Chapter 5

Syntax-Directed Translation

Overview

mansoori@shirazu.ac.ir

- Translation of languages guided by CFGs
- Attaching attributes to grammar symbols which represent the language constructs
- A syntax-directed definition (SDD) specifies the values of attributes by associating semantic rules with grammar productions
- A syntax-directed translation scheme (SDT) embeds program fragments (semantic actions) within production bodies

SDD vs. SDT

mansoori@shirazu.ac.ir

- Infix-to-postfix translator
  - SDD:

PRODUCTION	SEMANTIC RULE
$E \to E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

  - SDT: $E \to E_1 + T \;\{ \text{print } '+' \}$

- SDD: more readable ⇒ more useful for specifications
- SDT: more efficient ⇒ more useful for implementations

- Two class of syntax-directed translations
  - L-attributed: translation is done during top-down parsing
  - S-attributed: translation is done with bottom-up parsing

mansoori@shirazu.ac.ir

SDD

What is SDD?

mansoori@shirazu.ac.ir

- SDD = CFG + semantic rules
- SDD binds semantic rules to productions
- Terminals/nonterminals have attributes holding values
- Attributes are set by semantic rules

X: symbol

a: attribute of X

X.a: value of a at parse-tree node labeled X

Attributes

mansoori@shirazu.ac.ir

- Attributes may be of any kind
  - Numbers & strings
  - Table references & memory locations
  - Data types
  - Scoping information for local declarations
  - Intermediate code representation

- Two kinds of attributes for nonterminals
  - Synthesized attributes
  - Inherited attributes

Synthesized Attributes

mansoori@shirazu.ac.ir

- Synthesized attribute A.s for nonterminal A at parse-tree node N is defined by:
  - A semantic rule associated with production $A \rightarrow B\ b\ C$ at N
  - Only in terms of attributes at N and its children

$A.s = f\,(B.s,\ b.s,\ C.s)$ where attributes of B, b, C are synthesized

Production	Semantic rules
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
	$E.p = E_1.p \parallel T.p \parallel \text{'+'}$

$E.val = 14$

$E.val = 12$ $\quad$ $+$ $\quad$ $T.val = 2$

$E.p = \text{'34*2+'}$

$E.p = \text{'34*'}$ $\quad$ $+$ $\quad$ $T.p = \text{'2'}$

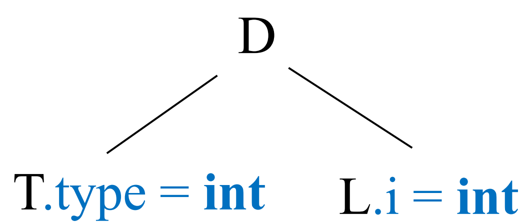
Inherited Attributes

mansoori@shirazu.ac.ir

- Inherited attribute C.i for nonterminal C at parse-tree node N is defined by:
  - A semantic rule associated with production $A \rightarrow B\ b\ C$ at the parent of N
  - Only in terms of attributes at N's parent, N itself, and N's siblings

$C.i = f\,(A.i,\ B.i,\ B.s,\ b.s)$

Production	Semantic rules
$D \rightarrow T\ L$	$L.i = T.type$



Annotated Parse Tree

mansoori@shirazu.ac.ir

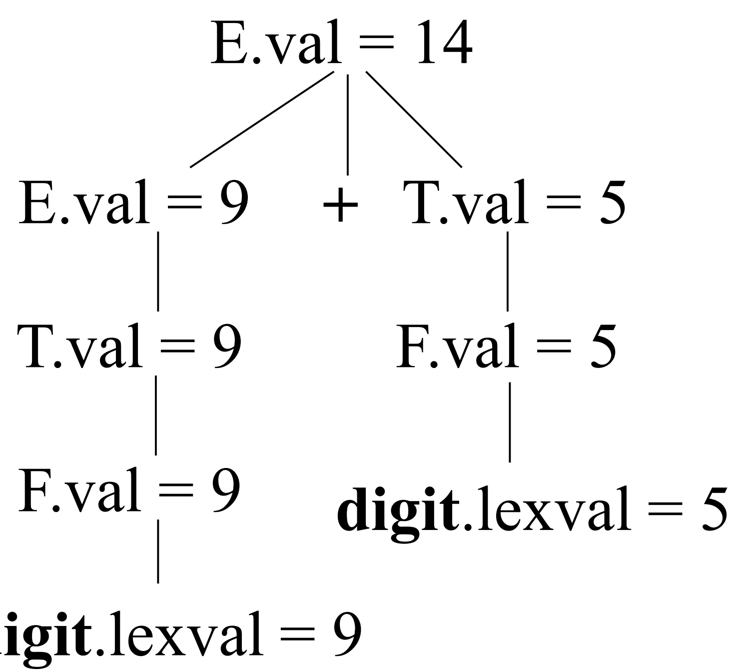
- Parse tree helps to visualize the translation by SDD
- Translators need not build parse trees
- Using the semantic rules of SDD to evaluate the attributes at nodes of parse tree (annotated parse tree)
- A depth-first postorder algorithm traverses the parse tree thereby executing semantic rules to assign attribute values
- After the traversal is complete the attributes contain the translated form of the input

