    - results in a postorder traversal

Postfix Translation Schemes for SDT

- Calculator Example

Postfix Syntax-Directed Translation Scheme

L	$\rightarrow$	E **.**	{ *print*($E.val$); }
E	$\rightarrow$	$E_1 + T$	{ $E.val = E_1.val + T.val$; }
E	$\rightarrow$	T	{ $E.val = T.val$; }
T	$\rightarrow$	$T_1 * F$	{$T.val = T_1.val * F.val$; }
T	$\rightarrow$	F	{ $T.val = F.val$; }
F	$\rightarrow$	**(** E **)**	**{** $F.val = E.val$; }
F	$\rightarrow$	**digit**	**{** $F.val$ = **digit**.*number*; }

Postfix Translation Schemes for SDT

- Parser-Stack Implementation
  - place attributes on stack along with handles
  - execute actions when reductions occur

			X	Y	Z	state/grammar symbol
			$X.x$	$Y.y$	$Z.z$	synthesized attribute(s)

top

Postfix Translation Schemes for SDT

- Parser-Stack Implementation of the Calculator

Production	***Action***
$L \rightarrow E\,.$	{ print(*stack*[*top*-1]); *top* = *top*-1; }
$E \rightarrow E_1 + T$	{ *stack*[*top*-2].*val* = *stack*[*top*-2].*val* + *stack*[*top*-1].*val*; *top* = *top*-2; }
$E \rightarrow T$	
$T \rightarrow T_1 * F$	{ *stack*[*top*-2].*val* = *stack*[*top*-2].*val* * *stack*[*top*-1].*val*; *top* = *top*-2; }
$T \rightarrow F$	
$F \rightarrow (\,E\,)$	{ *stack*[*top*-2].*val* = *stack*[*top*-1].*val* ; *top* = *top*-2; }
$F \rightarrow$ **digit**	

SDT's With Actions Inside Productions

- Conceptually, an action within the body of a production is executed as soon as all symbols to its left have been processed.

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

- Postfix and L-attributed SDT's can be implemented during a bottom-up and top-down parse, respectively.

- If the SDT is neither postfix nor L-attributed, it can be implemented as follows
  1. parse input and build a parse tree (ignoring all actions)
  2. for each (inner) node N add actions of node N as children in the order of appearance in the SDT
  3. during a preorder traversal, perform the actions of all actions nodes

SDT's With Actions Inside Productions

- Example: infix → prefix form translator

Postfix Syntax-Directed Translation Scheme

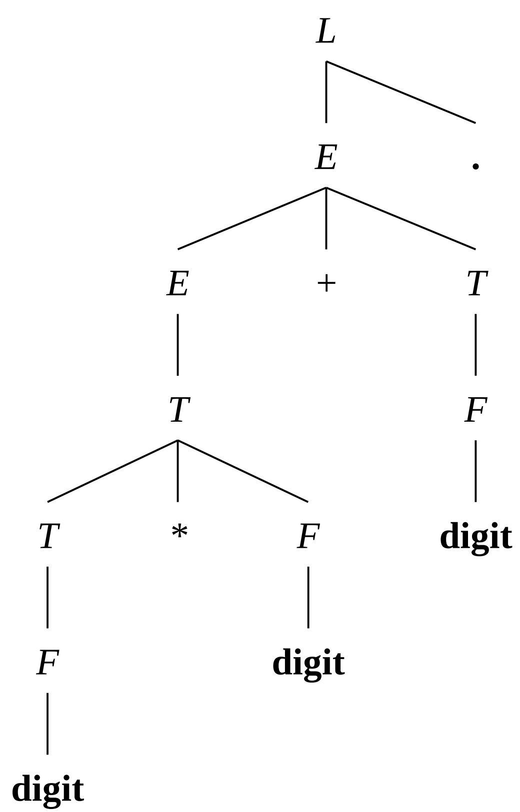
L	→	E **.**
E	→	{ print('+'); } $E_1 + T$
E	→	T
T	→	{ print('*'); } $T_1 * F$
T	→	F
F	→	**(** E **)**
F	→	**digit** { print(**digit**.lexval); }

