Compiler Design

Preet Kanwal
Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 3: Syntax Directed Definitions

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- Dependency graphs
- Topological Sort
- Evaluating SDDs - Example - Simple Desk Calculator
- S–Attributed SDD Examples :
  - Counting Binary bits
  - Binary to Decimal conversion
  - Counting parentheses
  - Identifying the type of an expression - int or float
  - Identifying the sign of the evaluated expression - positive or negative

Recap - Evaluating SDDs - Parse tree method

Evaluating an SDD over a given input consists of the following steps -

1. Construct Parse Tree for given input.
2. Construct Dependency graph.
3. Topologically sort the nodes of Dependency graph.
4. Produce as output Annotated Parse Tree.

Dependency Graphs

- Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.
- A Dependency Graph 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).
- Dependency Rule - If an attribute b depends from an attribute c, then we need to fire the semantic rule for c first and then the semantic rule for b.

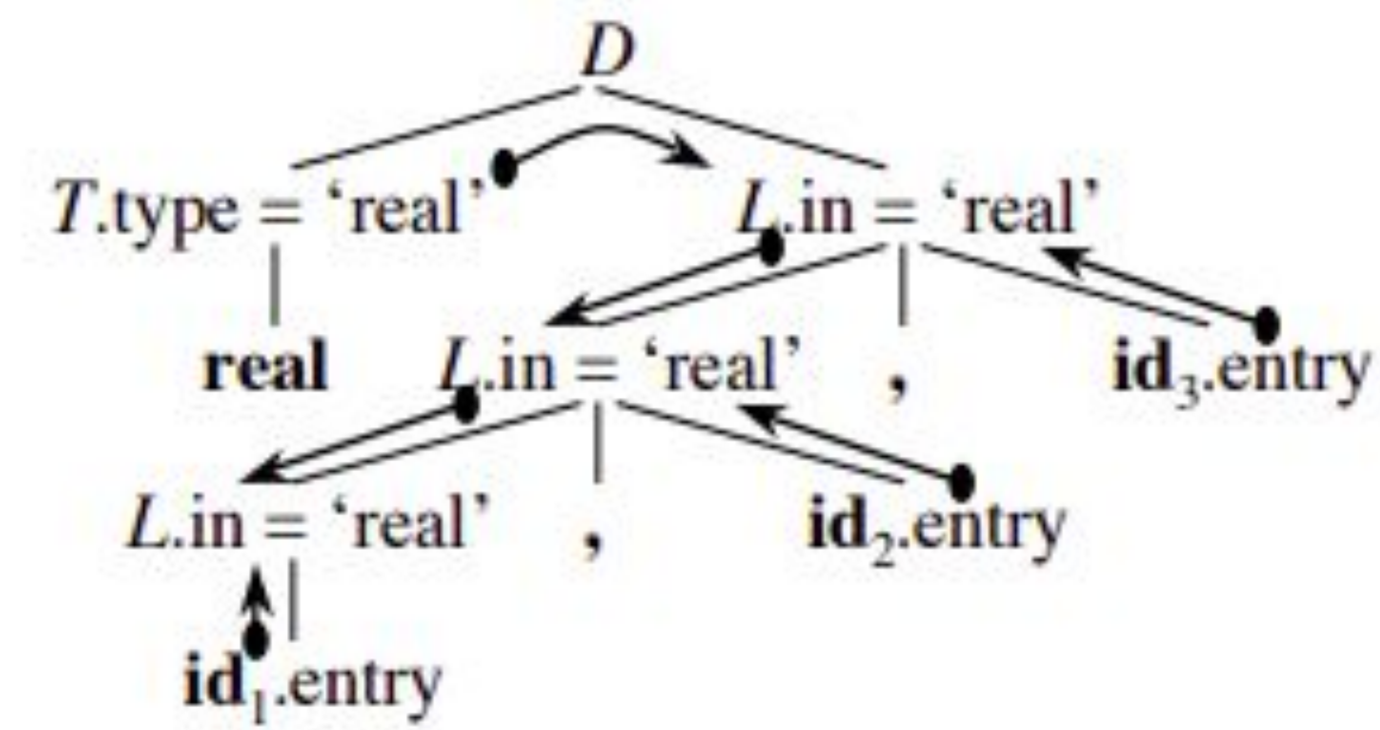
Dependency Graphs - Example

- Consider the earlier example of Syntax Directed Definition for Type Declarations.
- This SDD has both inherited and synthesised attributes.
- Construct the Annotated parse tree with dependency graph for the input **real id1, id2, id3**

Production	Semantic Rule
D $\rightarrow$ T L	*{ L.in = T.type;}*
T $\rightarrow$ int	*{ T.type = integer; }*
T $\rightarrow$ real	*{ T.type = float; }*
L $\rightarrow$ L_1 , id	*{ L_1.in= L.in;* *addType(id.entry, L.in); }*
L $\rightarrow$ id	*{ addType(id.entry, L.in); }*

Dependency Graphs - Example

- **The figure illustrates the annotated parse tree with the dependencies between attributes.**



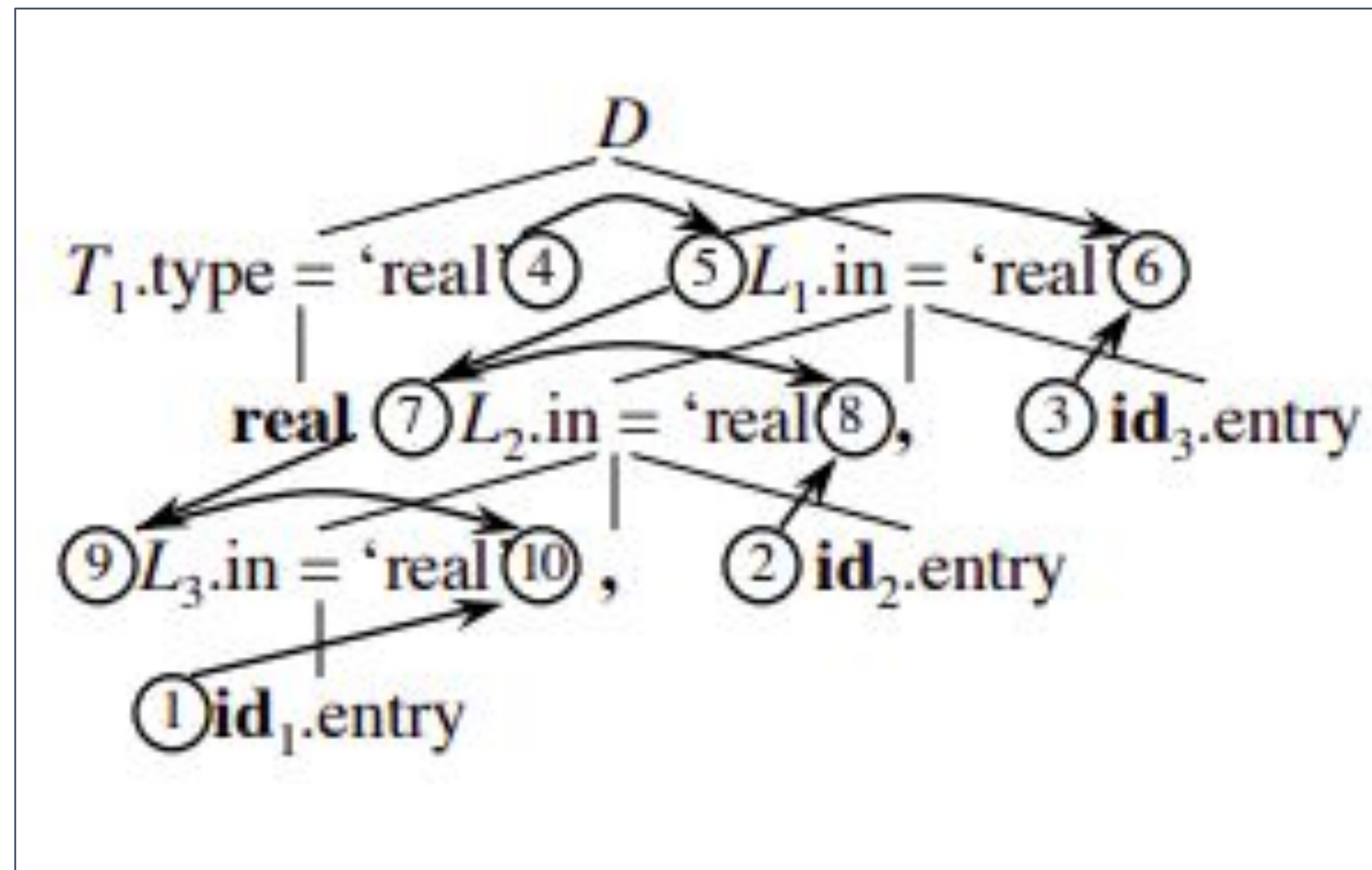
Topological Sort

- The evaluation order of semantic rules depends from a Topological Sort derived from the dependency graph.
- Topological Sort - Any ordering $m_1, m_2, \ldots, m_k$ such that if $m_i \rightarrow m_j$ is a link in the dependency graph then $m_i < m_j$.
- Any topological sort of a dependency graph gives a valid order to evaluate the semantic rules.

Topological Sort - Example

- Consider the previous example of SDD for Type declarations
- The topological sort of the parse tree is as follows -

1. Get id1.entry
2. Get id2.entry
3. Get id3.entry
4. T_1.type='real'
5. L_1.in=T_1.type
6. addtype(id3.entry, L_1.in)
7. L_2.in=L_1.in
8. addtype(id2.entry, L_2.in)
9. L_3.in=L_2.in
10. addtype(id1.entry, L_3.in)



Evaluating SDDs - General Remarks

- Attributes can be evaluated by building a dependency graph at compile-time and then finding a topological sort.
- Disadvantages
    1. This method fails if the dependency graph has a cycle - We need a test for non-circularity.
    2. This method is time consuming due to the construction of the dependency graph.
- Alternative Approach
    - Design the syntax directed definition in such a way that attributes can be evaluated with a fixed order avoiding the need to build the dependency graph - for example, S attributed definitions.
    - This method is followed by many compilers.

Evaluating S-Attributed Definitions

- Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.
- The parser keeps the values of the synthesized attributes in its stack.
- Whenever a reduction $A \longrightarrow \alpha$ 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 simply implemented by extending the stack of an LR-Parser.

Example - Evaluating SDD to implement a Simple Desk Calculator

- Evaluate the following SDD for the input 3 + 4 * 5

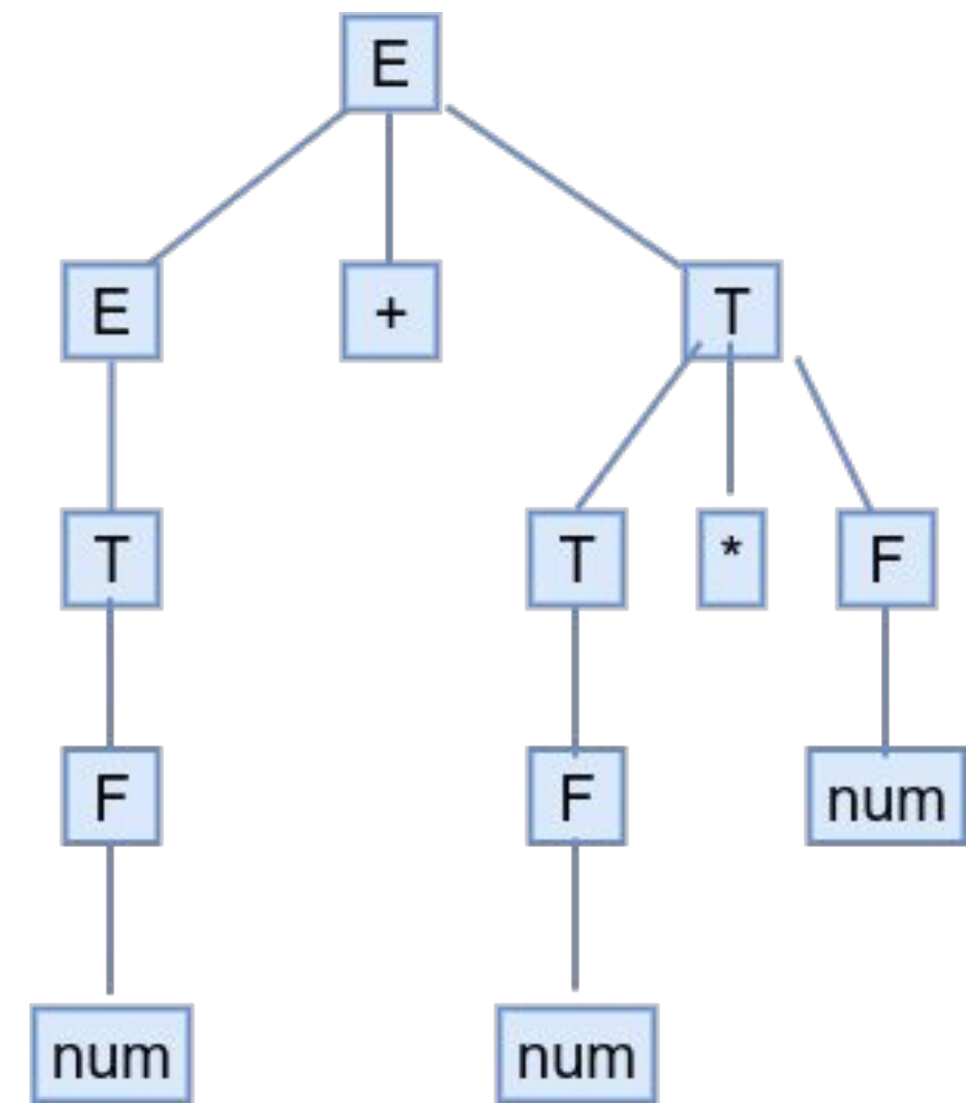
Production	Semantic Rule
$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val \}$
$E \rightarrow T$	$\{ E.val = T.val \}$
$T \rightarrow T_1 * F$	$\{ T.val = T_1.val * F.val \}$
$T \rightarrow F$	$\{ T.val = F.val \}$
$F \rightarrow num$	$\{ F.val = num.lexval \}$

Example - Evaluating SDD to implement a Simple Desk Calculator

Evaluate the following SDD for the input 3 + 4 * 5

Solution -

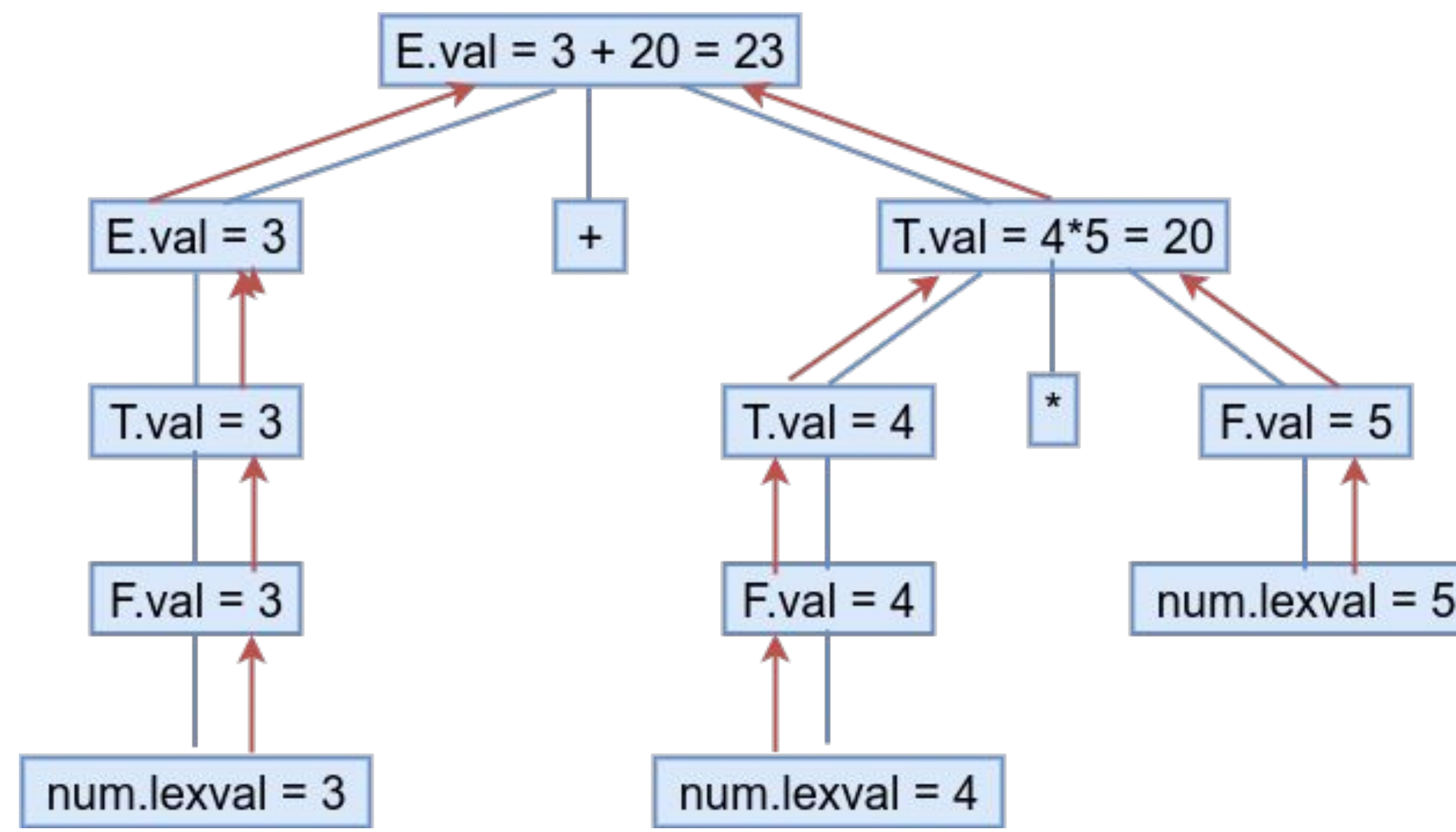
1. Construct Parse tree
   - Nodes are Terminals or Non-Terminals



Example - Evaluating SDD to implement a Simple Desk Calculator

Evaluate the following SDD for the input 3 + 4 * 5

Solution -



2. Construct dependency graph

- **Nodes are attributes**

Example - Evaluating SDD to implement a Simple Desk Calculator

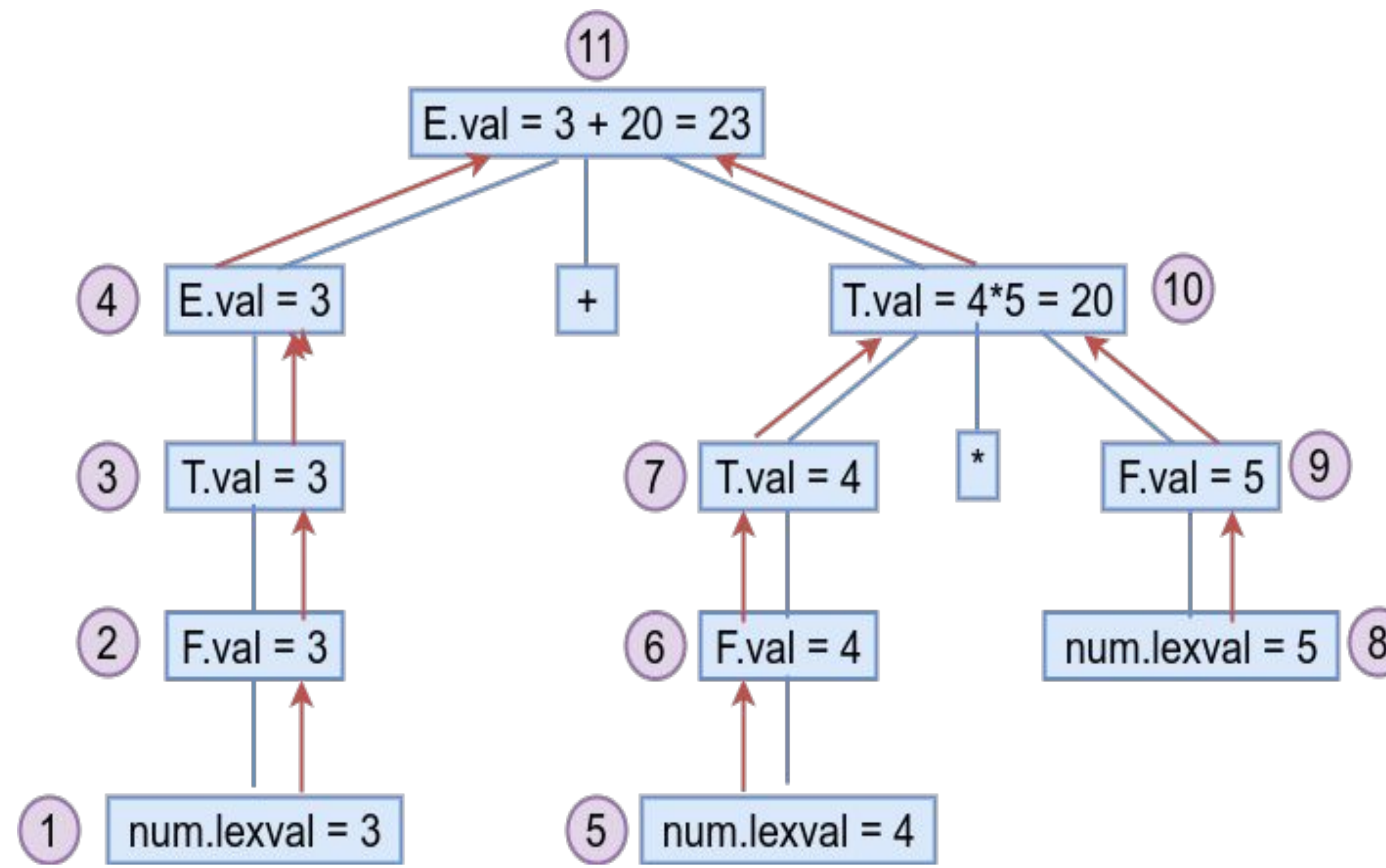
Evaluate the following SDD for the input **3 + 4 * 5**

Solution -

3. Decide Evaluation order of the nodes in Dependency graph

1,5,8,2,3,6,7,8,9,4,10,11

Output -Annotated Parse tree carrying out the input.