Evaluating SDD at Nodes of Parse Tree (Only Synthesized Attributes)

masoori@shirazu.ac.ir

- Synthesized attributes can be evaluated in any bottom-up order (postorder traversal of parse tree)

Production	Semantic rules
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval



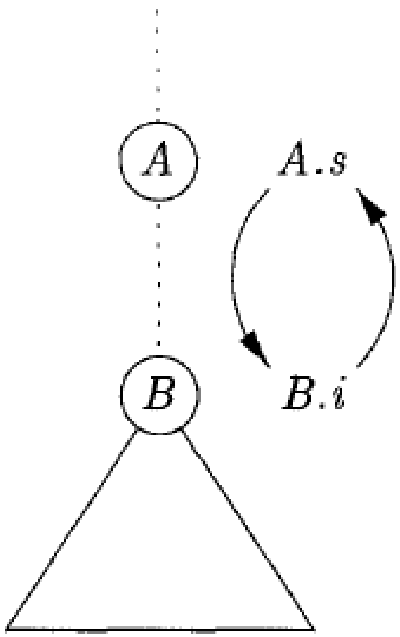
Annotated parse tree for: **9+5**

Evaluating SDD at Nodes of Parse Tree
(Both Synthesized & Inherited Attributes)

mansoori@shirazu.ac.ir

- No guarantee that there is even one order to evaluate attributes at nodes

Production	Semantic rules
$A \rightarrow B$	$A.s = B.i$
	$B.i = A.s + 1$

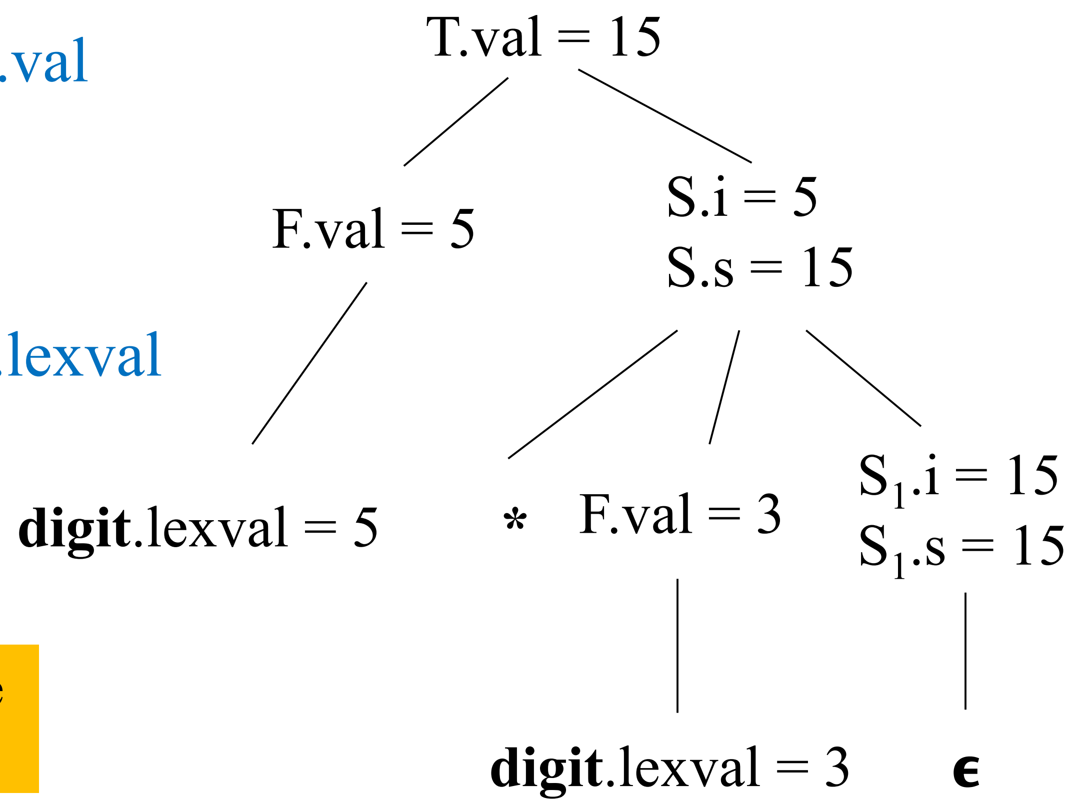


- Semantic rules are circular
- Impossible to evaluate
  - Either A.s at node N
  - Or B.i at child of N

Evaluating SDD at Nodes of Parse Tree
(Both Synthesized & Inherited Attributes)

mansoori@shirazu.ac.ir

Production	Semantic rules
$T \rightarrow F\ S$	$S.i = F.val$ $T.val = S.s$
$S \rightarrow *\ F\ S_1$	$S_1.i = S.i * F.val$ $S.s = S_1.s$
$S \rightarrow \varepsilon$	$S.s = S.i$
$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