SDT's With Actions Inside Productions

- Example: infix → prefix form translator

$$3 * 5 + 4$$

- parse tree



Postfix Syntax-Directed Translation Scheme

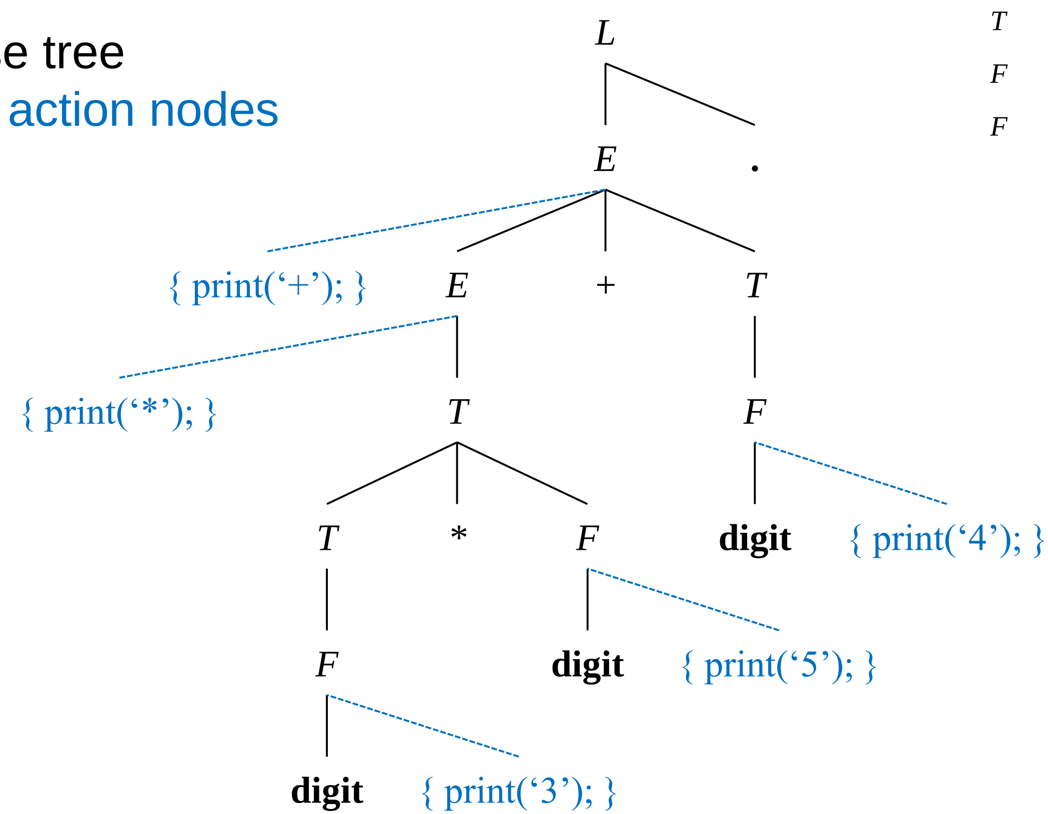
L	→	E .
E	→	{ print('+'); } E_1 + T
E	→	T
T	→	{ print('*'); } T_1 * F
T	→	F
F	→	**(** E **)**
F	→	**digit** { print(**digit**.lexval); }

SDT's With Actions Inside Productions

- Example: infix → prefix form translator

  3 * 5 + 4

  - parse tree
    with action nodes



Postfix Syntax-Directed Translation Scheme

L	→	E **.**
E	→	{ print('+'); } E_1 + T
E	→	T
T	→	{ print('*'); } T_1 * F
T	→	F
F	→	**(** E **)**
F	→	**digit** { print(**digit**.lexval); }