S-attributed SDDs - Examples

Example 1 - SDD to count no. of 1s in a Binary Number

Production	Semantic Rule
$L \rightarrow L_1\ B$	{ $L.count = L_1.count + B.count;$ }
$L \rightarrow B$	{ $L.count = B.count;$ }
$B \rightarrow 0$	{ $B.count = 0;$ }
$B \rightarrow 1$	{ $B.count = 1;$ }

Example 2 - SDD to count no. of 0s in a Binary Number

Production	Semantic Rule
$L \rightarrow L_1 B$	{ $L.count = L_1.count + B.count;$ }
$L \rightarrow B$	{ $L.count = B.count;$ }
$B \rightarrow 0$	{ $B.count = 1;$ }
$B \rightarrow 1$	{ $B.count = 0;$ }

Example 3 - SDD to count no. of bits in a Binary Number

Production	Semantic Rule
$L \rightarrow L_1\ B$	{ $L.count = L_1.count + B.count;$ }
$L \rightarrow B$	{ $L.count = B.count;$ }
$B \rightarrow 0$	{ $B.count = 1;$ }
$B \rightarrow 1$	{ $B.count = 1;$ }

Example 4 - SDD to convert Binary to Decimal

Production	Semantic Rule
$L \rightarrow L_1\ B$	{ $L.val = 2 * L_1.val + B.val;$ }
$L \rightarrow B$	{ $L.val = B.val;$ }
$B \rightarrow 0$	{ $B.val = 0;$ }
$B \rightarrow 1$	{ $B.val = 1;$ }

Example 5 - SDD to convert Binary Fraction to Decimal

Production	Semantic Rule
$S \rightarrow L_1 . L_2$	{ $S.val = L_1.val + L_2.val / 2^{L2.count}$; }
$L \rightarrow L_1 B$	{ $L.val = 2 * L_1.val + B.val;$ $L.count = L_1.count + B.count;$}
$L \rightarrow B$	{ $L.val = B.val;$ $L.count = B.count;$ }
$B \rightarrow 0$	{ $B.val = 0;$ $B.count = 1;$ }
$B \rightarrow 1$	{ $B.val = 1;$ $B.count = 1;$}

Example 6 - SDD to count no. of balanced Parentheses

Production	Semantic Rule
$S \rightarrow (S_1)$	*{ S.count = S_1.count + 1; }*
$S \rightarrow a$	*{ S.count = 0; }*

Example 7- SDD to convert infix to postfix

Production	Semantic Rule
$E \rightarrow E_1 + T$	*{ printf("+"); }*
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*{ printf("+"); }*
$T \rightarrow F$	
$F \rightarrow num$	*{ printf("%d",num.lexval); }*

Example 8 - SDD to determine type of each Term and Expression

The given grammar is for expressions involving operator **+** and integer or floating-point operands.
Floating-point numbers are distinguished by having a decimal point.

E -> E + T | T
T -> num . num | num

Write an SDD to determine the type of each term **T** and expression.

Example 8 - SDD to determine type of each Term and Expression

Production	Semantic Rule
$E \rightarrow E_1 + T$	*{ if(E_1.type == float || T.type == float) { E.type = float; } else { E.type = integer; } }*
$E \rightarrow T$	*{ E.type = T.type }*
$T \rightarrow$ num . num	*{ T.type = float ; }*
$T \rightarrow$ num	*{ T.type = integer ; }*

Example 9 - SDD to Identify the sign of the evaluated expression

- Given - an attribute grammar G for arithmetic expressions using multiplication, unary -, and unary +.
- Complete the SDD -
  - Add semantic actions to compute an attribute E.sign for non-terminal E to record whether the arithmetic value of E is positive or negative.
  - The attribute sign can have two values,either POS or NEG.
- Also, show the parse tree for input 2 * - 3

Production	Semantic Rule
$S \rightarrow E$	*{ if(E.sign == POS)* *print("Result is positive");* *else* *print("Result is negative"); }*
$E \rightarrow$ num	
$E \rightarrow + E_1$	
$E \rightarrow - E_1$	
$E \rightarrow E_1 * E_2$	

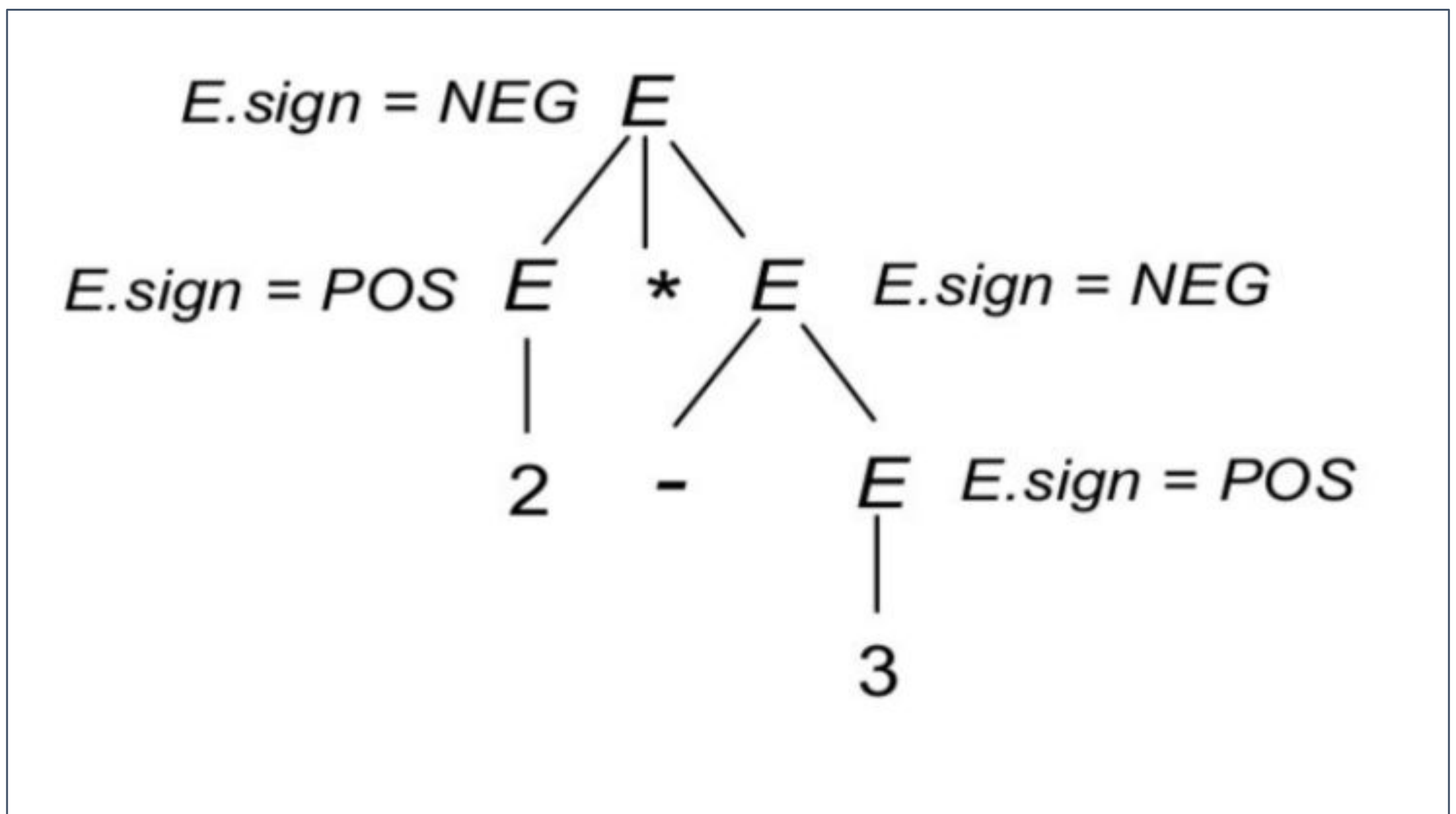
Example 9 - SDD to Identify the sign of the evaluated expression

Production	Semantic Rule
$S \longrightarrow E$	*{ if(E.sign == POS) print("Result is positive"); else print("Result is negative"); }*
$E \longrightarrow num$	*{ E.sign == POS; }*
$E \longrightarrow + E_1$	{ $E.sign = E_1.sign$}
$E \longrightarrow - E_1$	*{ if (E.sign == POS) E.sign = NEG; else E.sign = POS; }*
$E \longrightarrow E_1 * E_2$	{ if ($E_1.sign == E_2.sign$ *E.sign = POS;* *else* *E.sign = NEG; }*

Example 9 - SDD to Identify the sign of the evaluated expression (cont.)

Show the parse tree for input 2 * - 3 (where 2 and 3 are "unsigned_ints"). Indicate at each node what the values of associated attributes are.

Solution -



THANK YOU

Preet Kanwal
Department of Computer Science & Engineering
preetkanwal@pes.edu

PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 3: L-Attributed SDD

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- What is an L-attributed SDD?
- L–Attributed SDD Examples
  - Simple Type declaration
  - Array type Variable Declaration
  - Variable declaration verification
  - Desk calculator

- **Syntax Directed Definitions (SDD)** are a generalization of context-free grammars in which -
  - Grammar symbols have an associated set of attributes
  - Productions are associated with Semantic Rules for computing the values of attributes.
- Two kinds of attributes
  - **Synthesized Attributes** - computed from the values of the attributes of the children nodes.
  - **Inherited Attributes** - computed from the values of the attributes of both the siblings and the parent nodes.
- An SDD with only synthesized attributes is called an **S-attributed definition**.

What is an L-attributed SDD?

A Syntax Directed Definition is L-attributed if all attributes are either -

1. Synthesized
2. Extended synthesized attributes, which can depend not only on attributes at the children, but on inherited attributes at the node itself
3. Inherited, but depending only on inherited attributes at the parent and any attributes at siblings to the left.

- The formal definition of an L-Attributed SDD is as follows -

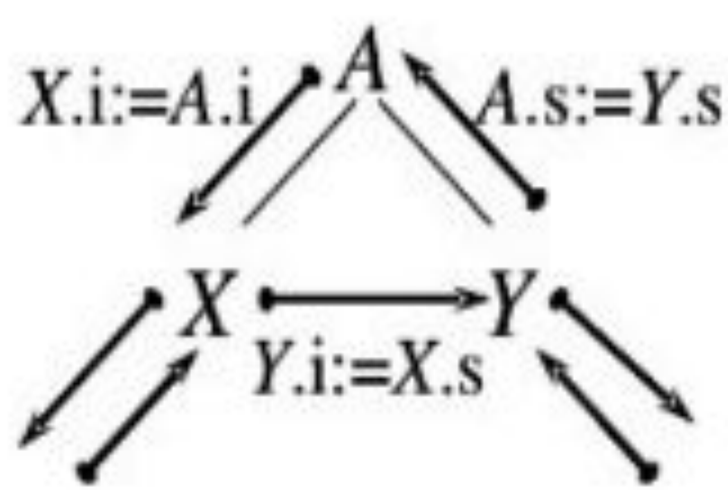
  A syntax directed definition is L-Attributed if each inherited attribute of X_j in a production $A \rightarrow X_1 \ldots X_j \ldots X_n$, depends only on -

  1. The attributes of the symbols to the left (this is what L in L-Attributed stands for) of X_j , i.e., $X_1\ X_2 \ldots X_{j-1}$
  2. The inherited attributes of A.

- Theorem - Inherited attributes in L-Attributed Definitions can be computed by a PreOrder traversal of the parse-tree.

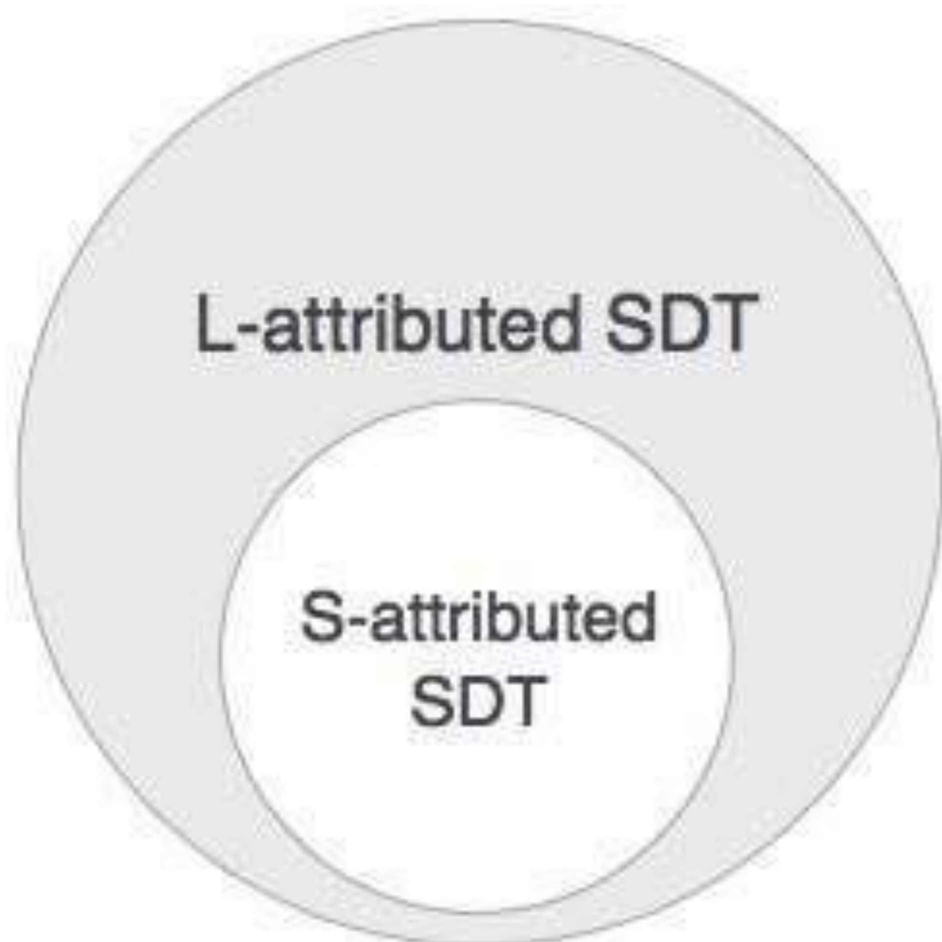
- L-attributed definitions allow for the natural order of evaluating attributes, i.e, depth first, left to right.
- For example -

  A → X Y



$X.i := A.i$
$Y.i := X.s$
$A.s := Y.s$

Every S-attributed Syntax-Directed Definition is also L-attributed.



L-Attributed SDD to implement a Simple Desk Calculator

Complete the semantic rules for the following L-attributed SDD.

Evaluate the SDD for the input 3 + 5

Production	Semantic Rule
$E \rightarrow T\ E'$	
$E' \rightarrow +\ T\ E'_1$	
$E \rightarrow \lambda$	
$T \rightarrow F\ T'$	
$T \rightarrow *\ F\ T'_1$	
$T' \rightarrow \lambda$	
$F \rightarrow \text{num}$	

L-Attributed SDD to implement a Simple Desk Calculator

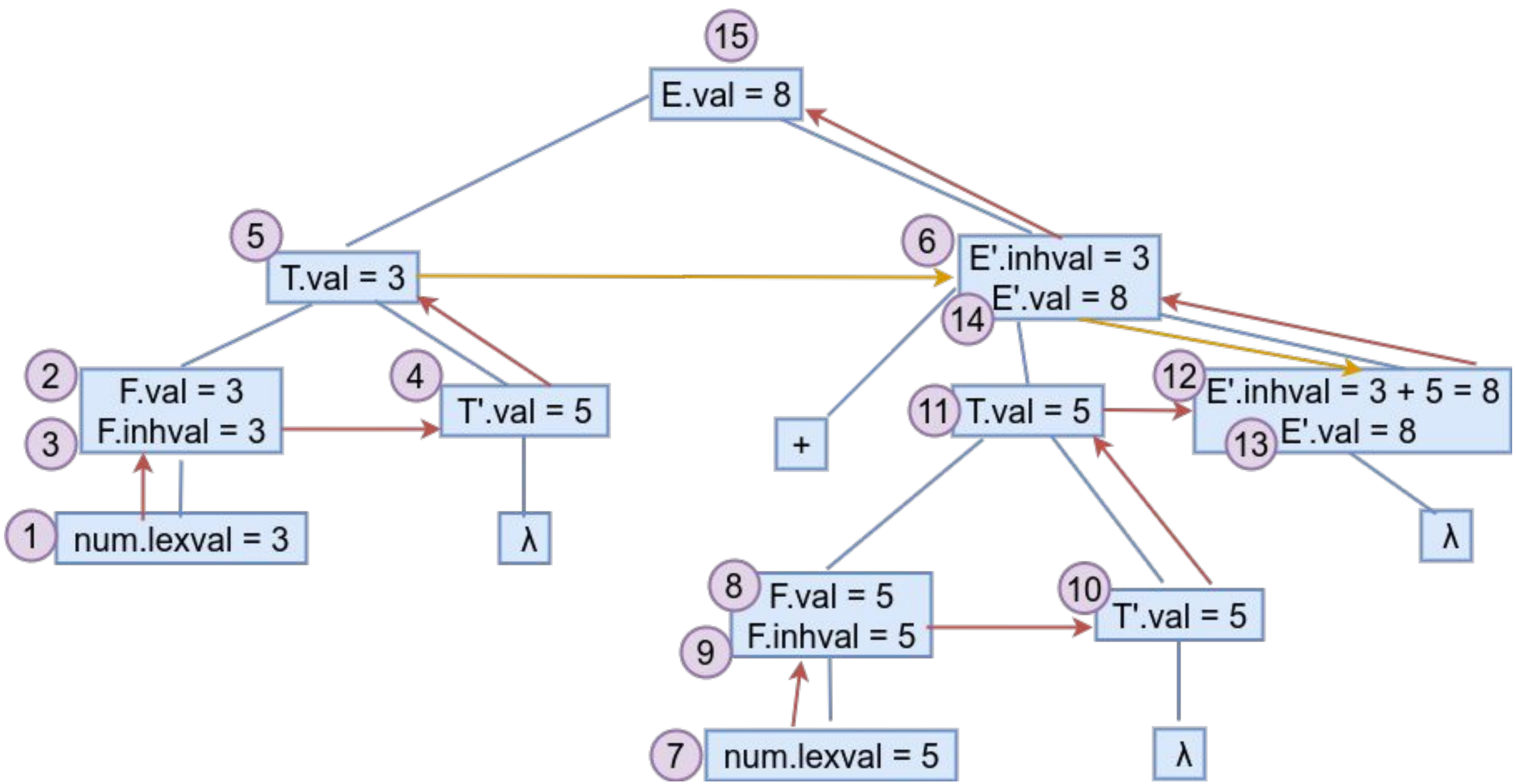
Production	Semantic Rule
$E \rightarrow T\ E'$	{ $E'.inhval = T.val;$ $E.val = E'.val;$ }
$E' \rightarrow +\ T\ E'_1$	{ $E'_1.inhval = E'.inhval + T.val;$ $E'.val = E'_1.val;$ }
$E \rightarrow \lambda$	{ $E'.val = E'_1.inhval;$ }
$T \rightarrow F\ T'$	{ $T'.inhval = F.val;$ $T.val = T'.val;$ }
$T \rightarrow *\ F\ T'_1$	{ $T'_1.inhval = T'.inhval + F.val;$ $T'.val = T'_1.val;$ }
$T' \rightarrow \lambda$	{ $T'.val = T'.inhval$ }
$F \rightarrow num$	{ $F.val = num.lexval$ }

This is an LDD -

In $E \rightarrow T\ E'$, the attribute for E' is inherited from T, which is the left-sibling.

L-Attributed SDD to implement a Simple Desk Calculator

Evaluate the SDD for the input 3 + 5



L-attributed SDD for Simple Type declaration

Complete the semantic rules for the following L-attributed SDD.