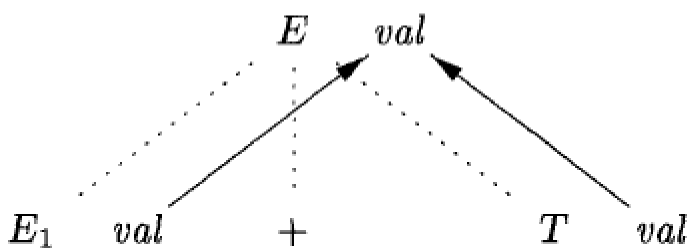


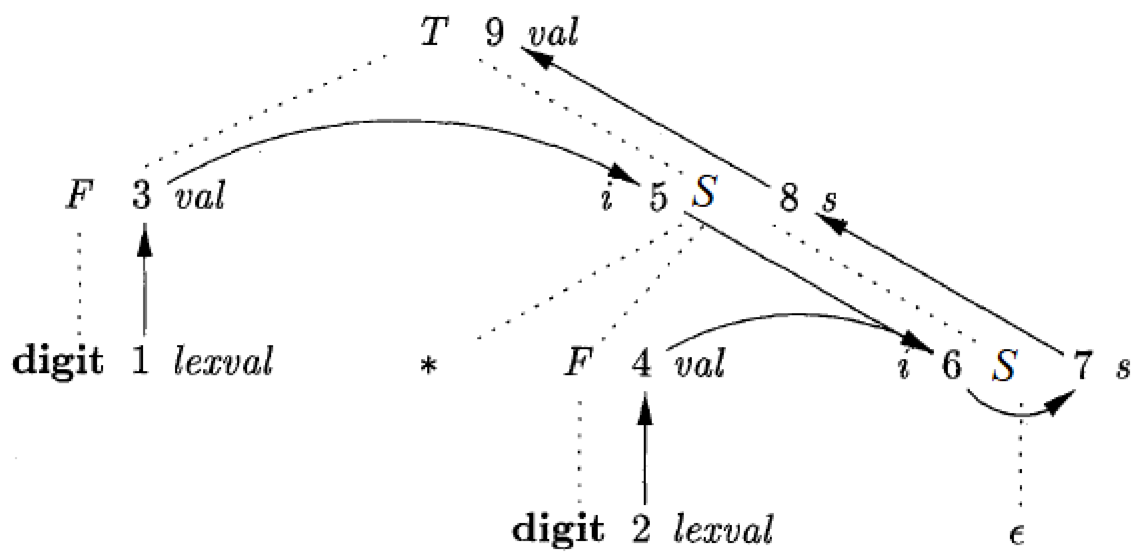
Annotated parse tree for: **5*3**

Dependency Graph

mansoori@shirazu.ac.ir

- Depicts the flow of information among the attribute instances in a parse tree
  - An edge from one attribute instance to another means that value of first is needed to compute the second
  - Edges express constraints implied by semantic rules





S-attributed SDD

mansoori@shirazu.ac.ir

- An SDD is S-attributed if every attribute is synthesized
- Each rule computes an attribute for head's nonterminal of a production from attributes of production's body
- Its attributes can be evaluated in any bottom-up order of the nodes of parse tree
  - By performing a postorder traversal of parse tree and evaluating the attributes at a node N when the traversal leaves N for the last time

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

S-attributed SDD

mansoori@shirazu.ac.ir

- Can be implemented during bottom-up parsing
  - In conjunction with an LR parser

Production	Semantic rules
$L \to E$ **n**	Print(E.val)
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to$ **digit**	F.val = **digit**.lexval

S-attributed SDD in Yacc

mansoori@shirazu.ac.ir

- Yacc/Bison only support S-attributed definitions

```
%token DIGIT
%%
L : E '\n'              { printf("%d\n", $1); }
  ;
E : E '+' T             { $$ = $1 + $3; }
  | T                   { $$ = $1; }
  ;
T : T '*' F             { $$ = $1 * $3; }
  | F                   { $$ = $1; }
  ;
F : '(' E ')'           { $$ = $2; }
  | DIGIT               { $$ = $1; }
  ;
%%
```

Synthesized attribute of parent node **F**

L-attributed SDD

mansoori@shirazu.ac.ir

An SDD is L-attributed if each attribute is either synthesized or inherited with these limiting rules:

- For production $A \rightarrow X_1\ X_2 \ldots\ X_n$ and inherited attribute $X_i.a$, compute $X_i.a$ by a rule which uses only:

  (a) Inherited attributes of head A

  (b) Inherited/Synthesized attributes of $X_1, X_2, \ldots, X_{i-1}$ located to left of X_i

  (c) Inherited/Synthesized attributes of X_i itself (no cycles in dependency graph of this X_i)

Note: every S-attributed SDD is also L-attributed

L-attributed SDD

mansoori@shirazu.ac.ir

- Inherited attributes
  - Are useful when the structure of parse tree differs from abstract syntax of input
  - Can be used to carry information from one part of the parse tree to another

- An L-attributed SDD:

Production	Semantic rules
$S \rightarrow * F S_1$	$S_1.i = S.i * F.val$ $S.s = S_1.s$

- A non L-attributed SDD:

Production	Semantic rules
$S \rightarrow A B$	$S.s = A.a$ $A.i = f(B.b, S.s)$

Semantic Rules with Side Effects

Declaration of Identifiers

mansoori@shirazu.ac.ir

- Semantic rules are allowed to have side-effects
  - Printing the result
  - Interacting with symbol table