SDT's With Actions Inside Productions

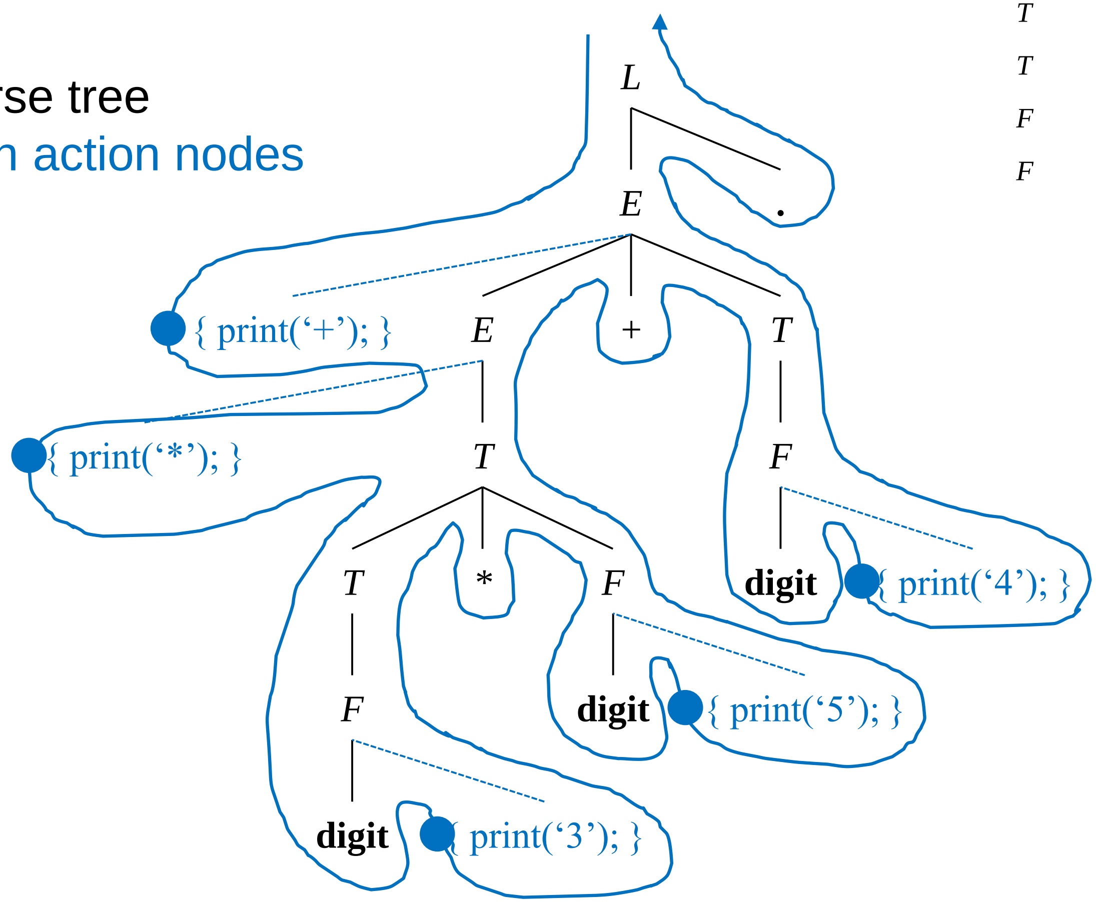
- Example: infix → prefix form translator

Postfix Syntax-Directed Translation Scheme

L	→	E .
E	→	{ print('+'); } $E_1 + T$
E	→	T
T	→	{ print('*'); } $T_1 * F$
T	→	F
F	→	(E)
F	→	**digit** { print(**digit**.lexval); }

3 * 5 + 4

- parse tree
  with action nodes



SDT's With Actions Inside Productions

- Executing actions during parsing

  We have seen that an action within the body of a production must be executed as soon as all symbols to its left have been processed

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

  Therefore, during a

  - bottom-up parse: perform action a as soon as X appears on top of the parsing stack
  - top-down parse: perform action a immediately before expanding Y

SDT's With Actions Inside Productions

- Not all SDT's can be implemented during parsing

  Consider again

Postfix Syntax-Directed Translation Scheme

L	→	E **.**
E	→	{ print('+'); } $E_1 + T$
E	→	T
T	→	{ print('*'); } $T_1 * F$
T	→	F
F	→	**(** E **)**
F	→	**digit** { print(**digit**.lexval); }

- the parser would have to perform the actions { print('+'); } / { print('*'); } before it knows which production will be applied

SDT's With Actions Inside Productions

- Which SDT's cannot be implemented during parsing?