Evaluate the SDD for the input int a,b

Production	Semantic Rule
D $\rightarrow$ T L	
T $\rightarrow$ int	
T $\rightarrow$ float	
L $\rightarrow$ L_1 , id	
L $\rightarrow$ id	

L-attributed SDD for Simple Type declaration

Production	Semantic Rule
$D \rightarrow T\ L$	{ $L.inhType = T.type;$ $L.inhWidth = T.width;$ }
$T \rightarrow$ int	{ $T.type = integer;$ $T.width = 4;$ }
$T \rightarrow$ float	{ $T.type = float;$ $T.width = 8;$ }
$L \rightarrow L_1$, id	{ $L_1.inhType = L.inhType;$ $L_1.inhWidth = L.inhWidth;$ update(id.entry, L.inhType,L.inhWidth); }
$L \rightarrow$ id	{update(id.entry, L.inhType,L.inhWidth); }

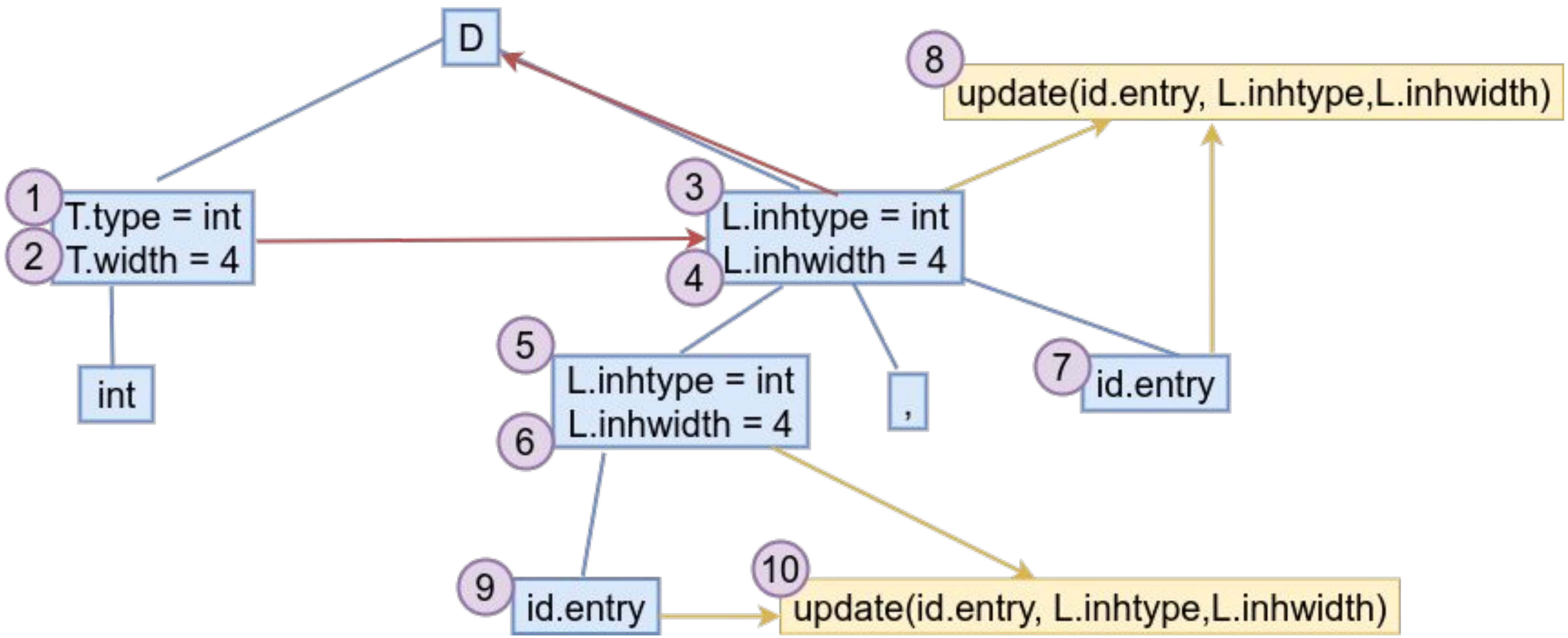
This is an LDD -

In $D \rightarrow T\ L$, the attribute for L is inherited from T, which is the left-sibling.

update() is used to update type and storage in the symbol table.

L-attributed SDD for Simple Type declaration

Evaluate the SDD for the input **int a,b;**



L-attributed SDD to identify Array Type

Complete the semantic rules for the following L-attributed SDD.

Evaluate the SDD for the input int [2][3]

Production	Semantic Rule
$T \rightarrow B\ C$	
$B \rightarrow$ int	
$B \rightarrow$ float	
$C \rightarrow$ [num] C_1	
$C \rightarrow \lambda$	

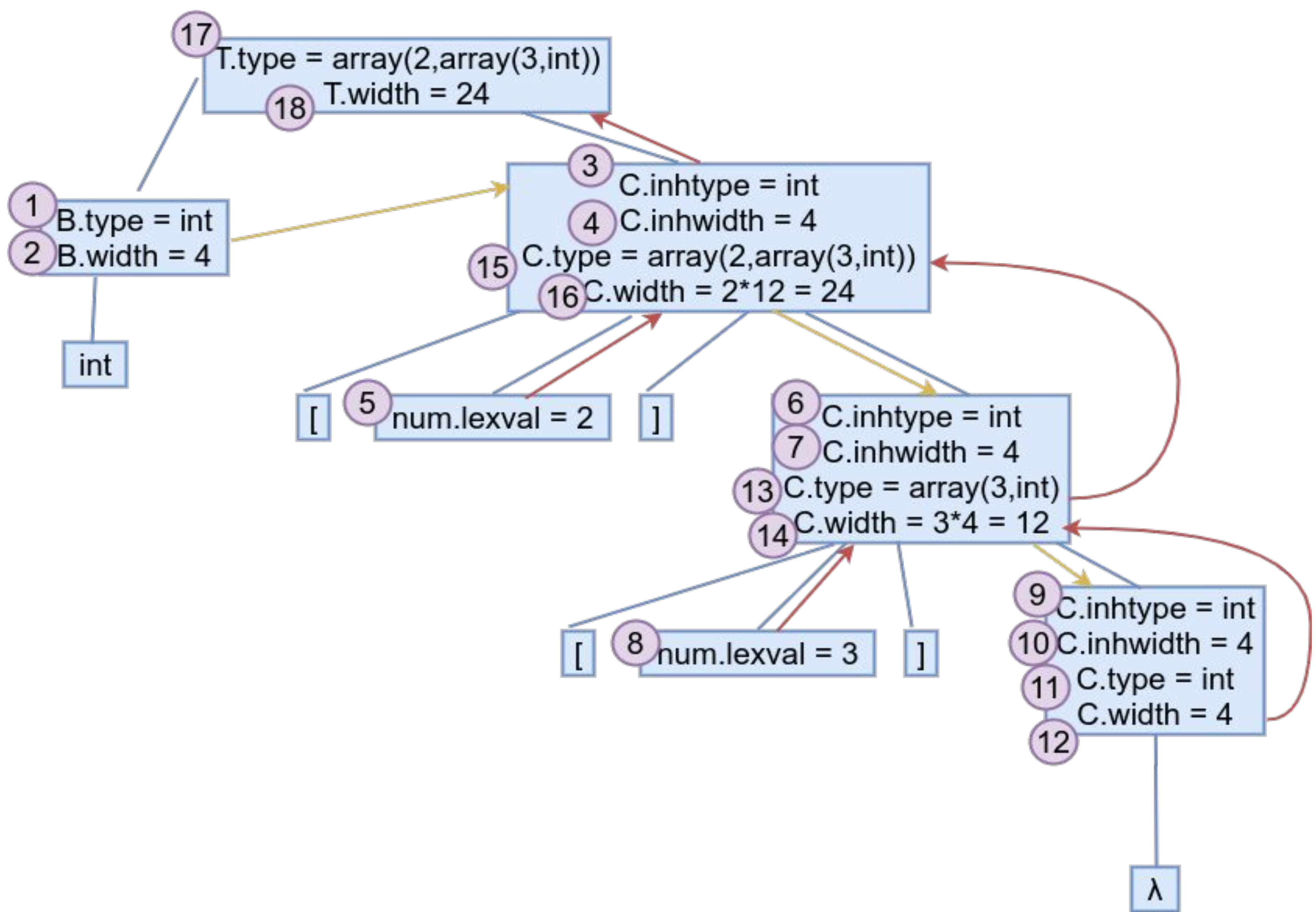
L-attributed SDD to identify Array Type

Production	Semantic Rule
T → B C	*{ C.inhType = B.type; C.inhWidth = B.width;* *T.type = C.type; T.width = C.width; }*
B → int	*{ B.type = integer; B.width = 4;}*
B→ float	*{ B.type = float; B.width = 8; }*
C → [num] C_1	*{ C_1.inhType = C.inhType; C_1.inhWidth = C.inhWidth;* *C.type = array(num.lexval , C_1.type);* *C.width = num.lexval * C_1.width ;);* *addType(id.entry, L.inhType);* *addWidth(id.entry, L.inhWidth; }*
C → λ	*{ C.type = C.inhType; C.width = C.inhWidth; }*

L-attributed SDD for Simple Type declaration

Evaluate the SDD for the input int a[2][3]

int a[2][3] translates to array(2,array(3,int))



L-attributed SDD for Basic vs Array Type declaration in C Semantics

Construct an SDD to identify an array of the following format -

float[4] x, y

Follow C Semantics - i.e, x is an array type, y is a basic type

Production	Semantic Rule
D→ T L	
T → B C	
B → int	
B→ float	
C → [num] C_1	
C → λ	
L → L_1 , id	
L → id	

L-attributed SDD for Basic vs Array Type declaration in C Semantics

Production	Semantic Rule
D→ T L	*{ L.inhType = T.type; L.inhWidth = T.width; L.inhbasicType = T.basicType; L.inhbasicWidth = T.basicWidth; }*
T → B C	*{ C.inhType = B.type; C.inhWidth = B.width; T.type = C.type; T.width = C.width; T.basicType = B.type; T.basicWidth = B.width;}*
B → int	*{ B.type = integer; B.width = 4;}*
B→ float	*{ B.type = float; B.width = 8; }*
C → [num] C_1	*{* C_1*.inhType = C.inhType;* C_1*.inhWidth = C.inhWidth; C.type = array(num.lexval ,* C_1*.type); C.width = num.lexval ** C_1*.width; }* Should be C
C → λ	*{* C_1*.inhType = C.inhType;* C_1*.inhWidth = C.inhWidth; }*

L-attributed SDD for Basic vs Array Type declaration in C Semantics

Production	Semantic Rule
…	…
$L \longrightarrow L_1$, id	*L_1.inhType = L.inhType;* *L_1.inhWidth = L.inhWidth;* *L_1.inhbasicType = L.inhbasicType;* *L_1.inhbasicWidth = L.inhbasicWidth;* *addType(id.entry, L.inhbasicType);* *addWidth(id.entry, L.inhbasicWidth;*
$L \longrightarrow$ id	*{ addType(id.entry, L.inhType);* *addWidth(id.entry, L.inhWidth; }*

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 3: Syntax Directed Definitions

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

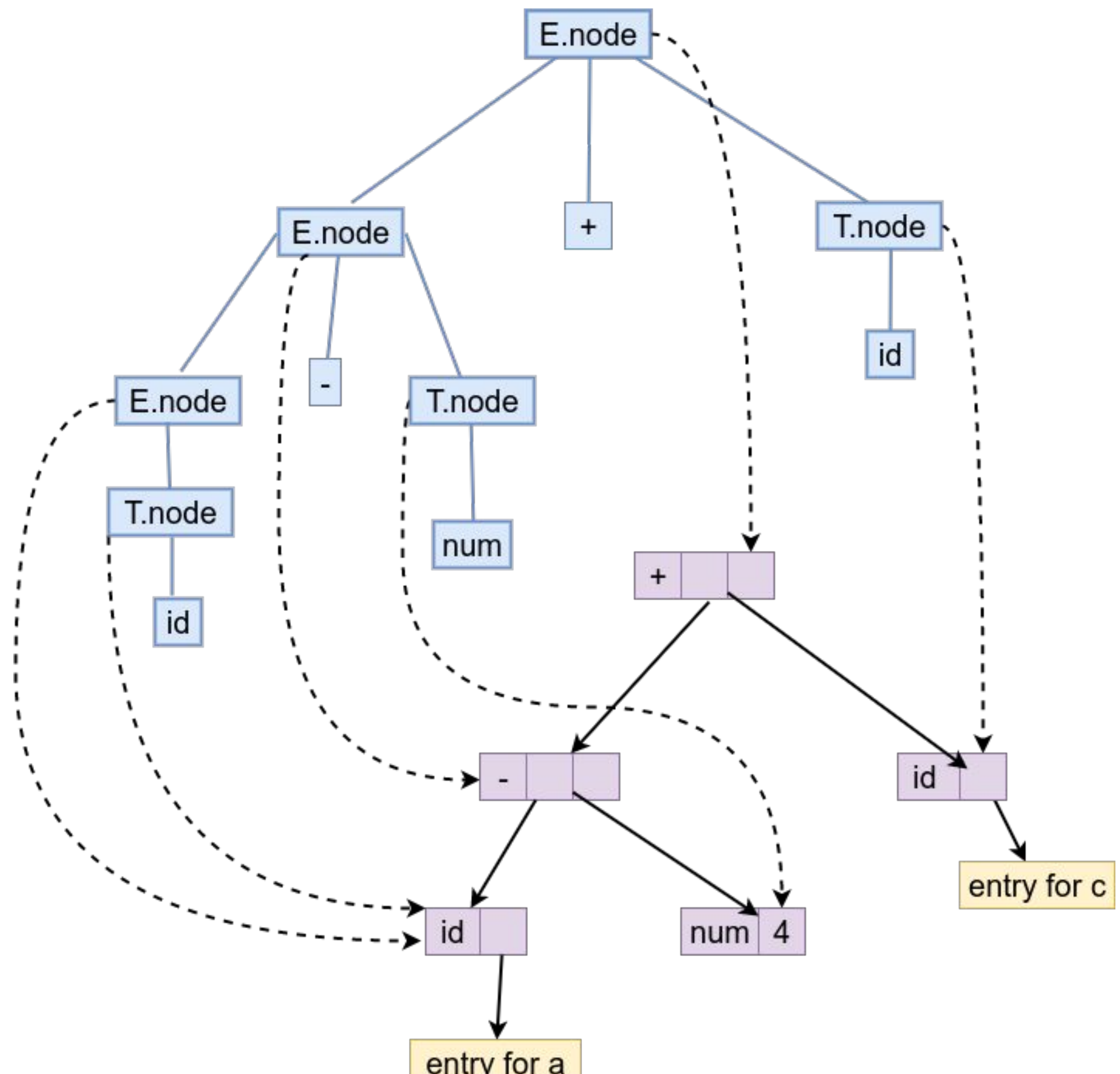
- S–Attributed SDD Examples :
  - To generate Syntax tree for Expressions
  - To generate Syntax tree for Statements

Example 1 - SDD to generate Syntax tree for Expressions

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 -> T	*{ E.node = T.node ; }*
T -> (E)	*{ T.node = E.node ; }*
T -> id	*{ T.node = new Leaf(id , id.entry); }*
T -> num	*{ T.node = new Leaf(num , num.lexval); }*

Example 1 - SDD to generate Syntax tree for Expressions

Use the previous grammar to construct the syntax tree for the input a - 4 + c



Example 2 - SDD to generate Syntax tree for Statements

Production	Semantic Rule
Stmt -> S Stmt	*{ Stmt.node = new Node(Seq, S.node, Stmt.node); }*
Stmt -> S	*{ Stmt.node = S.node; }*
S -> if (cond) { Stmt }	*{ S.node = new Node(if, Cond.node, Stmt.node); }*
S -> while (cond) { Stmt }	*{ S.node = new Node(while, Cond.node, Stmt.node); }*
S -> AssignExpr	*{ S.node = AssignExpr.node ; }*
Cond -> E_1 > E_2	*{ Cond.node = new Node(>, E_1.node, E_2.node); }*
Cond -> E_1 < E_2	*{ Cond.node = new Node(<, E_1.node, E_2.node); }*
Cond -> E_1 || E_2	*{ Cond.node = new Node(||, E_1.node, E_2.node); }*
Cond -> E_1 && E_2	*{ Cond.node = new Node(&&, E_1.node, E_2.node); }*
AssignExpr -> id = E;	*{ AssignExpr.node = new Node(=, new Leaf(id,id.entry), E.node); }*

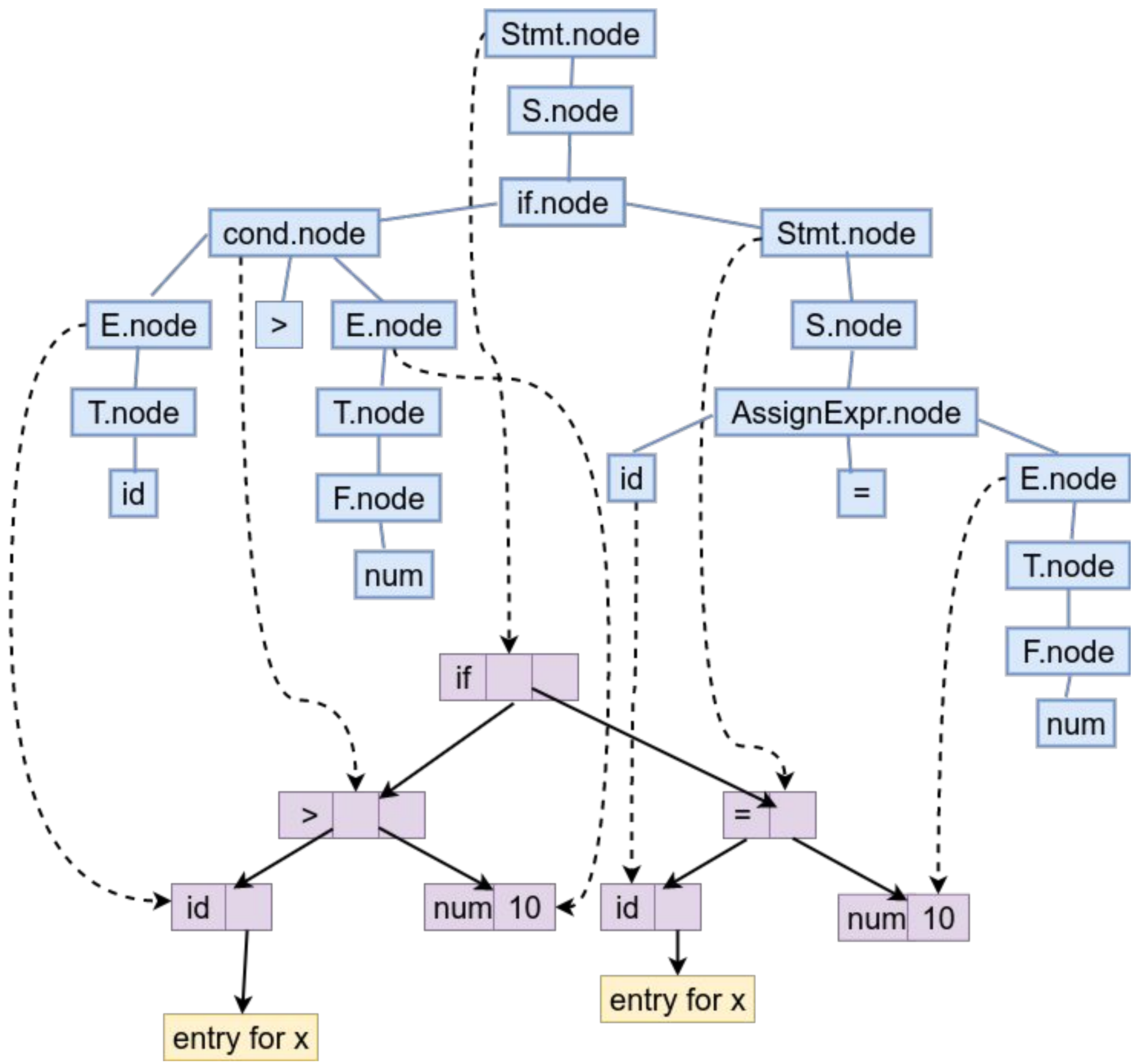
Example 2 - SDD to generate Syntax tree for Statements

Production	Semantic Rule
...	...
$E \rightarrow E_1 + T$	*{ E.node = new Node('+' , E_1.node, T.node); }*
$E \rightarrow T$	*{ E.node = T.node; }*
$T \rightarrow T_1 * F$	*{ T.node = new Node('*' , T_1.node, F.node); }*
$T \rightarrow F$	*{ T.node = F.node; }*
$F \rightarrow$ id	*{ F.node = new Leaf(id , id.entry); }*
$F \rightarrow$ num	*{ F.node = new Leaf(num , num.lexval); }*

Example 2 - SDD to generate Syntax tree for Statements

Use the previous grammar to construct the syntax tree for the input

```
if ( x > 10 )
{
    x = 10;
}
```



THANK YOU

Preet Kanwal
Department of Computer Science & Engineering
preetkanwal@pes.edu

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Sree Pranavi G

Compiler Design

Unit 3: L-Attributed SDD - Intermediate Code Generation

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- L–Attributed SDD to generate intermediate code for -
  - Expressions
  - Condition statement
  - If statement
  - If-else statement
  - While statement
  - Do - while statement
  - For statement
  - Boolean expressions

Recap

- There are 2 kinds of attributes - Synthesized and Inherited.
- An SDD with only synthesized attributes is an S-attributed definition.
- An SDD is L-attributed if all its attributes are either -
  - Synthesized
  - Extended synthesized dependent on children as well as inherited attributes.
  - Inherited but dependent only on inherited attributes at parent and any siblings at left.
- Every S-attributed SDD is also L-attributed.