Production	Semantic rules
$L \rightarrow E\ \mathbf{n}$	print(E.val)

- A simple declaration of identifiers

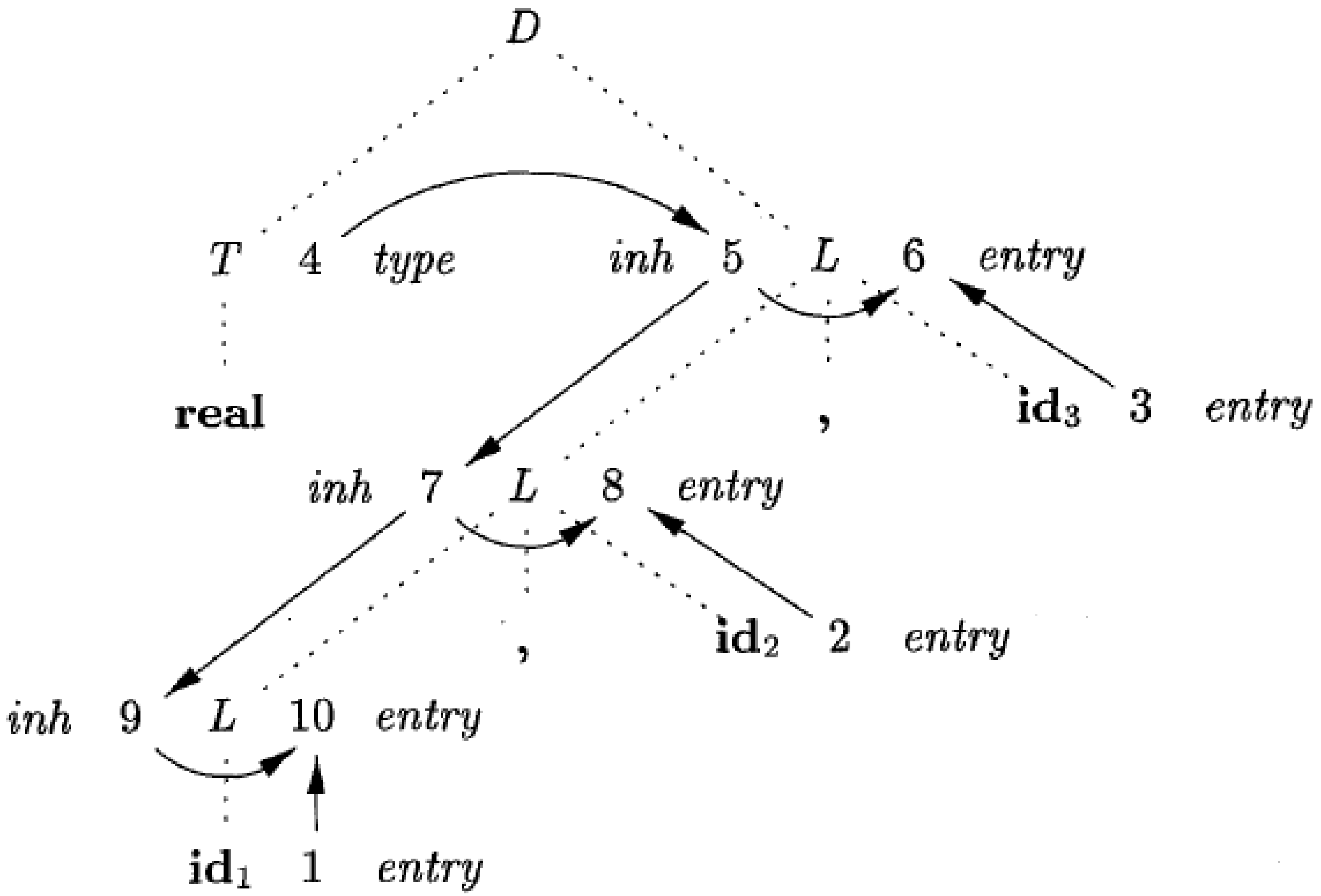
Production	Semantic rules
$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = float
$L \rightarrow L_1$ **, id**	L_1.inh = L.inh addType(**id**.entry, L.inh)
$L \rightarrow$ **id**	addType(**id**.entry, L.inh)

Semantic Rules with Side Effects

Declaration of Identifiers

mansoori@shirazu.ac.ir

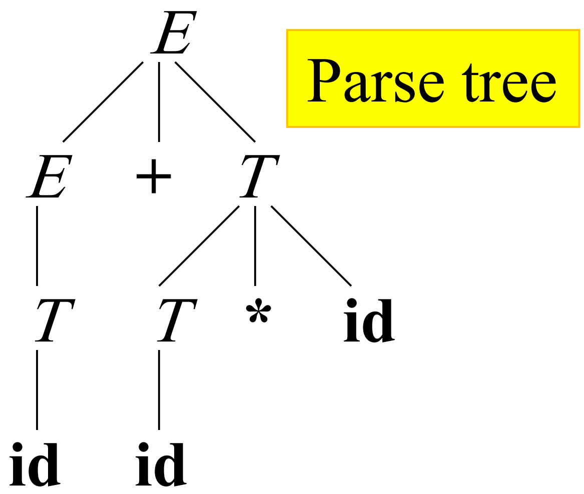
Dependency graph for declaration: **real id$_1$, id$_2$, id$_3$**