  Replace actions with marker nonterminals M_i and check for conflicts

Postfix Syntax-Directed Translation Scheme

L	→	E **.**
E	→	$M_1\ E_1 + T$
E	→	T
T	→	$M_2\ T_1 * F$
T	→	F
F	→	**(** E **)**
F	→	**digit** M_3
M_1	→	ε
M_2	→	ε
M_3	→	ε

- bottom-up parser: conflicts on reductions $M_1 \rightarrow \varepsilon$, $M_2 \rightarrow \varepsilon$, and shifting the digit
- top-down parser: grammar is left recursive

Eliminating Left-Recursion from SDT's

- What happens to the actions when eliminating left-recursion from a SDT?

- Simple case when only the order of actions must be preserved (i.e., if all actions only print something)
  1. treat actions as terminal symbols
  2. eliminate left-recursion as usual

$$
\begin{array}{lll}
E & \rightarrow & E_1 + T \; \{ \text{ print '+'); } \} \\
E & \rightarrow & T
\end{array}
\quad\Rightarrow\quad
\begin{array}{lll}
E & \rightarrow & E_1 + T \; \mathbf{a_1} \\
E & \rightarrow & T
\end{array}
$$

$$
\Downarrow
$$

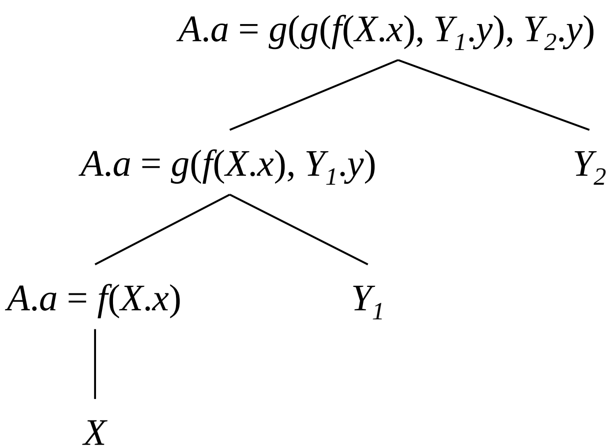
$$
\begin{array}{lll}
E & \rightarrow & T\,R \\
R & \rightarrow & + \; T \; \{ \text{ print '+'); } \} \; R \\
R & \rightarrow & \varepsilon
\end{array}
\quad\Leftarrow\quad
\begin{array}{lll}
E & \rightarrow & T\,R \\
R & \rightarrow & + \; T \; \mathbf{a1} \; R \\
R & \rightarrow & \varepsilon
\end{array}
$$

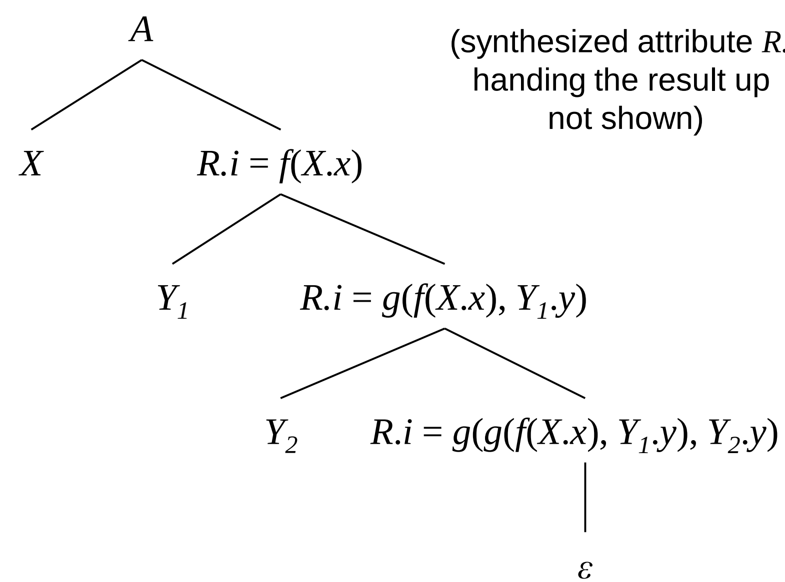
Eliminating Left-Recursion from SDT's

- If the actions compute attributes (instead of just printing some output), eliminating left-recursion is more complicated.
- The following always schema works for S-attributed SDT's