SDD to generate Intermediate Code - Arithmetic Expressions

Write the SDD to generate intermediate code.

The given example indicates the code and its corresponding intermediate code for an expression a = b + - c .

Input	Output
a = b + - c	*t1 = minus c* *t2 = b + t1* *a = t2*

SDD to generate Intermediate Code - Arithmetic Expressions

Assigning appropriate semantic rules to each production -

Production	Semantic Rule
S -> id = E;	*S.code = E.code || gen(id.lexval '=' E.addr)*
E -> E1 + E2	*E.addr = new Temp();* *E.code = E1.code || E2.code ||* *gen(E.addr '=' E1.addr '+' E2.addr)*
E -> -E1	*E.addr = new Temp();* *E.code = E1.code || gen(E.addr '=' 'minus' E1.addr)*
E -> (E1)	*E.addr = E1.addr* *E.code = E1.code*
E -> id	*E.addr = id.lexval* *E.code = ' '*

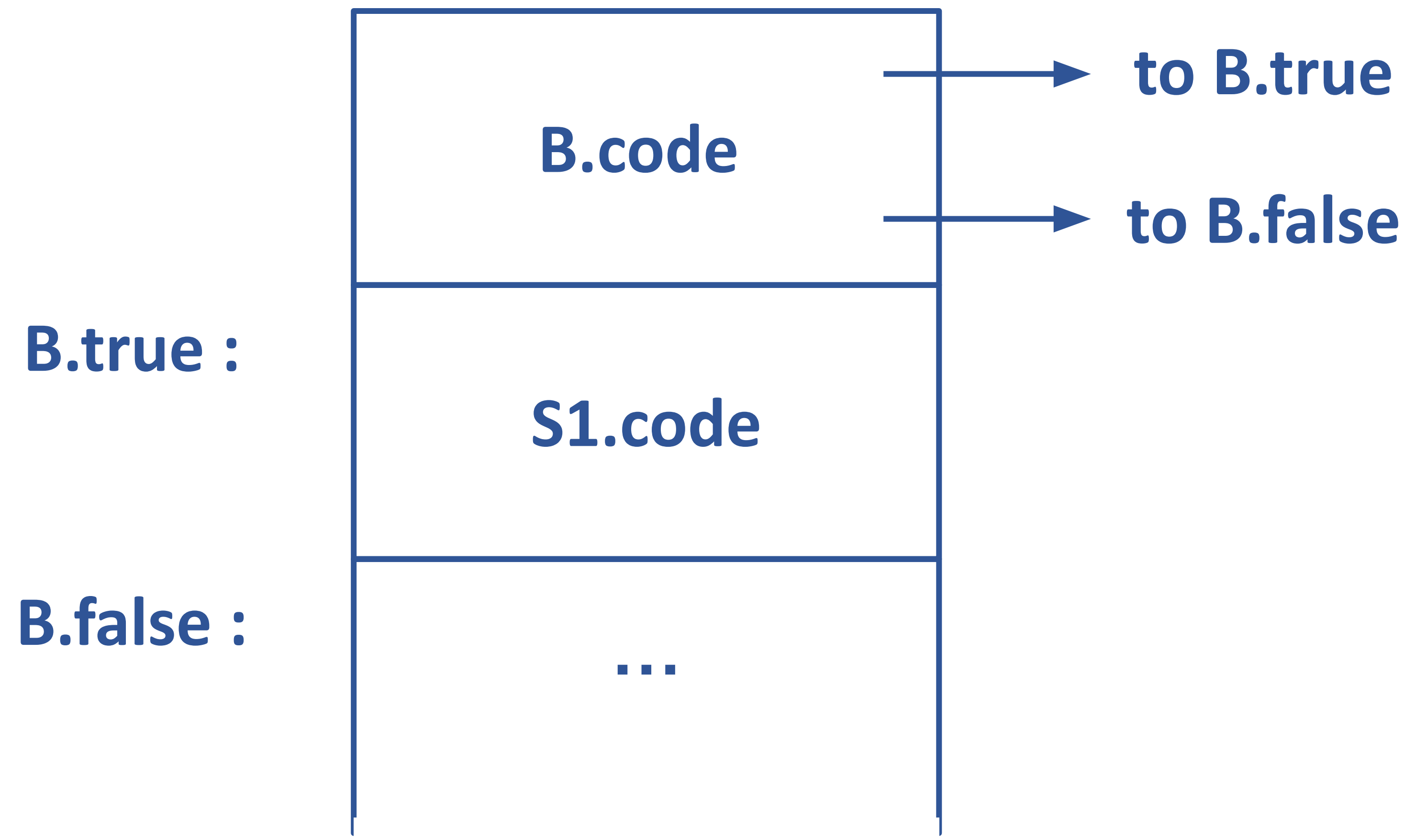
SDD to generate Intermediate Code - Conditions

Production	Semantic Rule
B -> id1 rel id2	*B.code = gen('if' id1.lexval rel.op id2.lexval 'goto' B.true)* *|| gen('goto' B.false)*
rel -> >	*rel.op = “>”*
rel -> <	*rel.op = “<”*
rel -> >=	*rel.op = “>=”*
rel -> <=	*rel.op = “<=”*

SDD to generate Intermediate Code - If Statement

Production	Semantic Rule
S -> if (B) S1	*L1 = new Label();* *B.true = L1;* *B.false = S.next;* *S1.next = S.next;* *S.code = B.code || label(L1) || S1.code*

SDD to generate Intermediate Code - If statement



SDD to generate Intermediate Code

- **If Else**

S -> if (C) S1 else S2

S2.next = S.next;

C.true = new label();

C.false = new label();

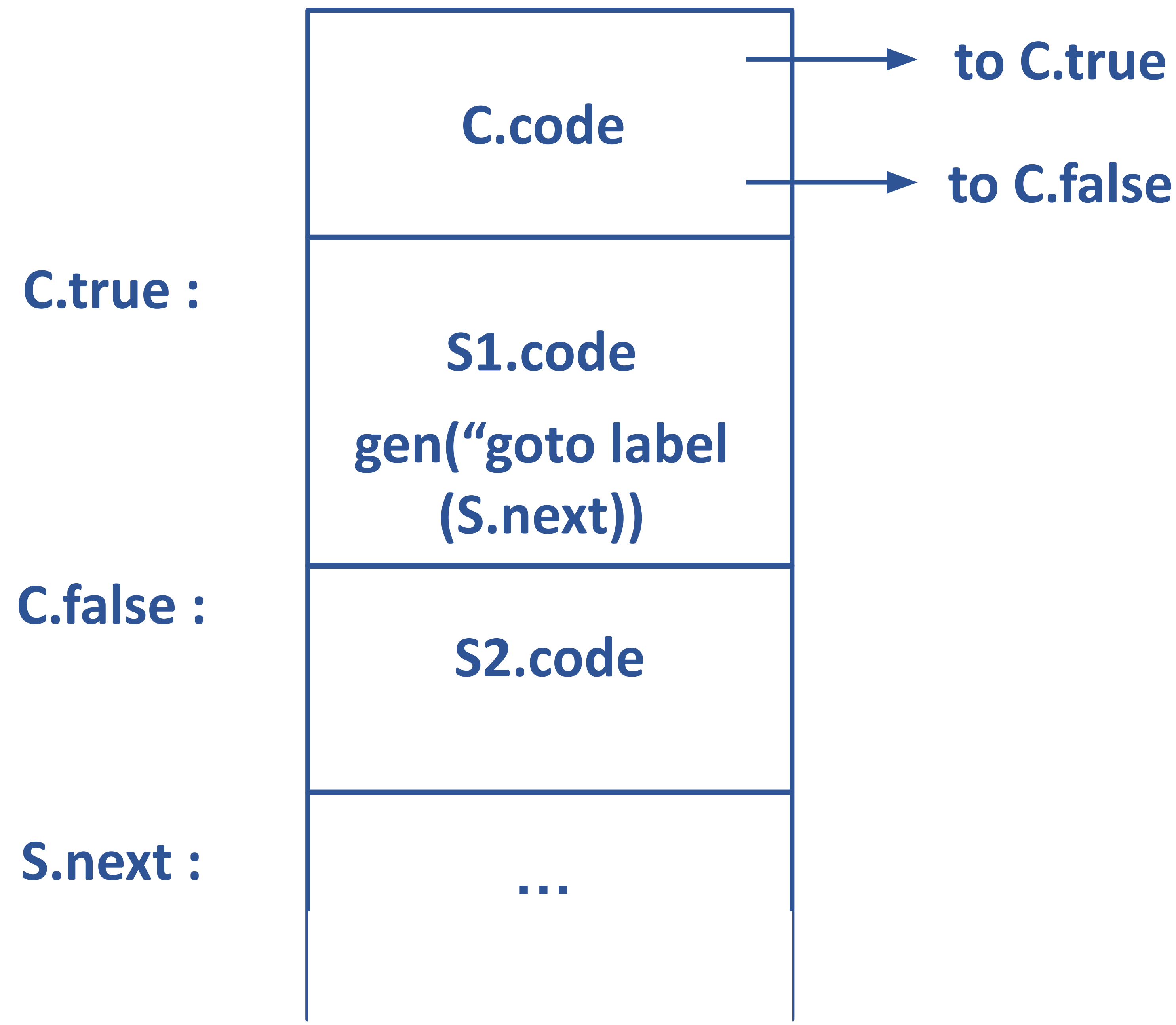
S1.next = S.next;

S.code = C.code || label(C.true) || S1.code || gen("goto" label(S.next)) || label(C.false) || S2.code

SDD to generate Intermediate Code

- If Else

  S -> if (C) S1 else S2



SDD to generate Intermediate Code

- **While**

S -> while (C) S1

begin = new label();

C.true = new label();

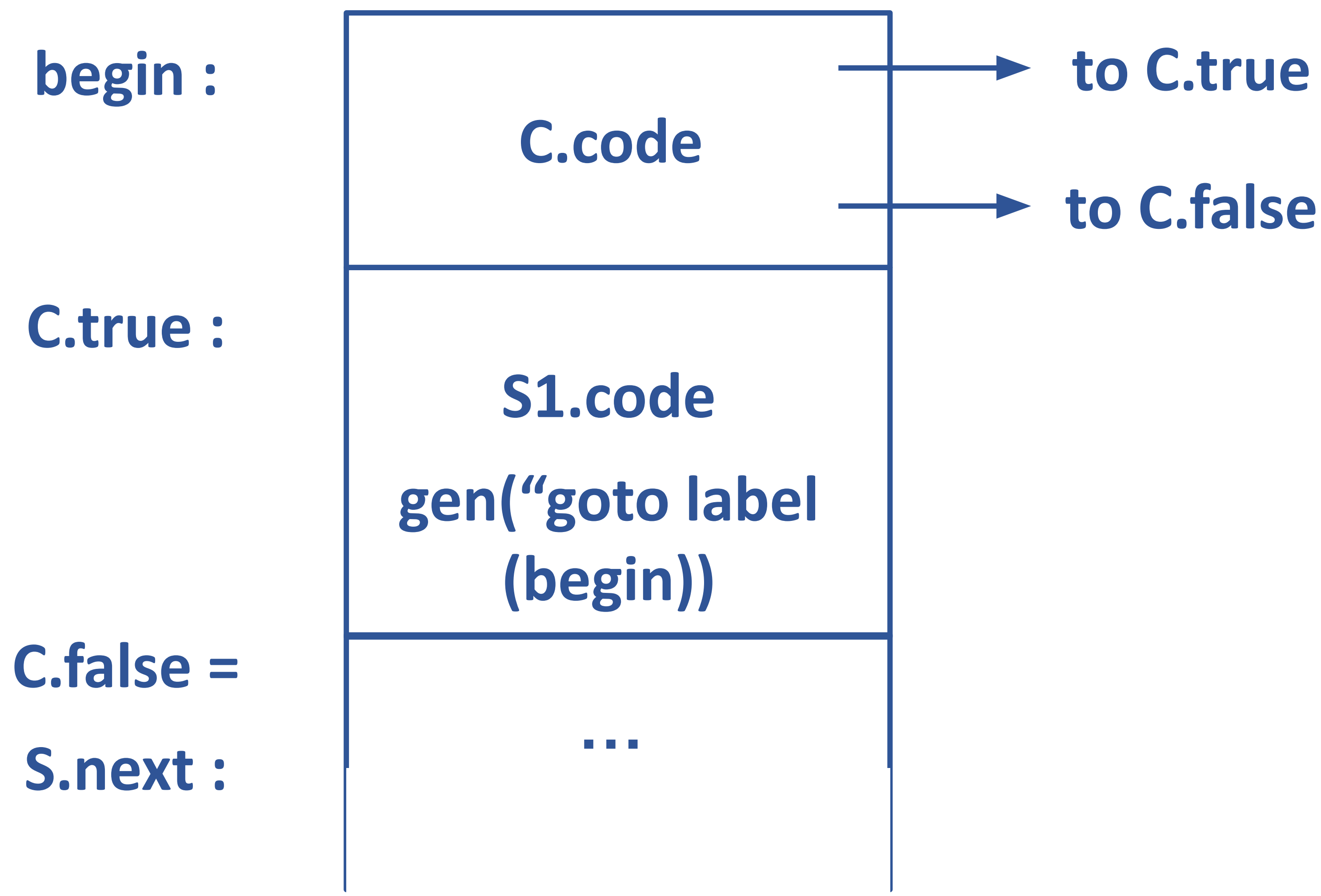
C.false = S.next;

S1.next = begin;

S.code = label(begin) || C.code || label(C.true) || S1.code || gen("goto" label(begin))

- While

  S -> while (C) S1



- **Do - While**

  S -> do (S1) while (C)

  C.true = new label();

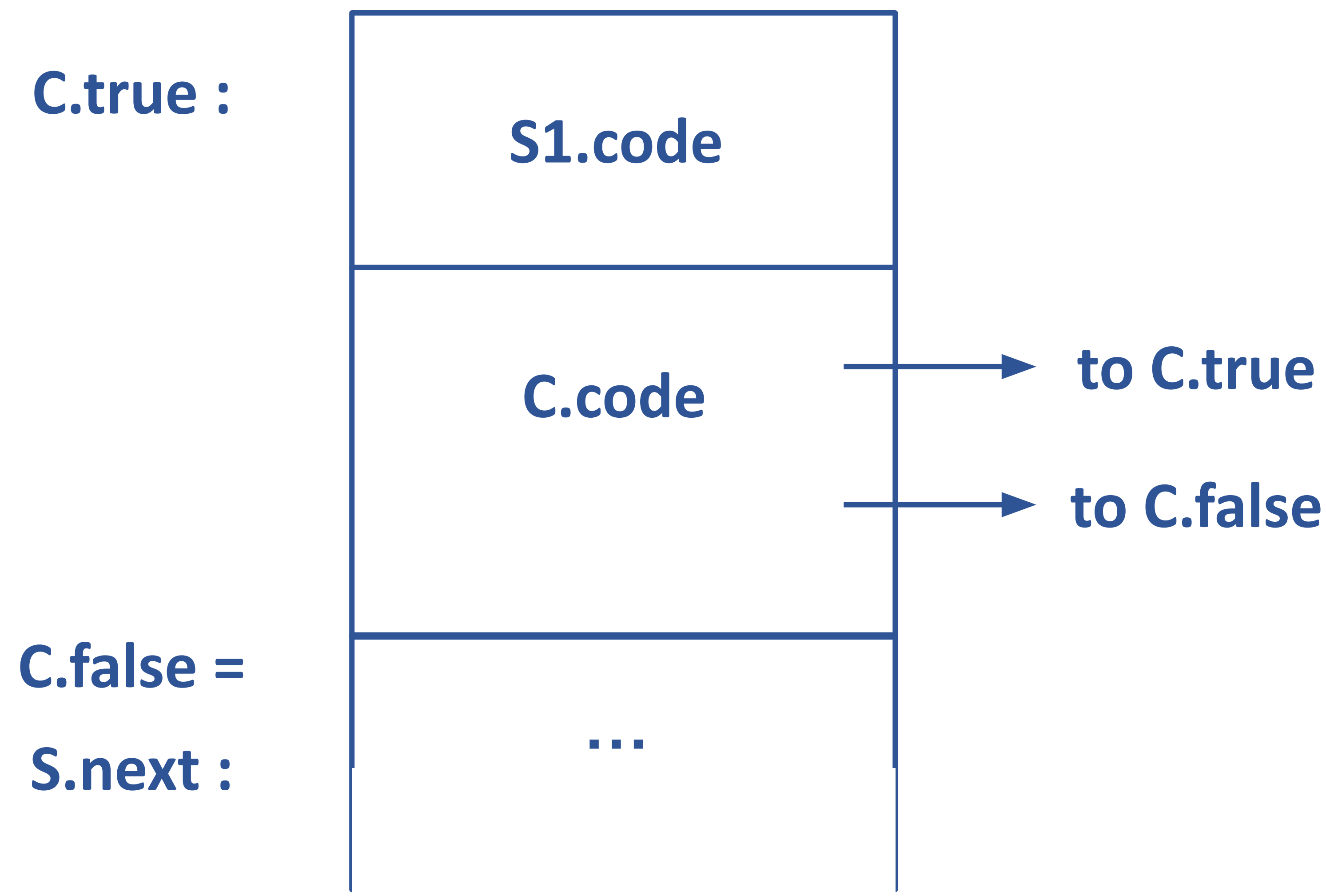
  C.false = S.next;

  S.code = label(C.true) || S1.code ||C.code

SDD to generate Intermediate Code

- Do - While

  S -> do (S1) while (C)



SDD to generate Intermediate Code

- **For**

S -> for (S1 ; C; S3) S4

C.true = new label();

C.false = S.next;

S3.next = new label();

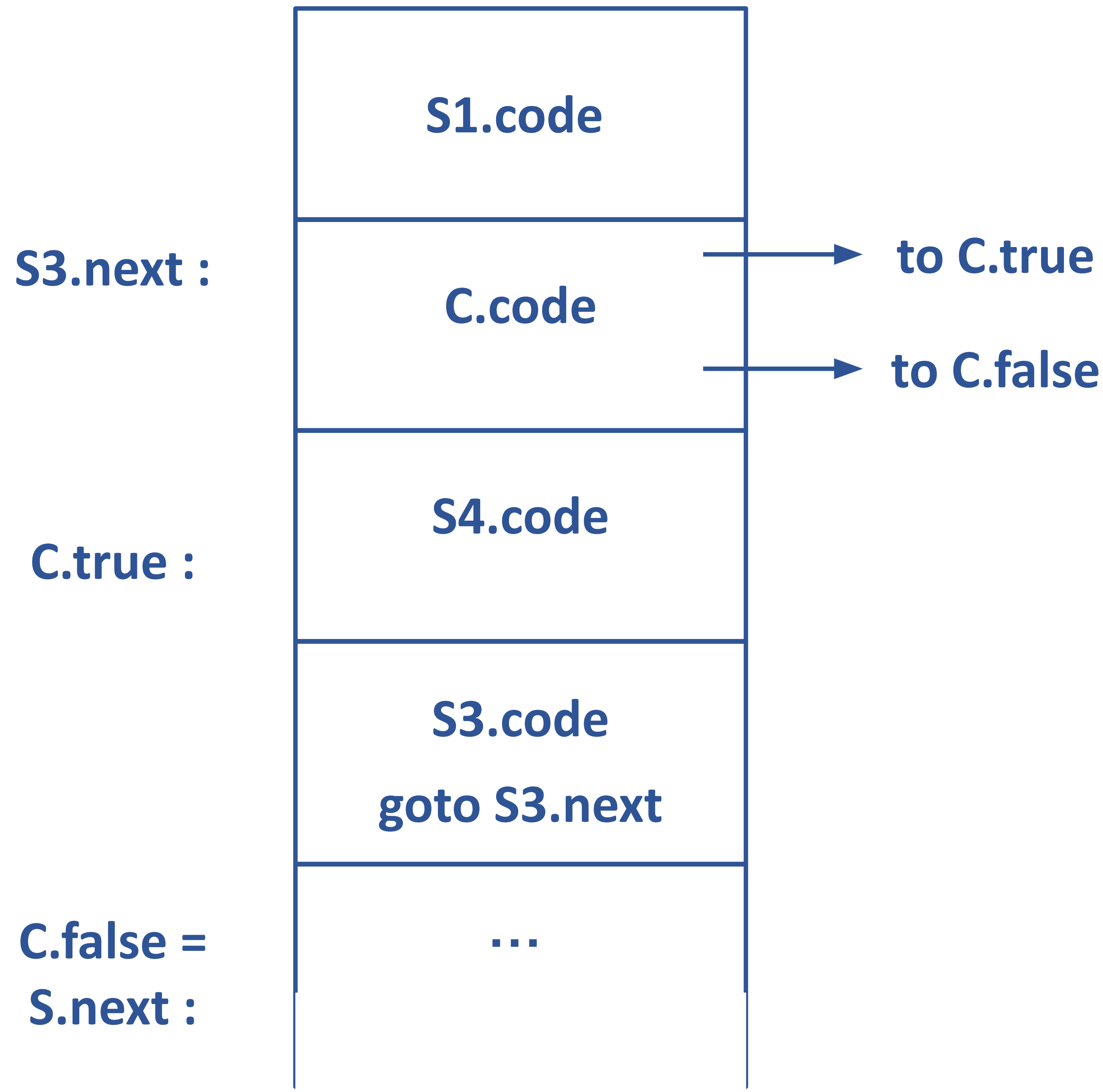
S.code = S1.code || label(S3.next) || C.code || label(C.true) || S4.code || S3.code || gen("goto S3.next);

Compiler Design

SDD to generate Intermediate Code

- For

  S -> for (S1 ; C; S3) S4



- **Boolean Expressions**

B -> B1 || B2

B.true = B1.true

B1.false = newl label();

B.true = B2.true

B.false = B2.false

B.code = B1.code || label (B1.false) || B2.code

- **Boolean Expressions**

B -> B1 && B2

B.false = B1.false

B1.true = newl label();

B.true = B2.true

B.false = B2.false

B.code = B1.code || label (B1.true) || B2.code

- **Boolean Expressions**

B -> ! B1

B.true = B1.false

B.false = B1.true

B.code = B1.code

SDD to generate Intermediate Code

- **Generate Intermediate Code for the following example :**

```
if ( x > 10)
{
    x = x + 1
}
```

- **Grammar -**

S -> if (B) { S1 }

B -> id1 > id2

S1 -> id = E;

E -> E1 + E2

E2 -> id

THANK YOU

Preet Kanwal
Department of Computer Science & Engineering

preetkanwal@pes.edu

Compiler Design

Preet Kanwal
Department of Computer Science & Engineering

Teaching Assistant : Sree Pranavi G

Compiler Design

Unit 3: Syntax Directed Translation

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- Syntax Directed Translation
- Design of Translation Schemes
- Types of Translation Schemes
- Problematic SDT
- Postfix schemes

Recap

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- Translations for programming language constructs guided by context-free grammars.
- There are 2 kinds of attributes - Synthesized and Inherited.
- An SDD with only synthesized attributes is an S-attributed definition.
- Every S-attributed SDD is also L-attributed.