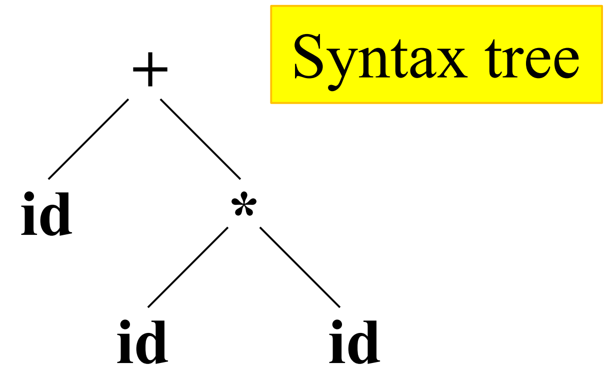


Application of SDD
Syntax Tree Construction

mansoori@shirazu.ac.ir

- Syntax tree is a convenient intermediate representation





- Syntax tree construction for expressions:
  1. S-attributed SDD (suitable for use during bottom-up parsing)
  2. L-attributed SDD (suitable for use during top-down parsing)

S-attributed SDD
Syntax Tree for Expressions

mansoori@shirazu.ac.ir

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

Steps for **a - 4 + c**

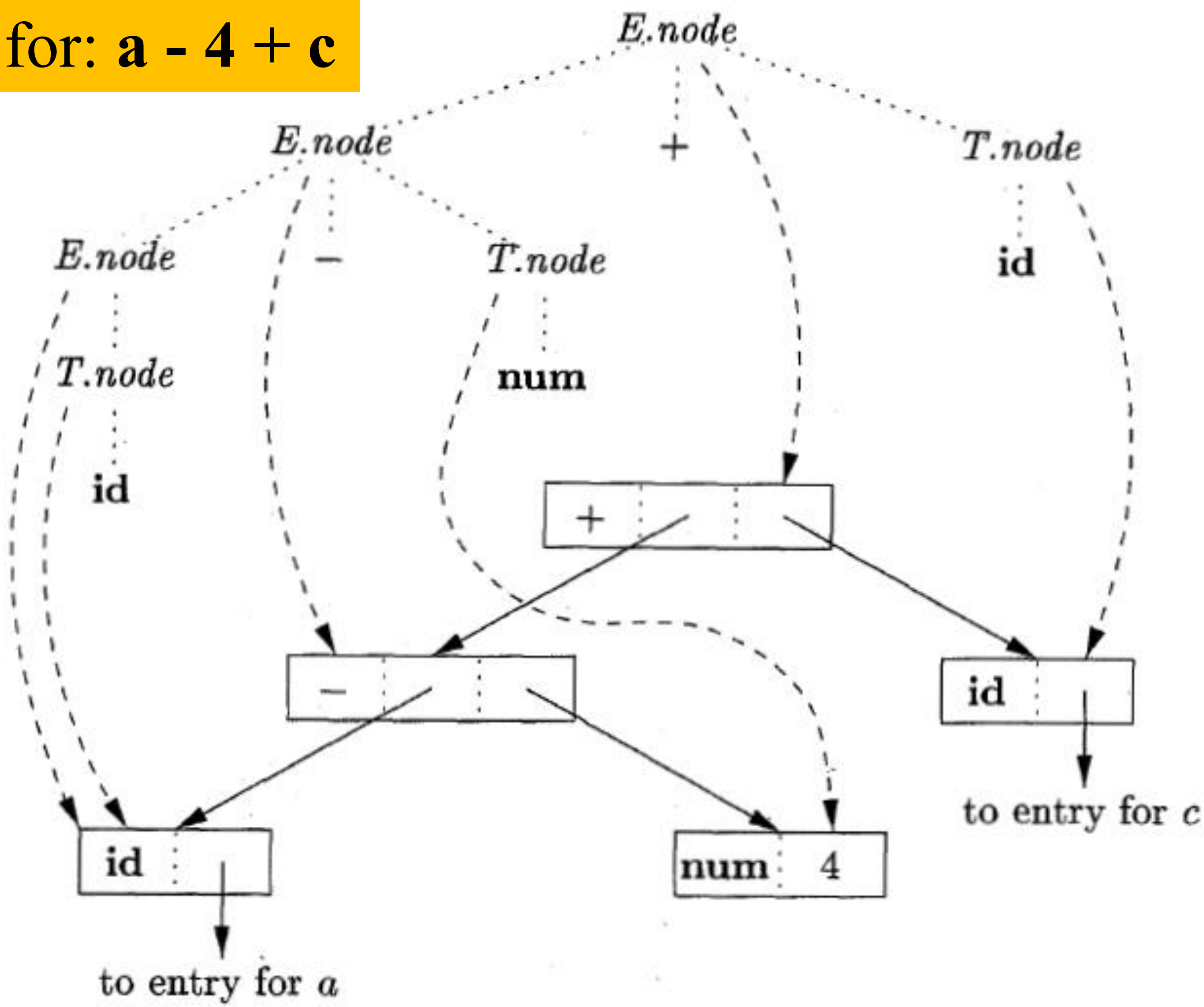
1) $p_1 = \textbf{new } Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new } Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new } Node('-', p_1, p_2);$
4) $p_4 = \textbf{new } Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new } Node('+', p_3, p_4);$