$$
\begin{array}{lll}
A & \rightarrow & A_1\ Y\ \{\ A.a = g(A_1.a,\ Y.y)\ \} \\
A & \rightarrow & X\ \{\ A.a = f(X.x)\ \}
\end{array}
\quad \xRightarrow[\text{(goal)}]{} \quad
\begin{array}{lll}
A & \rightarrow & X\ R \\
R & \rightarrow & Y\ R \mid \varepsilon
\end{array}
$$

  - for XYY





(synthesized attribute $R.s$ handing the result up not shown)

Eliminating Left-Recursion from SDT's

- If the actions compute attributes (instead of just printing some output), eliminating left-recursion is more complicated.
- The following always schema works for S-attributed SDT's

$$
\begin{aligned}
A &\rightarrow A_1\, Y \; \{\, A.a = g(A_1.a,\, Y.y) \,\} \\
A &\rightarrow X \; \{\, A.a = f(X.x) \,\}
\end{aligned}
\qquad \overset{\Longrightarrow}{\text{(goal)}} \qquad
\begin{aligned}
A &\rightarrow X\, R \\
R &\rightarrow Y\, R \mid \varepsilon
\end{aligned}
$$

$\Downarrow$

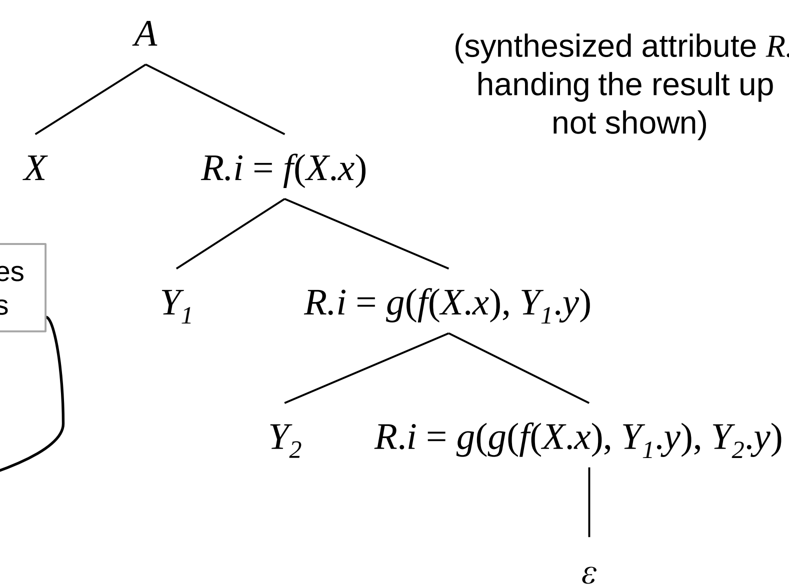
compute inherited attributes of a non-terminal immediately before its use in a production

$$
\begin{aligned}
A &\rightarrow X \; \{\, R.i = f(X.x) \,\} \; R \\
R &\rightarrow Y \; \{\, R_1.i = g(R.i,\, Y.y) \,\} \; R_1 \\
R &\rightarrow \varepsilon
\end{aligned}
$$

$\Downarrow$

compute synthesized attributes at the end of the productions

$$
\begin{aligned}
A &\rightarrow X \; \{\, R.i = f(X.x) \,\} \; R \; \{\, A.a = R.s \,\} \\
R &\rightarrow Y \; \{\, R_1.i = g(R.i,\, Y.y) \,\} \; R_1 \; \{\, R.s = R_1.s \,\} \\
R &\rightarrow \varepsilon \; \{\, R.s = R.i \,\}
\end{aligned}
$$