Recap - Syntax Directed Translation

- We associate **Attributes** to the grammar symbols representing the language constructs.
- Values for attributes are computed by **Semantic Rules** associated with grammar productions.
- There are two notations for attaching semantic rules:
  - **Syntax Directed Definitions** : High-level specification hiding many implementation details.
  - **Translation Schemes** : More implementation oriented, indicate the order in which semantic rules are to be evaluated.

- Translation Schemes are more implementation oriented than syntax directed definitions since they indicate the order in which semantic rules and attributes are to be evaluated.
- A Translation Scheme is a context-free grammar in which,
  - Attributes are associated with grammar symbols.
  - Semantic Actions are enclosed between braces {} and are inserted within the right-hand side of productions.
- Yacc uses Translation Schemes.

Design of Translation Schemes

- When designing a Translation Scheme we must be sure that an attribute value is available when a semantic action is executed.
- When the semantic action involves only synthesized attributes, the action can be put at the end of the production.
- If we have an L-Attributed SDD we must enforce the following restrictions:
    - An inherited attribute for a symbol in the right-hand side of a production must be computed in an action before the symbol
    - A synthesized attribute for the non terminal on the left-hand side can only be computed when all the attributes it references have been computed - The action is usually put at the end of the production.

Implementation

- Ignoring the actions, parse the input and produce a parse tree as a result.

- Then, examine each interior node N, say one for production A -> α. Add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and the actions of α.

- Perform a preorder traversal of the tree, and as soon as a node labelled by an action is visited, perform that action.

Infix to prefix example

L -> En

E -> {printf("+");} E + T

E -> T

T -> {printf("*");} T * F

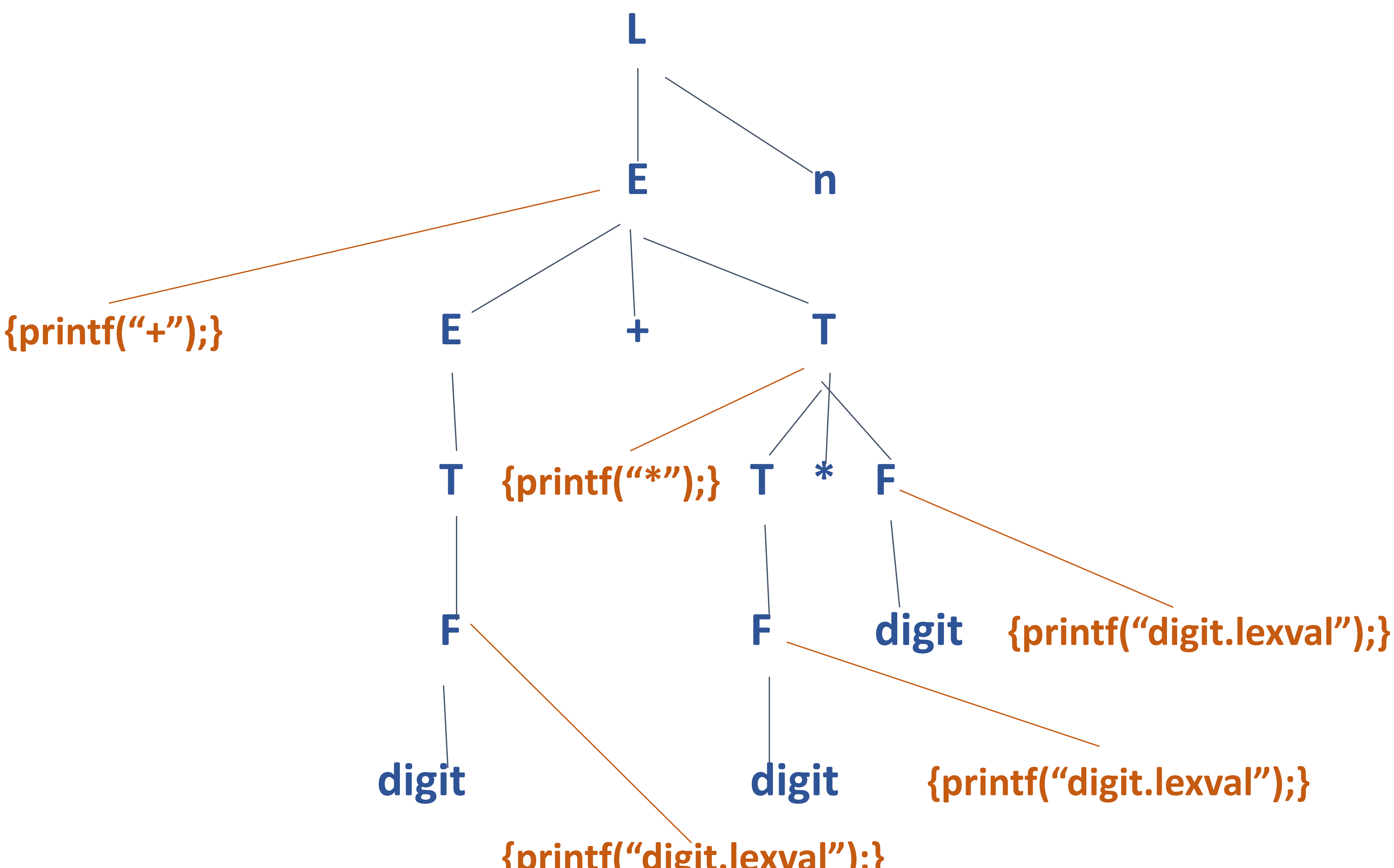
T -> F

F -> (E)

F -> digit {printf("digit.lexval");}

Problematic SDT

Infix to prefix example (contd.)



Problematic SDT

What does the following SDT scheme print for 5 + 4 - 2

E -> TR

R⟶+T {print("+");} R1

R⟶−T {print("-");} R1

R⟶ ε

T -> F

F -> digit {print(digit.lexval);}

- **S attributed to SDT.**

Evaluation of S-attributed SDD

- **S-attributed SDDs will have only synthesized attributes and can be evaluated by a bottom up parser.**

- **Since the attributes in the semantic actions are only synthesized, the actions can be placed at the end of the production.**

Rules for evaluation

- Consider the following production,

  A -> BCD

  before reducing BCD to A, the attributes of B, C and D must be computed before attribute of A which appears on the stack.

- Corresponding semantic action associated with the production must be executed.

Rules for evaluation

- **The parser stack is extended to have parallel value stack.**
- **If the Action appears at the end of production in a SDT, such SDTs are called Postfix SDTs.**

A	A.a
B	B.b
⋮	⋮

Parser / Value Stack

Parser Stack

Value Stack

THANK YOU

Preet Kanwal
Department of Computer Science & Engineering
preetkanwal@pes.edu

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Sree Pranavi G

Compiler Design

Unit 3: SDD to SDT

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- L-attributed SDD to SDT
- SDT Implementation
- S-attributed SDD to SDT (Postfix SDT)

Conversion of L-attributed SDD to SDT

Translation by traversing a parse tree

- **Build parse tree and annotate.**
- **Build the parse tree, add actions and execute the actions in pre-order.**

PES UNIVERSITY ONLINE

Translation by traversing a parse tree

- Build parse tree and annotate.
- Build the parse tree, add actions and execute the actions in pre-order.

Conversion of L-attributed SDD to SDT

Translation during parsing

- Use a RDP with a function for each non-terminal.
- Generate code on the fly, using a RDP.
- Implement an SDT in conjunction with an LL parser.
- Implement an SDT in conjunction with an LR parser.

Conversion of L-attributed SDD to SDT

- Place the computation of inherited attributes for a non-terminal before that non-terminal appears in the right hand side of the production.

- Place the computation of synthesized attributes at the end of production.

Evaluation of L-attributed SDD

- L-attributed SDDs can have both synthesized attributes and inherited attributes.

Rule to be followed for evaluation

- Place the semantic rule corresponding to the inherited attributes of the non-terminal before the non-terminal appears on the right hand side of the production.

- Place the semantic rule corresponding to the synthesized attributes of the non-terminal at the end of the production.

Conversion of L-attributed SDD

Example

Production	Semantic Rule
T -> FT’	T .in = F.syn; T.syn = T .syn
T ->*FT’	T’.inh = T’.inh * F.syn; T’ .syn = T’1.syn
T’ -> □	T’.syn = T’.inh
F -> digit	F.syn = digit.lexval

Example (contd.)

SDT

T -> F { T’.in = F.syn;} T’ {T.syn = T’.syn}

T’ -> *F {T’1.inh = T’.inh * F.syn;} T’1 {T’.syn = T’1.syn}

T1 -> □ { T’.syn = T’.inh;}

F -> digit { F.syn = digit.lexval; }

L-attributed SDD to SDT - Type Declarations

Production	Semantic Rule	Translation Scheme
D $\rightarrow$ T L	*{ L.in = T.type;}*	D $\rightarrow$ T { L.in := T.type } L
T $\rightarrow$ int	*{ T.type = integer; }*	T $\rightarrow$ int { T.type := 'integer' }
T $\rightarrow$ real	*{ T.type = float; }*	T $\rightarrow$ real { T.type := 'real' }
L $\rightarrow$ L_1 , id	*{ L_1.in= L.in;* *addType(id.entry, L.in); }*	L $\rightarrow$ { L1.in := L.in } L1, id { addtype(id.entry, L.in) }
L $\rightarrow$ id	*{ addType(id.entry, L.in); }*	L $\rightarrow$ id { addtype(id.entry, L.in) }

Convert the L-attributed SDD to generate intermediate code for If statement to SDT -

Production	Semantic Rule
S → if(C) S1	*L1 = new label();* *C.true = L1;* *C.false = S.next;* *S1.next = S.next;* *S.code = C.code || label(L1) || S1.code*

PES UNIVERSITY ONLINE

Translation Scheme -

S → if ({ L1 = new label (); C.true = L1 C.false = S.next S1.next = S.next } C) S1 { S.code = C.code || L1 || S1.code }

Convert the L-attributed SDD to generate intermediate code for while statement to SDT :

Production	Semantic Rule
S ⟶ while (C) S1	*L1 = new label();* *L2 = new label();* *C.true = L2* *C.false = S.next* *S1.next = L1* *S.code = label(L1) || C.code || label(L2) || S1.code || gen('goto' begin)*

PES UNIVERSITY ONLINE

Translation Scheme -

S → while ({ L1 = new label ();
L2 = new label ();
C.true = L2;
C.false = S.next;
S1.next = L1; }
C) S1 { S.code = L1 || C.code ||
L2 || S1.code }

- Ignoring the actions, parse the input and produce a parse tree as a result.
- Examine each interior node N.
- Add additional children to N for the actions.
- Perform a preorder traversal of the tree, and as soon as a node labelled by an action is visited, perform that action.

- Consider the following translation scheme.

  S → ER

  R → *E { print("*"); } R | ε

  E → F + E { print("+"); } | F

  F → (S) | id { print(id.value); }

Here id is a token that represents an integer and id.value represents the corresponding integer value.

For an input 2 * 3 + 4, this translation scheme prints ________________.

SDT Implementation - Infix to Postfix

- Convert the SDD for infix expression to prefix to SDT.

Production	Semantic Rule
$E \longrightarrow E1 + T$	
$E \longrightarrow T$	
$T \longrightarrow T1 * F$	
$T \longrightarrow F$	
$F \longrightarrow num$	
$F \longrightarrow id$	

SDT Implementation - Infix to Postfix

- **Convert the SDD for infix expression to prefix to SDT.**

Production	Semantic Rule
E → E1 + T	*E → {printf("+");} E1+ T*
E → T	*E → T*
T → T1 * F	*T → {printf("*");} T1* F*
T → F	*T → F*
F → num	*F → num {printf("%d", num.lexval);}*
F → id	*F → id {printf("%d", id.lexval);}*

Evaluation of S-Attributed Definitions

- Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.
- The parser keeps the values of the synthesized attributes in its stack.
- Whenever a reduction $A \longrightarrow \alpha$ 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 simply implemented by extending the stack of an LR-Parser.

Extending a Parser Stack

- Extra fields are added to the stack to hold the values of synthesized attributes.
- In the simple case of just one attribute per grammar symbol the stack has two fields - state and val.
- The current top of the stack is indicated by the pointer top.

- Synthesized attributes are computed just before each reduction :
  - Before the reduction $A \longrightarrow XY\ Z$ is made, the attribute for A is computed :

    A.a := f (s[top].val, s[top − 1].val,
    s[top − 2].val).

state	val
Z	*Z.x*
Y	*Y.x*
X	*X.x*
…	…

Compiler Design

Postfix SDT scheme for simple desk calculator

E → E1 + T {stack[top-2].val = stack[top-2].val + stack[top].val;
top = top - 2; }

E → T

T → T1 * F {stack[top-2].val = stack[top-2].val * stack[top].val;
top = top - 2; }

T → F

F → (E) {stack[top-2].val = stack[top-1].val;
top = top - 2; }

F → digit

Implementing L-attributed SDD during LR Parsing

- Introduce a Marker Non-terminal in place of each embedded action.

- There is one production for each Marker M,

  M → epsilon

Example :

A → alpha {a} beta

would convert to

A → alpha M beta

M → epsilon {a}

- **Example**

S → while ({ L1 = new label (); L2 = new label (); C.false = S.next; } C) { S1.next = L1; } S1 { S.code = L1 || C.code || L2 || S1.code }

Implementing L-attributed SDD during LR Parsing

- **Example continued…**

$S \rightarrow$ while (M C) N S1

{
S.code = L1 ||
C.code || L2 ||
S1.code
}

Implementing L-attributed SDD during LR Parsing

- **Example continued…**

M → epsilon { L1 = new label(); L2 = new label(); C.true = L2; C.false = S.next; }

N → epsilon { S1.next = L1; }

Parser Stack Structure

Stack record	*A*	*Synthesized attributes of A*
Record of Marker A	Inherited attributes of A	

Note : We perform general style of bottom-up parsing - shift-reduce parsing.
A - non terminal

Implementing L-attributed SDD during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$	while (cond)Stmt $ (dummy string)	Shift while
$ while while $	(cond)Stmt $ S → while (M C) N S1	Shift (
$ while ((while $	cond)Stmt $ L1 = new label(); L2 = new label(); C.true = L2; C.false = S.next;	Reduce using M → epsilon and execute the action

Implementing L-attributed SDD during LR Parsing

- **Example continued…**

<table>
<tr><th>Stack</th><th>Input Buffer</th><th>Action</th></tr>
<tr><td>$ while (M
<table><tr><td>M</td><td>L1</td><td>L2</td><td>C.true</td><td>C.false</td></tr><tr><td colspan="5">(</td></tr><tr><td colspan="5">while</td></tr><tr><td colspan="5">$</td></tr></table></td><td>cond)Stmt $</td><td>Shift C</td></tr>
<tr><td>$ while (M C
<table><tr><td colspan="2">C</td><td colspan="3">C.code</td></tr><tr><td>M</td><td>L1</td><td>L2</td><td>C.true</td><td>C.false</td></tr><tr><td colspan="5">(</td></tr><tr><td colspan="5">while</td></tr><tr><td colspan="5">$</td></tr></table></td><td>)Stmt $</td><td>Shift)</td></tr>
</table>

Implementing L-attributed SDD during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ while (M C)	Stmt $	Reduce using N → epsilon { S1.next = L1; }

Stack contents (top to bottom):

)				
C			C.code	
M	L1	L2	C.true	C.false
(				
while				
$				

S → while (M C) N S1

Implementing L-attributed SDD during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ while (M C) N	Stmt $	Shift S1

N	S1.next = s[top-3].L1			
)				
C	C.code			
M	L1	L2	C.true	C.false
(				
while				
$				

Implementing L-attributed SDD during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ while (M C) N S	$	Reduce using S → while(MC)NS and execute the action

pos					
top	S1		S1.code		
top-1	N		S1.next = s[top-3].L1		
top-2	)				
top-3	C		C.code		
top-4	M	L1	L2	C.true	C.false
top-5	(				
top-6	while				
	$				

s[top-6].code = s[top-4].L1 ||
s[top-3].code || s[top-4].L2 ||
s[top].code;

top = top-6;

$$\left\{ \begin{array}{c} \text{S.code = L1 ||} \\ \text{C.code || L2 ||} \\ \text{S1.code} \end{array} \right\}$$

Implementing L-attributed SDD during LR Parsing

- **Example continued…**

Stack	Input Buffer	Action
$ S <table><tr><td>S</td><td>S.code</td></tr><tr><td colspan="2">$</td></tr></table>	$	Accept

Implementing L-attributed SDD during LR Parsing

- **Example**

S → if ({ L1 = new label (); C.true = L1 C.false = S.next; } C) { S1.next = S.next; } S1 { S.code = C.code || L1 || S1.code }

Implementing L-attributed SDD during LR Parsing

PES UNIVERSITY ONLINE

- Example

S → if (M C) N S1 { S.code = C.code || L1 || S1.code }

M → epsilon { L1 = new label(); C.true = L1; C.false = S.next; }

N → epsilon { S1.next = S.next; }

PES UNIVERSITY ONLINE

THANK YOU

Preet Kanwal
Department of Computer Science & Engineering
preetkanwal@pes.edu

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 4: Intermediate Code Generation

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

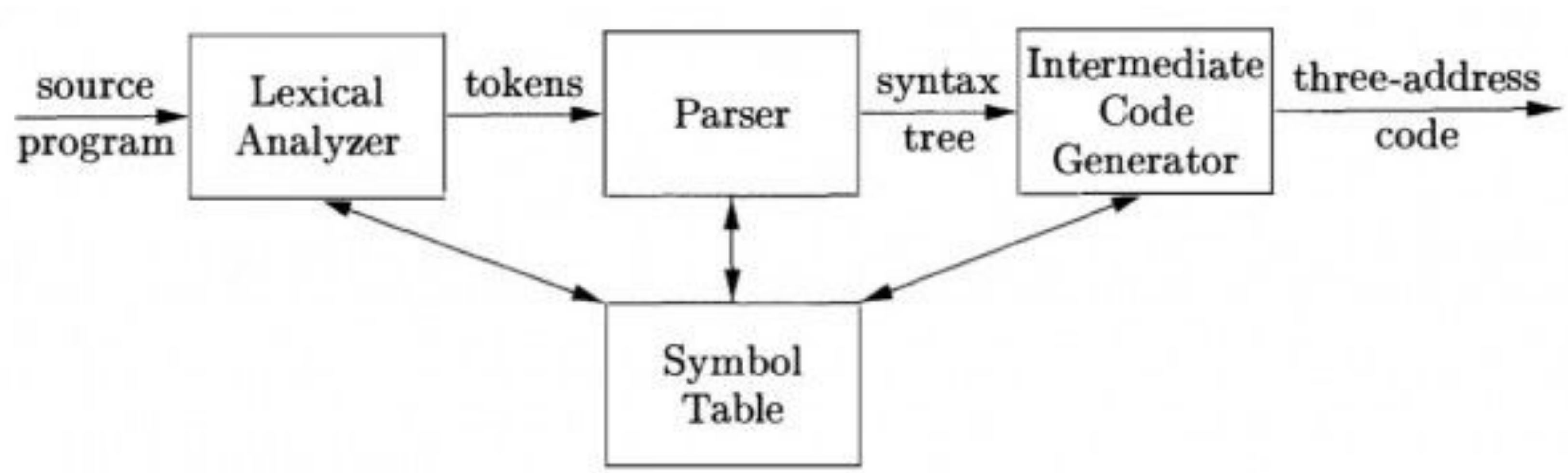
In this lecture, you will learn about -

- What is intermediate code?
- Why intermediate code generation?
- Advantages of ICG
- Types of Intermediate Representation
- Directed Acyclic Graph
    - Applications
    - SDD to construct a DAG
    - Examples of Syntax tree vs DAG

What is Intermediate code?