S-attributed SDD

Syntax Tree for Expressions

mansoori@shirazu.ac.ir

Syntax tree for: **a - 4 + c**



L-attributed SDD
Syntax Tree for Expressions

mansoori@shirazu.ac.ir

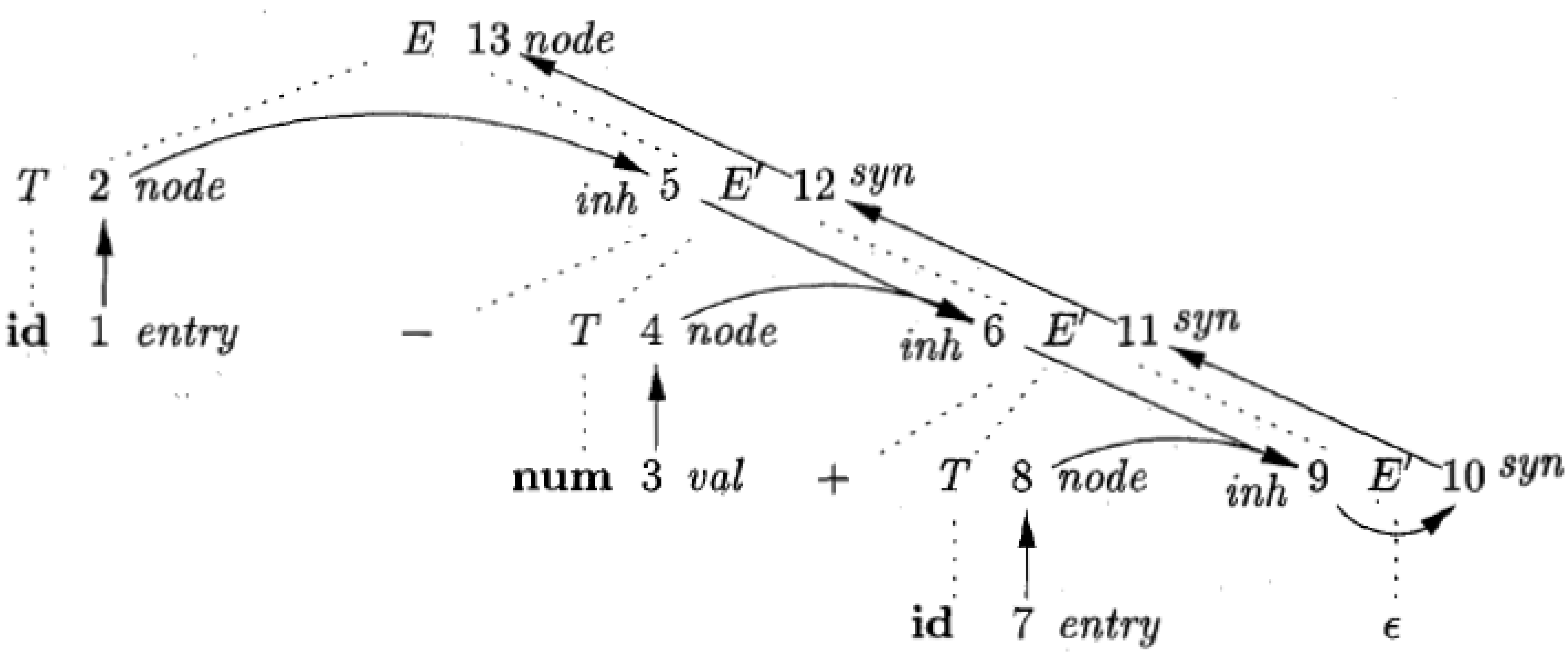
	Production	Semantic Rules
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E'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

L-attributed SDD

Syntax Tree for Expressions

mansoori@shirazu.ac.ir

Dependency graph for **a - 4 + c**



L-attributed SDD
Structure of Simple Types

mansoori@shirazu.ac.ir

- A mismatch in structure due to design of language

- A simple declaration of identifiers in C:

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

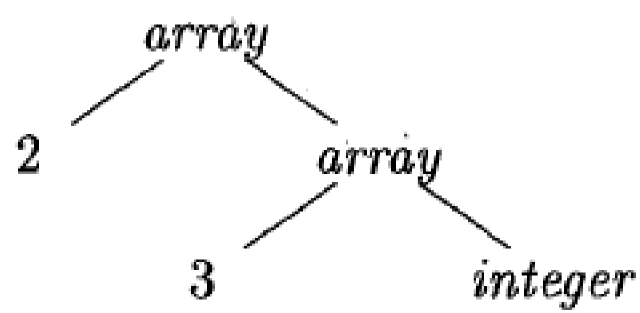
L-attributed SDD

Structure of Array Types

mansoori@shirazu.ac.ir

- In C:
  - **int** [2][3] ➔ array of 2 arrays of 3 integers
  - Type expression ➔ array(2, array(3, integer))