(synthesized attribute $R.s$ handing the result up not shown)

SDT's for L-Attributed Definitions

- Postfix translation schemes only work for S-attributed SDD's

- More general case: L-attributed SDD's

  Conversion rules

  1. embed the action that computes the *inherited* attribute for a non-terminal A immediately before that occurrence of A in the body of the production. If A has more than one inherited attribute, choose an order according to a topological sort of the dependence graph.
  2. place the actions that compute a *synthesized* attribute for the head of a production at the end of the body of that production.

SDT's for L-Attributed Definitions

- Example: immediate code generation for a `while` statement

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

  - code for the condition C and the statement sequence S_1 are generated directly by the respective non-terminals
  - control flow implemented by issuing statements of the form "**label** L"
  - attributes
    - $S.next$ labels the beginning of the code to be executed after S is finished
    - $S.code$ intermediate code that implements S and ends with a jump to $S.next$
    - $C.true$ labels the beginning of the code to be executed if C is true
    - $C.false$ idem for $C == false$
    - $C.code$ intermediate code that implements C and jumps to $C.true/false$ depending on whether C evaluates to true or false

SDT's for L-Attributed Definitions

- Example: immediate code generation for a `while` statement

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

  - L-attributed SDD

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

CSE 컴퓨터공학부
Department of Computer Science & Engineering

SDT's for L-Attributed Definitions

- Example: immediate code generation for a `while` statement

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

  - Conversion to SDT according to the rules establieshed before (slide 47)
  - remaining issue
    - $L1$, $L2$ are variables, not attributes
    - like before treat actions as dummy non-terminals
      → variables can be viewed as synthesized attributes of those non-terminals

SDT

S	→	**while (**	{ $L1 = newLabel(); L2 = newLabel(); C.false = S.next; C.true = L2;$ }
		C **)**	{ $S_1.next = L1;$ }
		S_1	{ $S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;$ }

Translation of SDT's during Top-Down Parsing

- Given: a recursive-descent parser with one function per non-terminal

- For each non-terminal A extend the corresponding function `A()` as follows:
  - the *arguments* to `A()` are the inherited attributes of non-terminal A

  - the *body* of `A()` needs to parse as well as deal with actions in A
    - decide which production to apply (possibly using the lookahead)
    - consume terminals when they appear in the production
    - call functions corresponding to non-terminals and provide them with the proper arguments
    - (use local variables as needed to save/preserve attributes)

  - the *return value* of `A()` is the collection of synthesized attributes of non-terminal A

Translation of SDT's during Top-Down Parsing

- `while` example from earlier

```
string S(label next) {
  string Scode, Ccode;
  label L1, L2;

  switch (token) {
    …
    case tWHILE:
      consume(tWHILE);  consume('(');
      L1 = newLabel(); L2 = newLabel();
      Ccode = C(next, L2);
      consume(')');
      Scode = S(L1);
      return 'label' || L1 || Ccode || 'label'|| L2 || Scode;
    …
  }
}
```

SDT

S	$\rightarrow$	**while (**	{ $L1$ = $newLabel$(); $L2$ = $newLabel$(); $C.false$ = $S.next$; $C.true$ = $L2$; }
		C **)**	{ $S_1.next$ = $L1$; }
		S_1	{ $S.code$ = **label** ‖ $L1$ ‖ $C.code$ ‖ **label** ‖ $L2$ ‖ $S_1.code$; }

Translation of SDT's during Top-Down Parsing

- Same example, *but on-the-fly code-generation* instead of returning strings

```
void S(label next) {
  label L1, L2;

  switch (token) {
    …
    case tWHILE:
      consume(tWHILE);  consume('(');
      L1 = newLabel(); L2 = newLabel();
      print("label %s\n", L1);
      C(next, L2);
      consume(')');
      S(L1);
      print("label %s\n", L2);
    …
  }
}
```