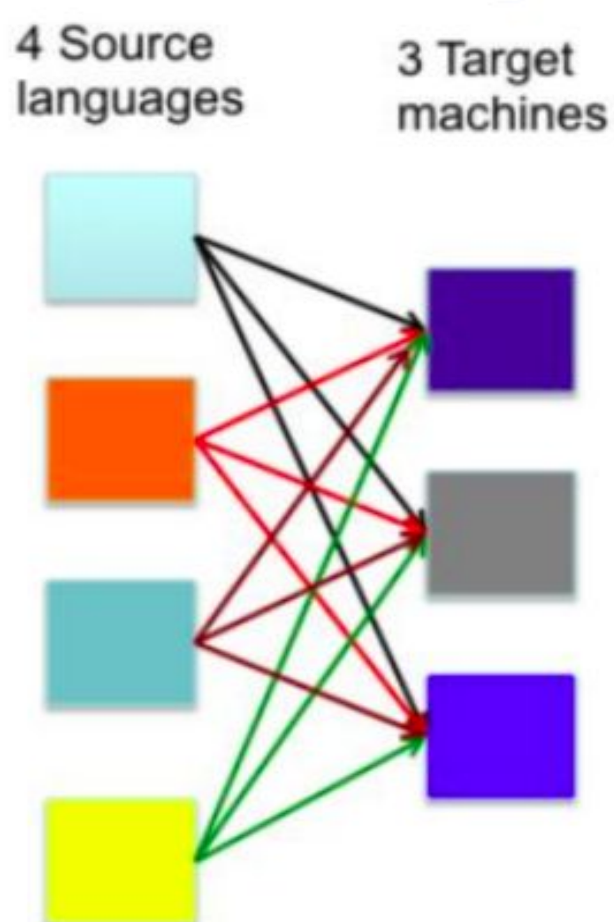
- Intermediate code is used to translate the source code into the machine code.
- It lies between the high-level language and the machine language.
- The Intermediate code generator receives input from the semantic analyzer.

  It takes input in the form of an annotated syntax tree.
- Using the intermediate code, the second phase of the compiler (synthesis phase) is changed according to the target machine.

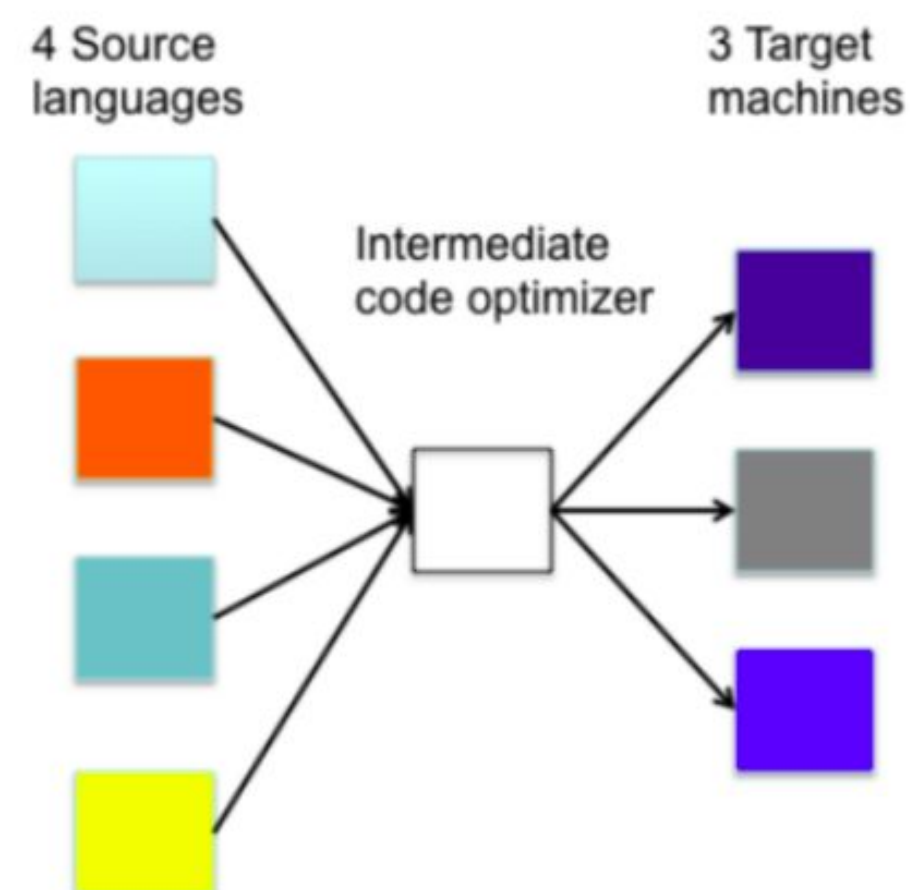


Why Intermediate code generation?

- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.



4 front-ends + 4x3 optimisers + 4x3 code generators



4 front-ends + 1 optimiser + 3 code generators

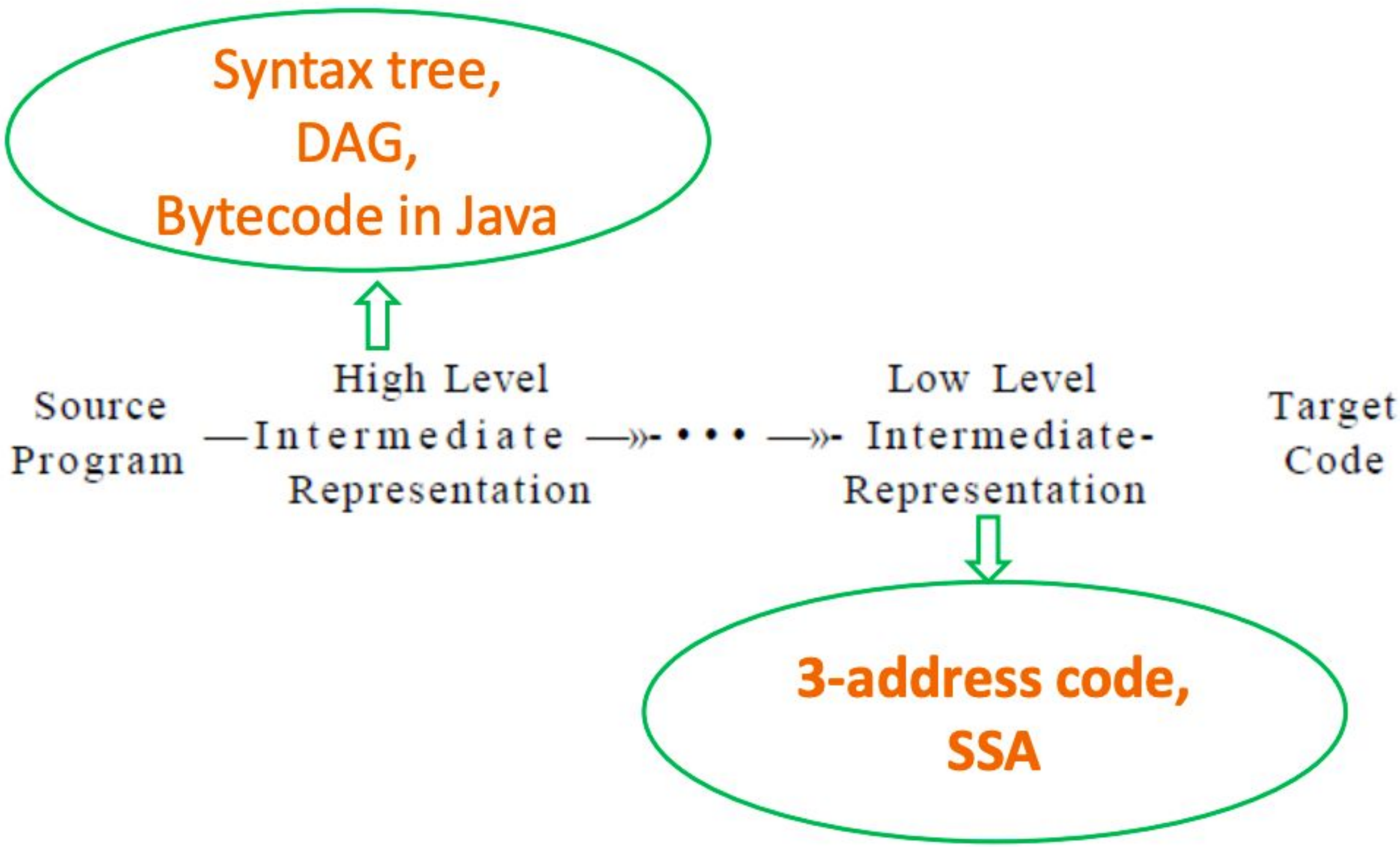
Advantages of Intermediate Code Generation

- ICG makes it easier to construct compilers for different architectures.
- Target code can be generated for any machine just by attaching new machine as the back end - this is called retargeting.
- It is possible to apply machine independent code optimization - helps in faster generation of code.

Types of Intermediate Representation

- An intermediate representation is a representation of a program between the source and target languages.
- A good IR is one that is fairly independent of the source and target languages - this maximizes its ability to be used in a retargetable compiler.
- There are three ways to classify Intermediate representation -
    - High-level or Low-level
    - Language-specific or Language independent
    - Graphical or Linear

Intermediate Representation - High level vs Low level representation



Intermediate Representation - High level representation

- High-level intermediate code representation is very close to the source language itself.
- They can be easily generated from the source code
- Code modifications can be easily applied to enhance performance.
- Examples- Syntax trees, DAG, Java Bytecode

Intermediate Representation - Low level representation

- Low level intermediate code representation is close to the target machine.
- This makes it suitable for register and memory allocation, instruction set selection, etc.
- It is good for machine-dependent optimizations.
- Examples - Three Address Code, SSA

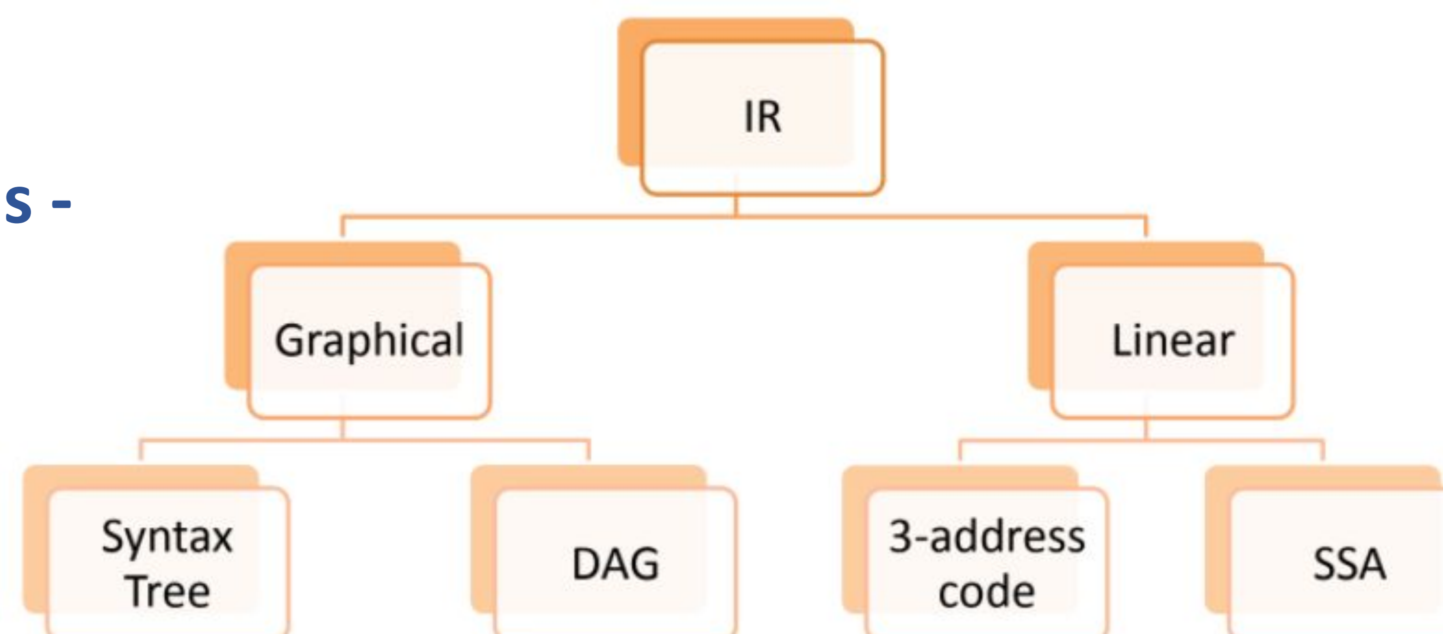
Intermediate Representation

In terms of language, Intermediate code can be either -

- Language specific - Byte Code for Java, P-code for Pascal
- Language independent - three-address-code

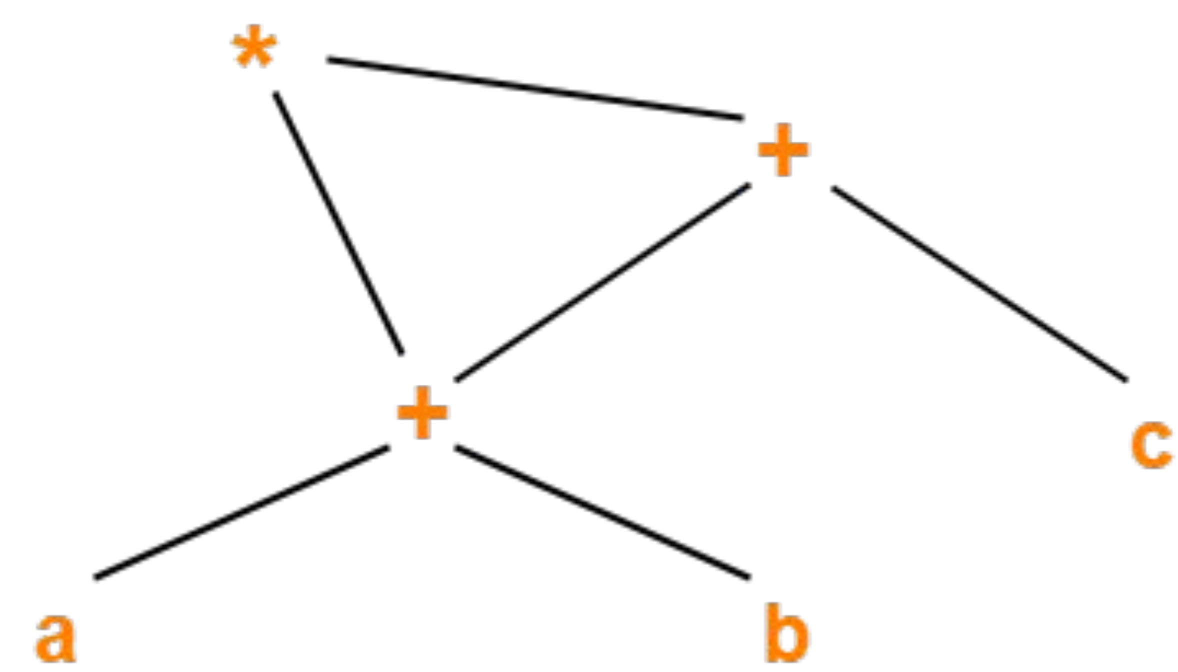
Intermediate code can be also classified as -

- Graphical
- Linear



DAG - Directed Acyclic Graph

- It is a variant of Syntax tree with a unique node for each value.
- It does not contain cycles.
- In a DAG,
  - Interior nodes always represent the operators.
  - Exterior nodes (leaf nodes) always represent the names, identifiers or constants.
- The given figure represents the DAG for the expression (a + b) x (a + b + c)



Directed Acyclic Graph

Applications of DAG

- It helps optimize code by identifying common subexpressions in a syntax tree.
- It reduces no. of calculations to be done - calculate once,refer anywhere.
- It can be used to determine the names whose computation has been done outside the block but used inside the block.
- It can also be used to determine the statements of the block whose computed value can be made available outside the block.

SDD to construct a DAG

- SDD used to generate Syntax tree will be used to construct DAG too, with a simple check -

    if an identical node exists

        RETURN existing node

    else

        CREATE a new node

- The assignment instructions of the form x:=y are not performed unless they are necessary.
- The process of making this check is an overhead; hence constructing DAG is costly.

Exercise 1 - Construct DAG for the given expression

Consider the following unambiguous grammar -

E -> E + T | E – T | T

T -> T * F | T / F | F

F -> (E) | [E] | id

Using this, construct Syntax tree and DAG for the following expression -

((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))

Exercise 1 - Solution

Given - ((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))

Step 1 - Rewrite the expression for clear understanding

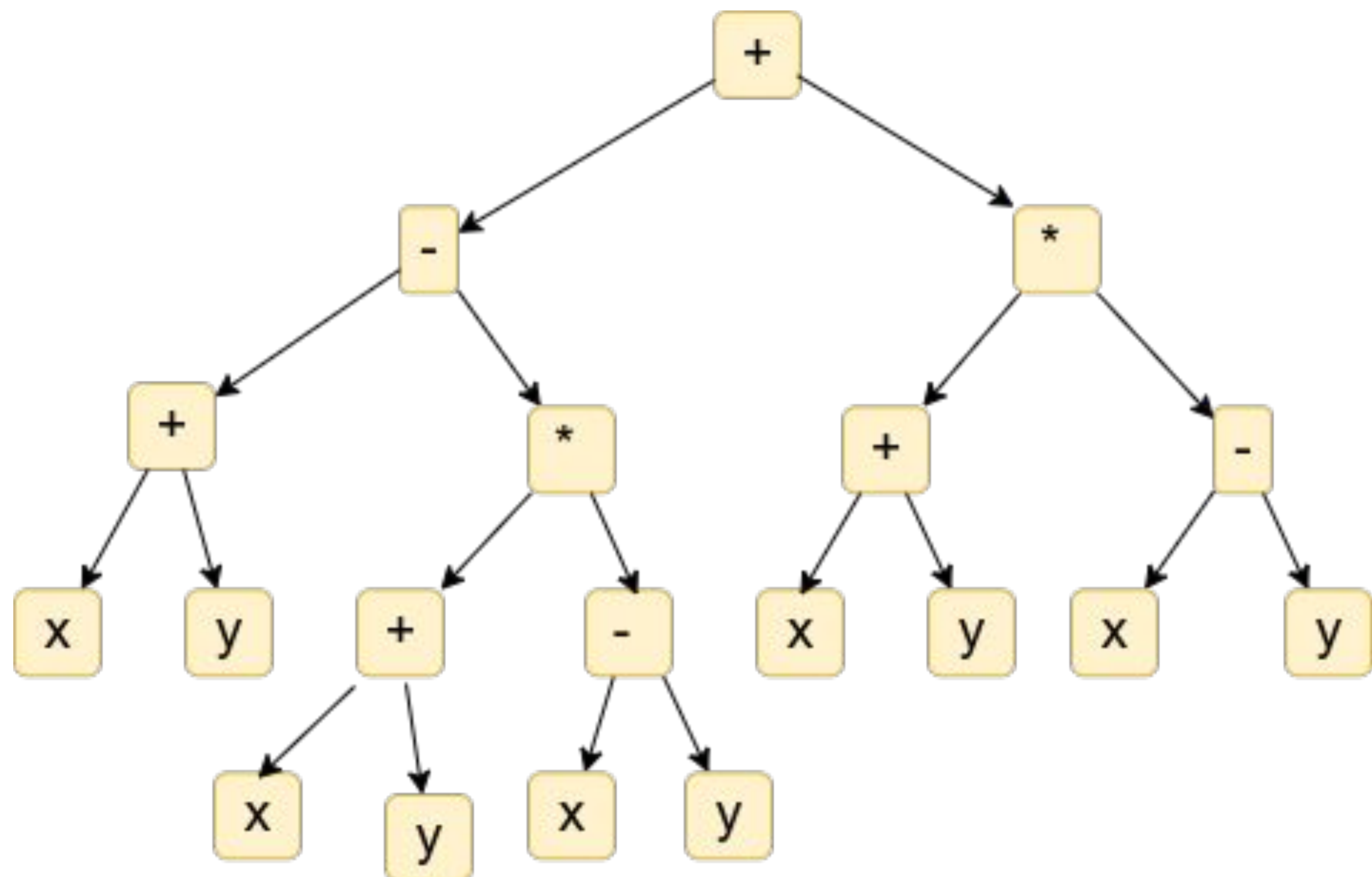
```
(
    (x + y) - (
                (x + y) * (x - y)
              )
)
+
(
    (x + y) * (x - y)
)
```

Compiler Design

Exercise 1 - Solution

Given - ((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))