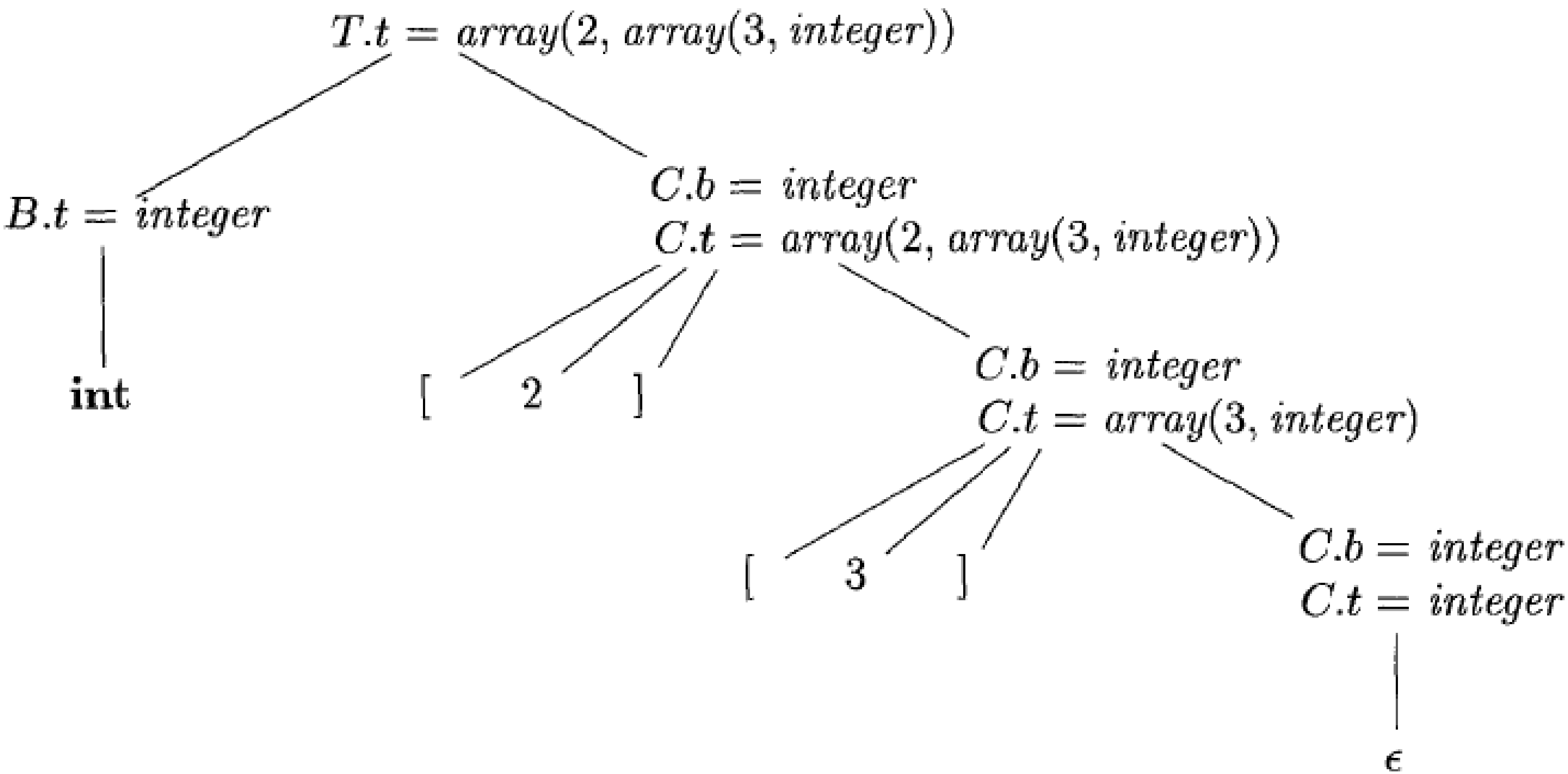
PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



L-attributed SDD

Structure of a Type

mansoori@shirazu.ac.ir



mansoori@shirazu.ac.ir

SDT

What is SDT?

mansoori@shirazu.ac.ir

- SDT is a complementary notation to SDD
- SDT is a CFG with program fragments {semantic actions} embedded within production bodies

$$A \rightarrow X \{actions\} Y$$

- Semantic actions can appear at any position within a production body
- Any SDT can be implemented by:
  - Building a parse tree, and then
  - Performing actions in a left-to-right depth-first postorder traversal

General Implementation of SDT

mansoori@shirazu.ac.ir

1. Ignoring the actions, parse the input and produce a parse tree as a result
2. Examine each interior node N (with production $A \rightarrow \alpha$) add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α
3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action

SDT Implementation during Parsing

mansoori@shirazu.ac.ir

- SDTs are implemented during parsing, without building a parse tree
- SDTs to implement two important classes of SDDs:
  - CFG is LR-parsable, SDD is S-attributed
  - CFG is LL-parsable, SDD is L-attributed
- How semantic rules in an SDD can be converted into an SDT with actions (executed at right time)
- During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched

SDT Implementation during Parsing

mansoori@shirazu.ac.ir

- An action is performed immediately after all symbols to its left are processed: $B \rightarrow X \{a\} Y$
  - Action a is done after X is recognized (X: terminal) or all the terminals derived from X are recognized (X: nonterminal)

- In bottom-up parsing, action a is performed as soon as this occurrence of X appears on top of parsing stack

- In top-down parsing, action a is performed just before expanding this occurrence of Y (Y: nonterminal) or checking for Y on input (Y: terminal)

Turning S-attributed SDD into SDT

mansoori@shirazu.ac.ir

- For a CFG which is bottom-up LR-parsable and its SDD is S-attributed, the SDT can be constructed by placing all actions at the right ends of the production bodies

- An SDT with all actions at the right ends of the production bodies is called postfix SDT

- The actions are executed when bottom-up LR parser reduces the production bodies to their heads

Turning S-attributed SDD into SDT

Simple Calculator

mansoori@shirazu.ac.ir

Production	Semantic rules
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

S-attributed SDD

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

S-attributed SDT

LR-parsable

Turning L-attributed SDD into SDT

mansoori@shirazu.ac.ir

- For a CFG which is top-down LL-parsable and its SDD is L-attributed, the SDT can be constructed by:

  1. Embed the action that computes inherited attributes for nonterminal A immediately before that occurrence of A in production's body

  2. Place the actions, that compute a synthesized attribute for head of production, at end of production's body

- The actions are executed when top-down LL parser reaches them in productions' bodies

Turning L-attributed SDD into SDT

Declaration of Identifiers

mansoori@shirazu.ac.ir

L-attributed SDD

Production	Semantic rules
$D \rightarrow T\ L$	$L.i = T.type$
$T \rightarrow$ **int**	$T.type = integer$
$T \rightarrow$ **float**	$T.type = real$
$L \rightarrow L_1$ **, id**	$L_1.i = L.i$ addType(**id**.entry, L.i)
$L \rightarrow$ **id**	addType(**id**.entry, L.i)

L-attributed SDT

$D \rightarrow T$ { L.i = T.type } L

$T \rightarrow$ **int** { T.type = integer }

$T \rightarrow$ **float** { T.type = float }

$L \rightarrow$ { L_1.i = L.i } L_1 **, id** { addType(**id**.entry, L.i) }

$L \rightarrow$ **id** { addType(**id**.entry, L.i) }

LL-parsable?

CFG is left-recursive

Turning L-attributed SDD into SDT
Intermediate Code Generation

mansoori@shirazu.ac.ir

Production: $S \rightarrow$ **while (C)** S_1

Inherited attributes

- S.next : labels the beginning of code must be executed after S
- C.true : labels the beginning of code must be executed if C is true
- C.false : labels the beginning of code must be executed if C is false

Synthesized attributes

- S.code : sequence of intermediate codes generated for S and ends with a jump to S.next
- C.code : sequence of intermediate codes generated for C and ends with a jump to either C.true or C.false (whether C is true or false)

Turning L-attributed SDD into SDT

Intermediate Code Generation

mansoori@shirazu.ac.ir

Semantic rules	Production
L1 = **new** Label(); L2 = **new** Label(); C.false = S.next; C.true = L2; S_1.next = L1; S.code = **label** || L1 || C.code || **label** || L2 || S_1.code;	$S \rightarrow$ **while (** C **)** S_1

⇩

S → **while (** { L1 = **new** Label(); L2 = **new** Label();
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; }

LL-parsable

Implementing L-attributed SDD/SDT

mansoori@shirazu.ac.ir

- Translation methods by traversing parse tree
  - SDD: Build the parse tree and annotate it
  - SDT: Build the parse tree, add actions, and execute the actions in preorder

- Translation methods during parsing
  1. Use a recursive-descent parser with one function for each nonterminal
  2. Generate code on the fly, using a recursive-descent parser

Implementing L-Attributed SDT
Translation During Recursive-Descent Parsing

mansoor @shirazu.ac.ir

- Recursive-descent parser has a function A() for each nonterminal *A*:
  - Inherited attributes of *A*: arguments of A()
  - Synthesized attributes of *A*: return-values of A()
- In body of A(), do parsing and handling attributes
  1. Decide upon the true production of *A* to expand
  2. Match the terminals that appear in input
  3. Preserve, in local variables, all computed attributes
     - Inherited attributes of nonterminals in the body
     - Synthesized attributes for *A*
  4. Call functions of nonterminals, in the body, with proper arguments

Implementing L-Attributed SDT

Translation During Recursive-Descent Parsing

mansoori@shirazu.ac.ir

$S \rightarrow$ **while (** { L1 = **new** Label(); L2 = **new** Label(); 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 ; }

```
string S(label Snext) {
  if( lookahead == while ) {
     match(while);  match('(');
     label L1 = new Label();  label L2 = new Label();
     label Cfalse = Snext;  label Ctrue = L2;
     string Ccode = C(Cfalse,Ctrue);
     match(')');
     label S1next = L1;
     string S1code = S(S1next);
     string Scode = Concat("label",L1,Ccode,"label",
                                      L2,S1code);
  }
  else /* other statements */
  return Scode;
}
```