Step 2 - Draw the Syntax tree

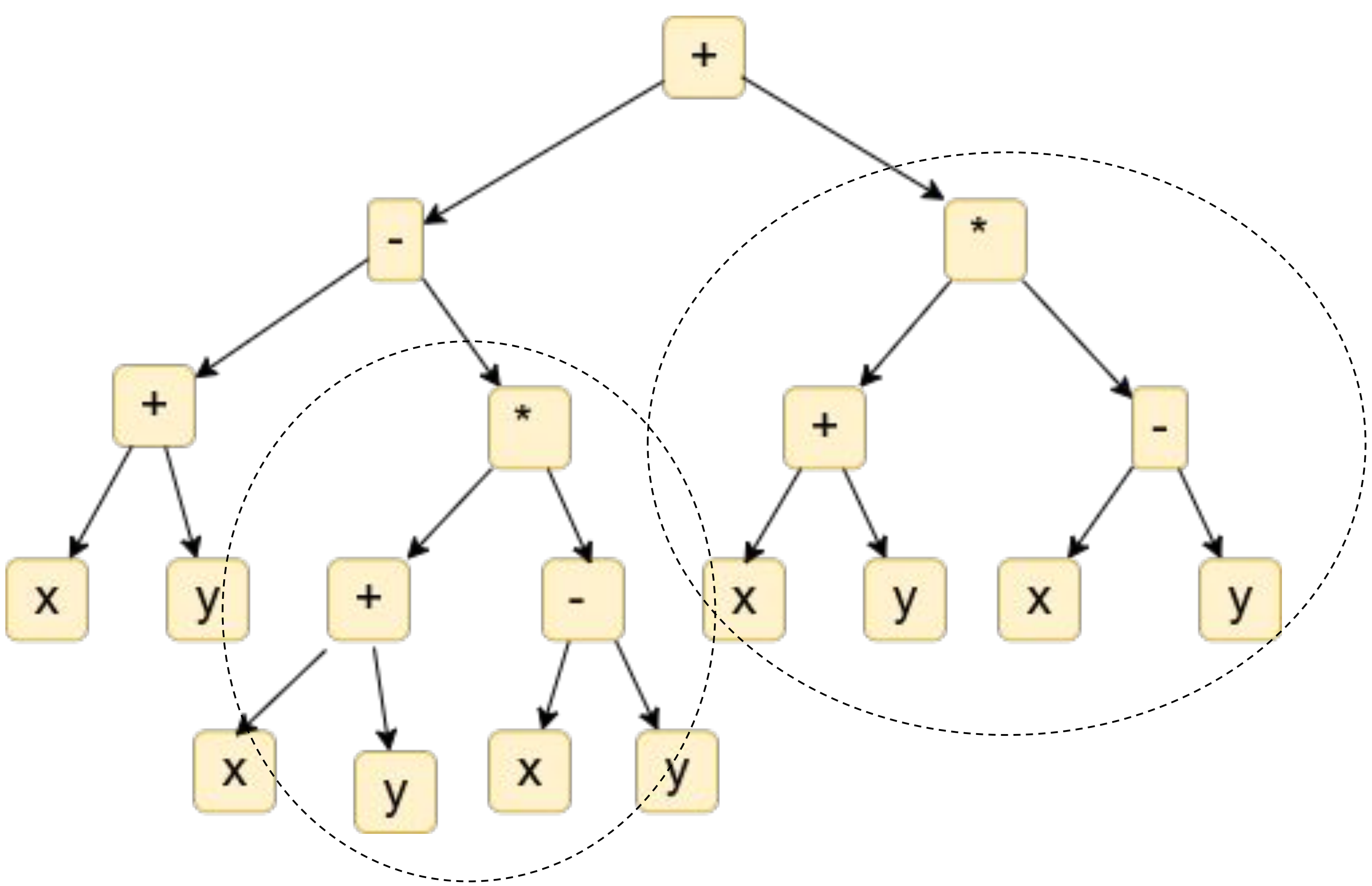


Compiler Design

Exercise 1 - Solution

Given - ((x + y) – ((x + y) * (x – y))) + ((x + y) * (x – y))

Step 3 - Identify the common subexpressions and eliminate step wise



Compiler Design

Exercise 2

Consider the following unambiguous grammar -

E -> E + T | E – T | T

T -> T * F | T / F | F

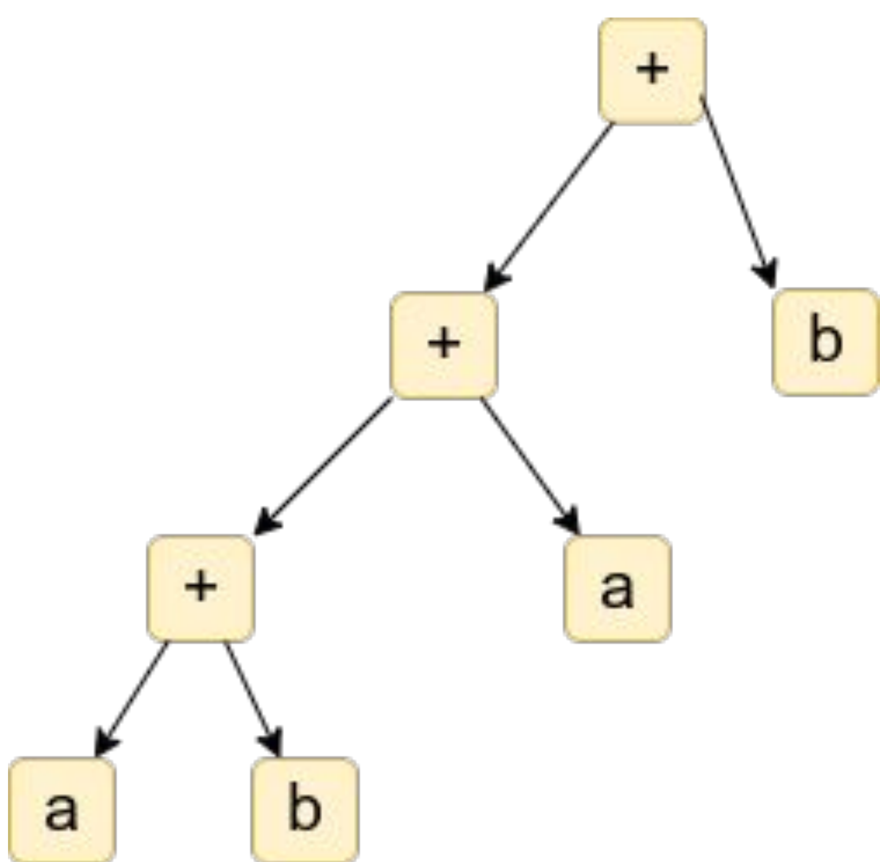
F -> (E) | [E] | id

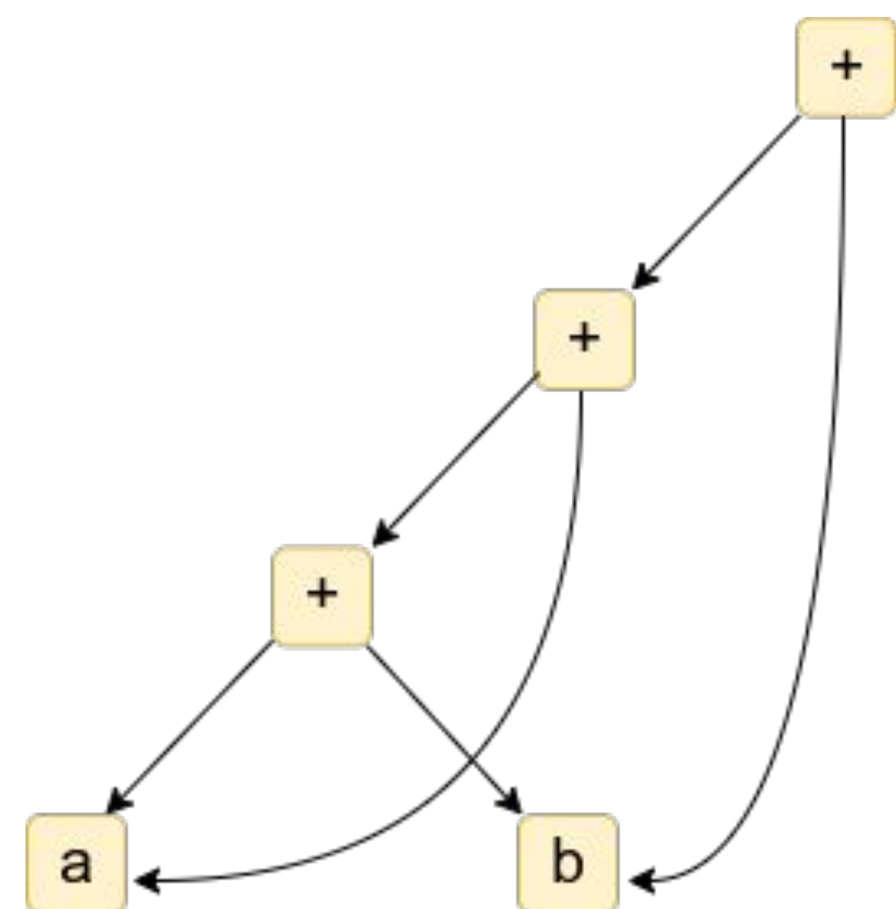
Using this, construct Syntax tree and DAG for the following expressions -

1) a + b + a + b
2) a + b + (a + b)
3) a + a * (b – c) + (b – c) * d
4) (((a + a) + (a + a)) + ((a + a) + (a + a)))
5) [(a + b) * c + ((a + b) + e) * (e + f)] * [(a + b) * c]

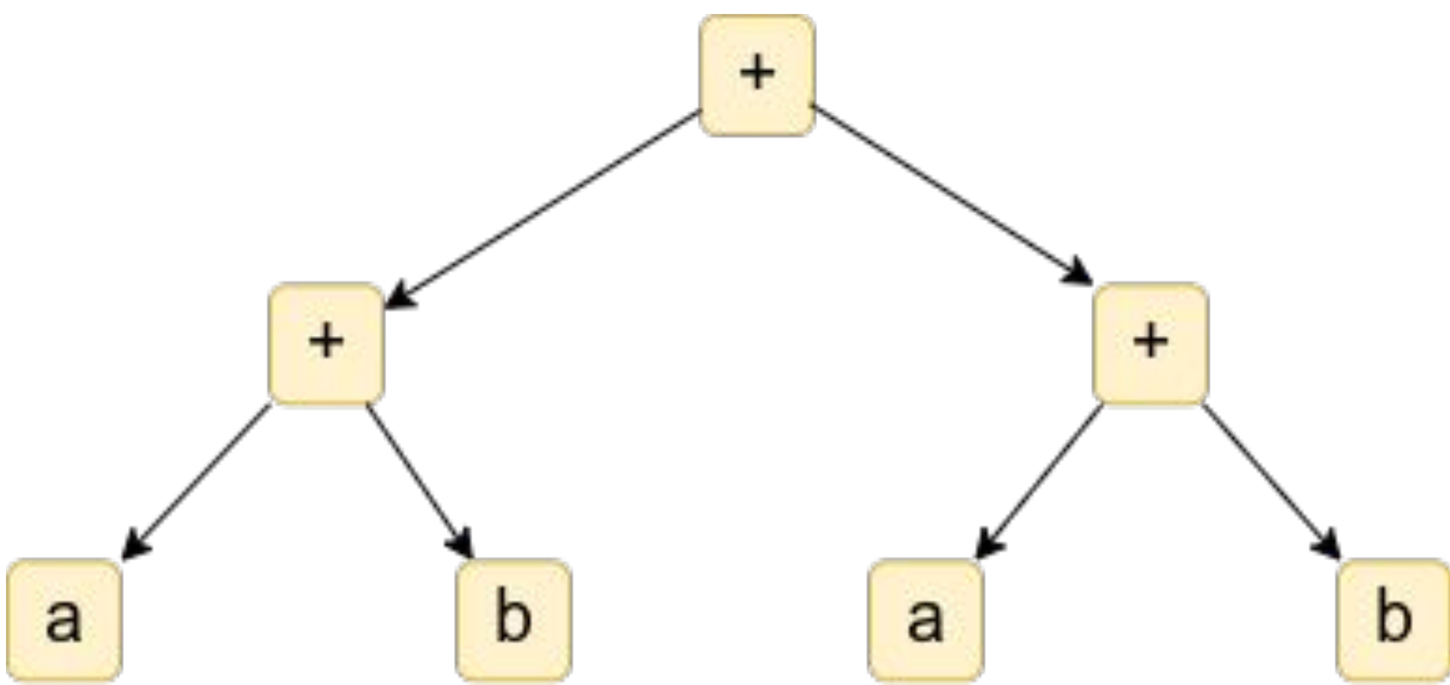
Exercise 2 - Solutions

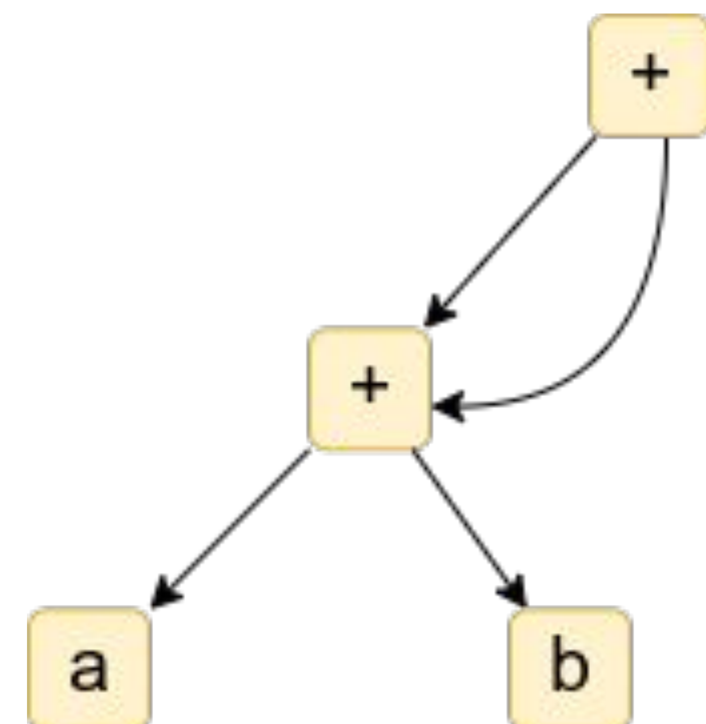
1) a + b + a + b





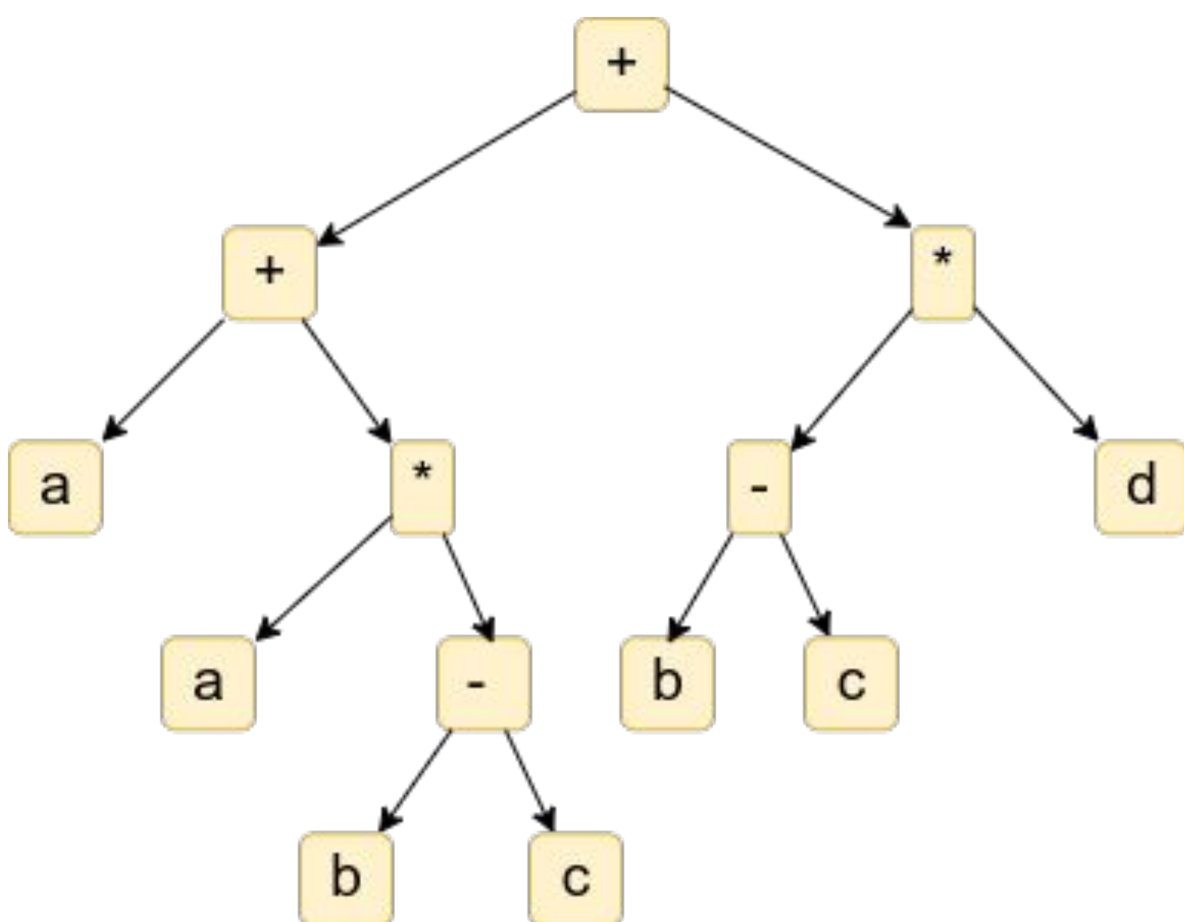
2) a + b + (a + b)

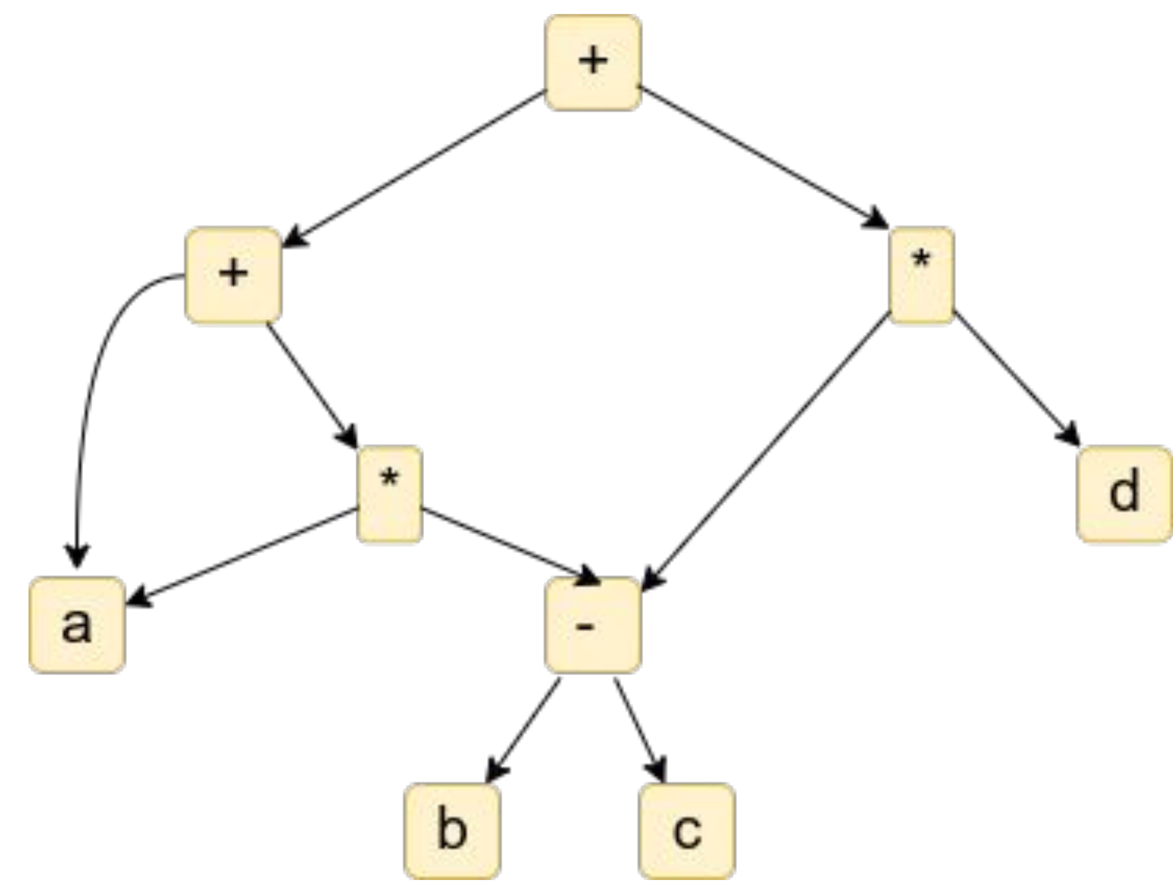




Exercise 2 - Solutions

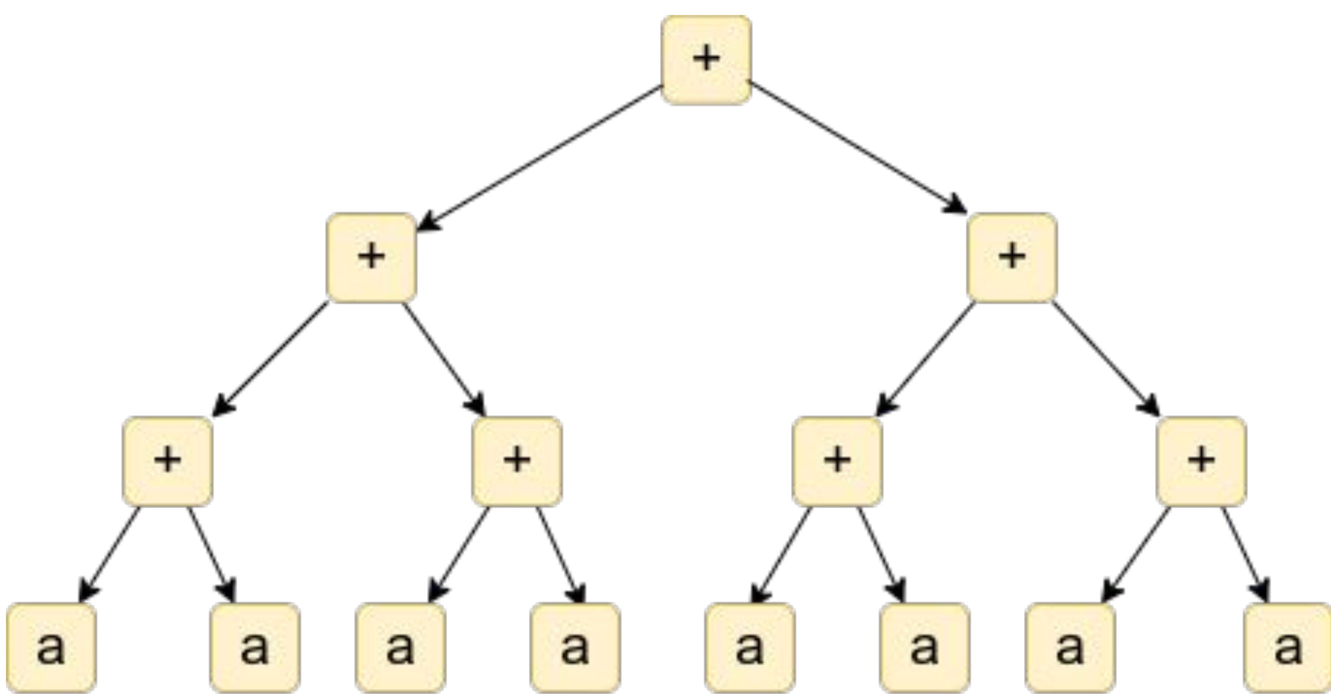
3) a + a * (b – c) + (b – c) * d

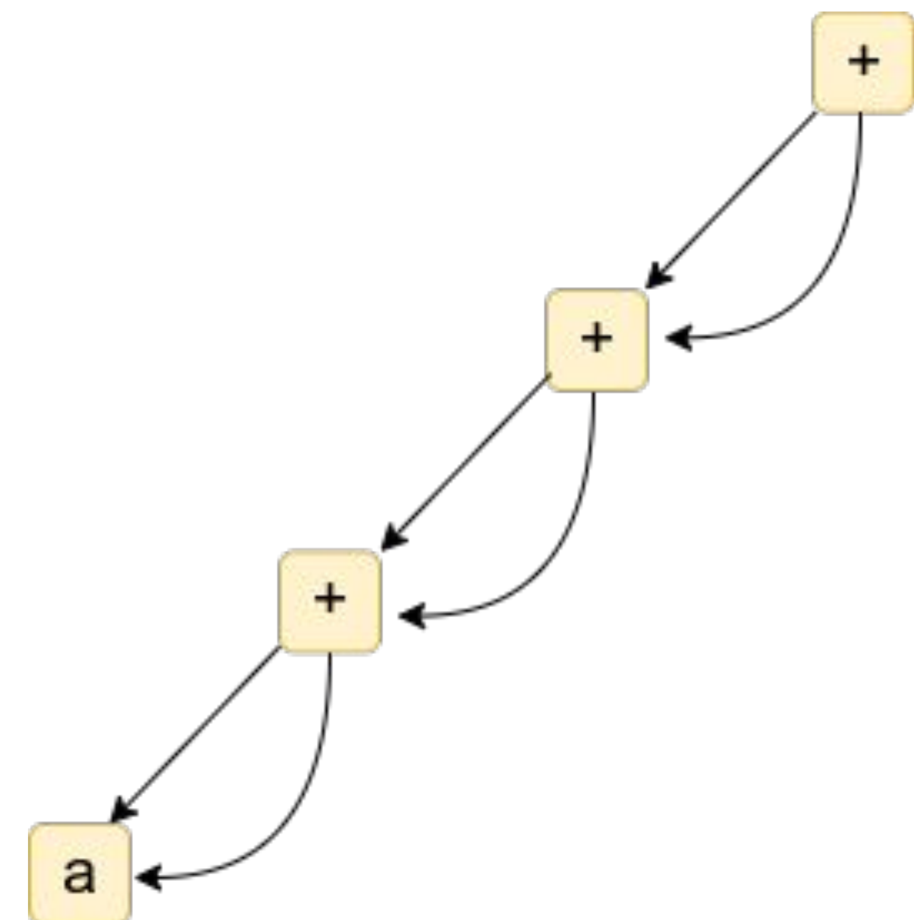




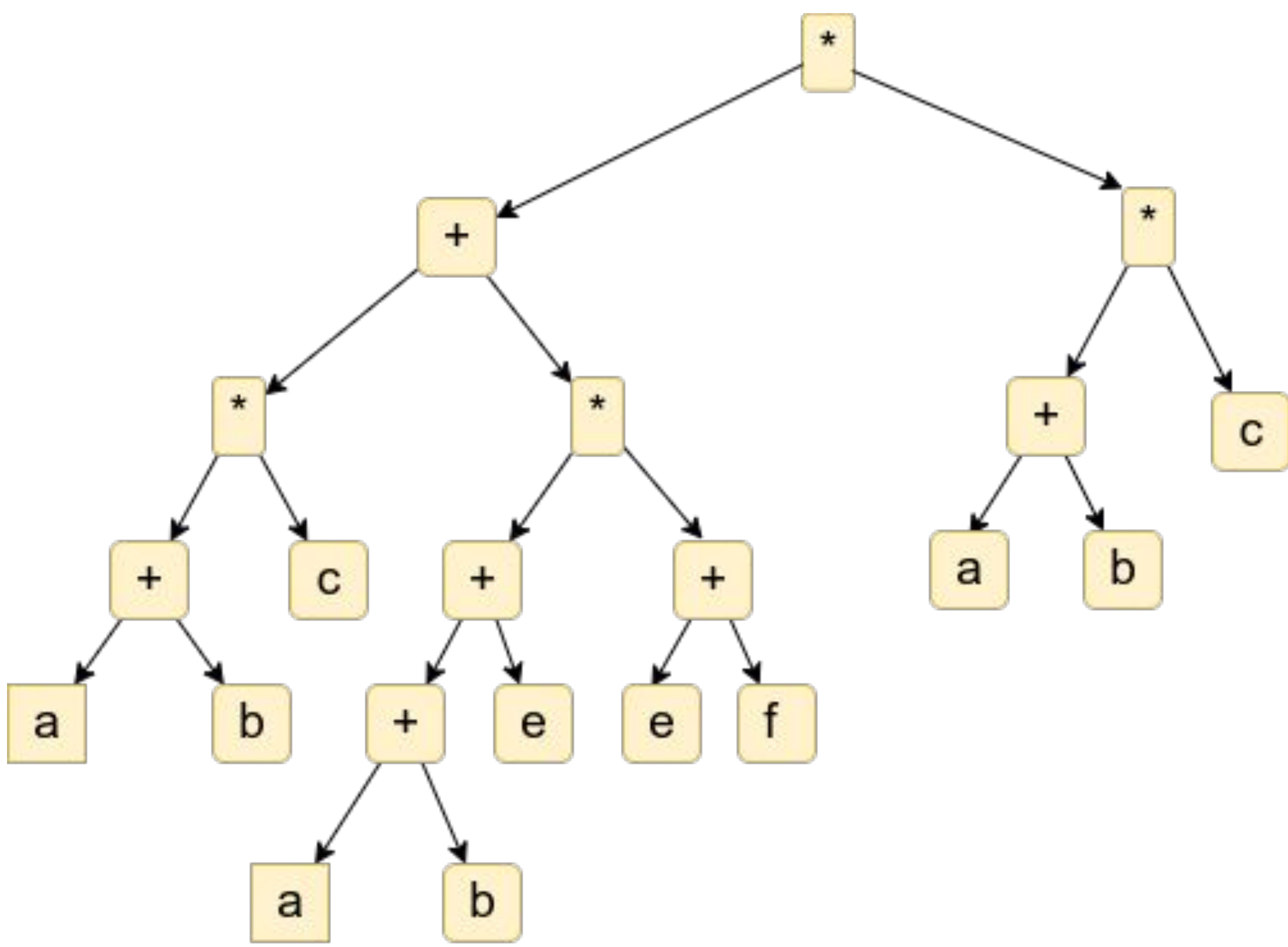
Exercise 2 - Solutions

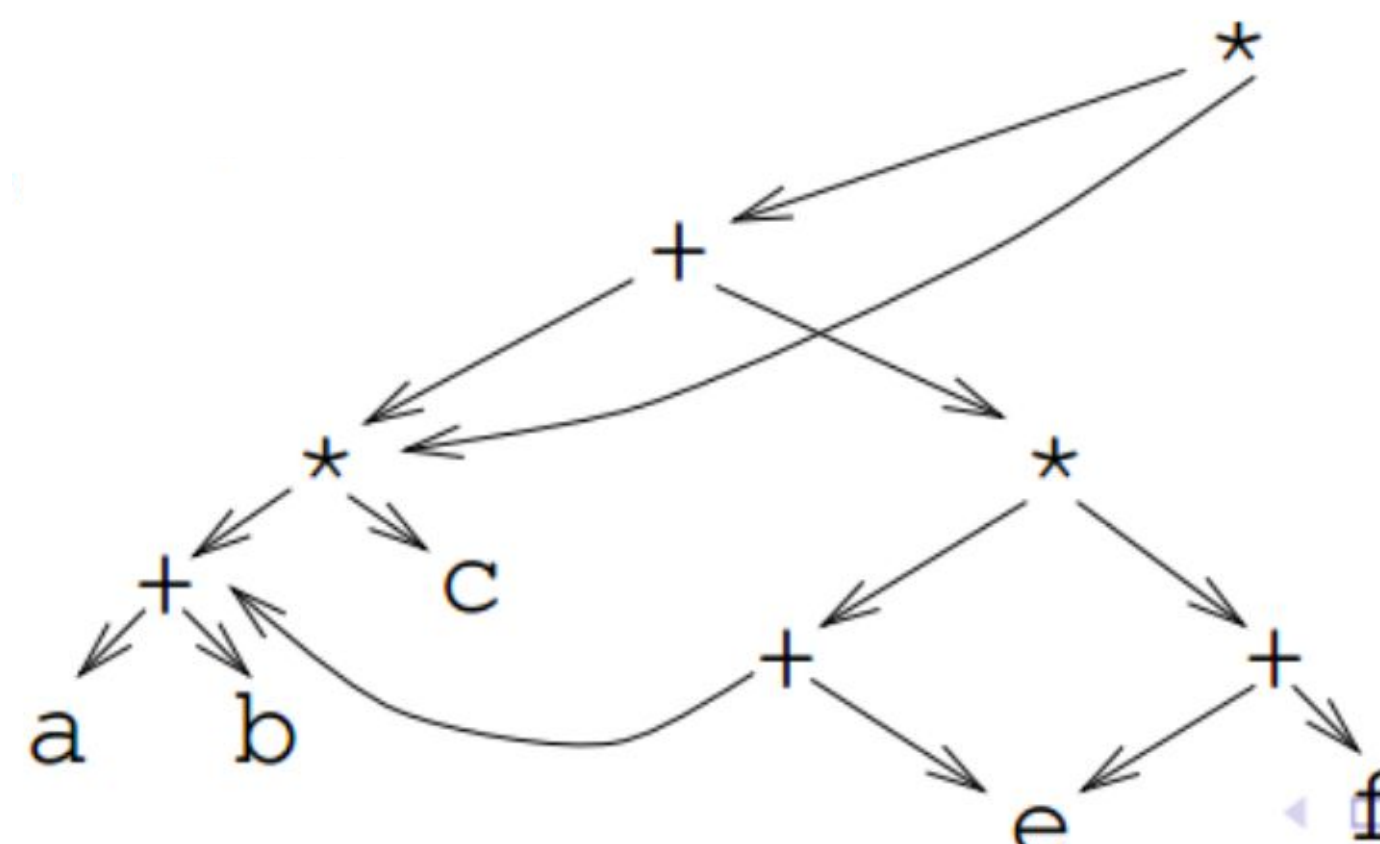
4) (((a + a) + (a + a)) + ((a + a) + (a + a)))





5)[(a + b) * c + ((a + b) + e) * (e + f)] * [(a + b) * c]





THANK YOU

Preet Kanwal
Department of Computer Science & Engineering
preetkanwal@pes.edu

Compiler Design

Preet Kanwal
Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 4: Intermediate Code Generation

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- What is Three-Address Code?
- Format of TAC instructions
- Recap - Address Calculation for 1-D and 2-D arrays
- Example Questions

What is Three Address Code?

- Three-Address Code(TAC) is a Linearized representation of syntax tree or DAG.
- It has at most one operator on RHS of an instruction.
- Each instruction can have up to three addresses.
- The Address can either be a
  - Name (identifier)
  - Constant (number)
  - Temporary (holds an intermediate result)

Format of TAC instructions

The following table represents statements and their corresponding TAC format -

Statement	TAC Format
Assignment Statement	x = y op z (op : Binary operator) x = op y (op : Unary operator)
Copy statement	x = y
Unconditional jumps	goto L
Conditional Jumps	if x goto L ifFalse goto L
Compare and jump	if x relop y goto L ifFalse x relop y goto L

Format of TAC instructions

Statement	TAC Format
Address or Pointers	x = &y z = * x *x = a
Indexed Copy	x[i] = y y = x[i]
Procedure call : foo(a, b, ...)	param a param b ... call (foo, n) where, n is the number of arguments in function foo().
return statement	return y

Exercise 1

Generate Three-Address Code for the following statements -

1) a + b * c – d / b * c
2) x = *p + &y
3) x = f(y+1) + 2
4) x = foo (2 * x + 3, y + 10, g(i), h(3, j))
5) x = f(g(i), h(3, j))
6) alpha = (65 <=c && c<=90) || (97 <= c && c<=122)

Exercise 1.1 - Solution

Given Statements	Three Address Code
a + b * c – d / b * c	t1 = b * c t2 = a + t1 t3 = d / b t4 = t3 * c t5 = t2 - t4

Exercise 1.2 - Solution

Given Statements	Three Address Code
x = *p + &y	t1 = *p t2 = &y t3 = t1 + t2 x = t3

Exercise 1.3 - Solution

Given Statements	Three Address Code
x = f(y+1) + 2	t1 = y + 1 param t1 t2 = call f, 1 t3 = t2 + 2 x = t3

Exercise 1.4 - Solution

Given Statements	Three Address Code	
x = foo (2 * x + 3, y + 10, g(i), h(3, j))	t1 = 2 * x t2 = t1 + 3 param t2 t3 = y + 10 param t3 param i t4 = call g, 1 param t4 param 3 param j	t5 = call h, 2 param t5 t6 = call foo, 4 x = t6

Exercise 1.5 - Solution

Given Statements	Three Address Code
x = f(g(i), h(3, j))	param i t1 = call g, 1 param t1 param 3 param j t2 = call h,2 param t2 t3 = call f, 2 x = t3

Exercise 1.6 - Solution

Given Statements	Three Address Code
alpha = (65 <=c && c<=90) || (97 <= c && c<=122)	t1 = 65 <= c iffalse t1 goto L1 t2 = c <=90 iffalse t2 goto L1 L0 : alpha = true goto next L1 : t3 = 97<=c iffalse t3 goto L3 t4 = c <=122 iffalse t4 goto L3 goto L0 L3 : alpha = false next :

Exercise 2

Generate Three-Address Code for the following function -

```
void main() {
    int x, y;
    int m2 = x * x + y * y;
    while (m2 > 5)
    {
        m2 = m2 – x;
    }
}
```

Exercise 2 - Solution

Given Statements	Three Address Code	

void main() {
int x, y;
int m2 = x * x + y * y;
while (m2 > 5)
{
m2 = m2 – x;
}
}
	void main( )
	{
	int x;
	int y;
	int m2;
	t1 = x * x
	t2 = y * y
	t3 = t1 + t2
	m2 = t3
		L1:
		ifFalse m2 > 5 goto L2
		t4 = m2 - x
		m2 = t4
		goto L1
		L2:

Exercise 3

Generate Three-Address Code for the following code snippet -

```
x = i + 10;
switch(x)
{
    case 1 : x = x * i;
    break;
    case 2 : x = 5;
    case 3 : x = i;
    default: x = 0;
}
```

Exercise 3 - Solution

Given Statements	Three Address Code	
	t1 = i + 10	L4 : if x ==3 goto L5
x = i + 10;	x = t1	goto L6
switch(x)	if x == 1 goto L1	L5 : x = i
{	goto L2	L6 : x = 0
case 1 : x = x * i;	L1 : t2 = x * i	
break;	x = t2	next :
case 2 : x = 5;	goto next	
case 3 : x = i;	L2 : if x ==2 goto L3	
default: x = 0;	goto L4	
}	L3 : x = 5	
	goto L5	

Recap - Address Calculation for 1-D Arrays

Array of an element of an array say A[i] is calculated using the following formula -

Address of A [i] = $A + W * (i – L_B)$

where,

A = Name of the array denotes the Base address

W = Storage Size of one element stored in the array (in bytes)

i = Subscript of element whose address is to be found

L_B = Lower limit of subscript, if not specified assume 0

Compiler Design

Exercise 4

Generate Three-Address Code for the following code snippets -

1) a = b[i]

2)
```
do
    i = i + 1;
while(a[i] < v)
```

3)
```
Product = 0;
i = 1;
do
    Product = Product + A[i] * B[i];
    i = i + 1;
while( i < 20)
```

Exercise 4 - Solutions

Given Statements	Three Address Code
a = b[i]	t1 = 4 * i t2 = b + t1 or t2 = b[t1] a = t2

do
i = i + 1;
while(a[i] < v)
	L1: t1 = i + 1
	i = t1
	t2 = 4 * i
	t3 = a[t2]
	if t3 < v goto L1

Exercise 4 - Solutions

Given Statements	Three Address Code

Product = 0;
i = 1;
do
    Product = Product + A[i] * B[i];
    i = i + 1;
while(i < 20)
	Product =0
	i = 1
	L1 : t1 = 4 * i
	t2 = A[t1]
	t3 = 4 * i
	t4 = B[t3]
	t5 = t2 * t4
	t6 = product + t5
	product = t6
	t7 = i + 1
	i = t7
	if i < 20 goto L1

Recap - Address Calculation for 2-D Arrays

- While storing the elements of 2-D array in memory, elements are allocated a contiguous memory locations.
- A 2-D array must be **linearized** so as to enable their storage.
- There are two ways to achieve linearization -
  - Row-major
  - Column-major

Recap - Address Calculation for 2-D Arrays - Row Major

The address of a location in Row Major System is calculated using the following formula:

$$\text{Address of } A[i][j] = A + W * [N * (i - L_r) + (j - L_c)]$$

where,

N = Number of columns of the given matrix

L_r = Lower limit of row/start row index of matrix, if not given assume 0

L_c = Lower limit of column/start column index of matrix, if not given assume 0

Recap - Address Calculation for 2-D Arrays - Column Major

The address of a location in Row Major System is calculated using the following formula:

Address of A [i][j] = $A + W * [(i - L_r) + M * (j - L_c)]$

where,

N = Number of columns of the given matrix

L_r = Lower limit of row/start row index of matrix, if not given assume 0

L_c = Lower limit of column/start column index of matrix, if not given assume 0

TAC for 2-D Arrays -Assumptions

- Assume all 2-D arrays follow row-major method.
- If the size of array is not mentioned assume it to be m x n array.
- Assume array type as integer and width of an array element as 4 bytes.

Exercise 5

Generate Three-Address Code for the following code snippets -

1)
```
for(i = 0; i < n; i ++)
    for(j = 0; j <n ; j++)
        c[i][j] = 0;
```
where c is a 5x5 array

2)
```
for (i=1; i<=10 ; i++)
    for(j = 1; j <= 10; j++)
        A[i][ j]=  A[i][j] + B[i] [j];
```
where A and B are 10x10 arrays of type float, assume the arrays are 1-indexed.

Exercise 5.1 - Solution

1) for(i = 0; i < n; i ++)

    for(j = 0; j <n ; j++)

        c[i][j] = 0;

where c is a 5x5 array

Address calculation for c[i][j]

$$c[i][j] = B + W * [\ N * (\ i - L_r\ ) + (\ j - L_c\ )\ ]$$

$$= c + 4 * [\ n * (i - 0) + (\ j - 0)\ ]$$

$$= c + 4 * (\ 5 * i + j\ )$$

Exercise 5.1 - Solution

Given Statements	Three Address Code	
	i = 0	c[t5] = 0
for(i = 0; i < n; i ++)	L0: t1 = i < n	t6 = j + 1
for(j = 0; j <n ; j++)	if t1 goto L1	j = t6
c[i][j] = 0;	goto next	goto L4
	L1 : j=0	L3 : t7 = i + 1
	L4 : t2 = j < n	i = t7
	if t2 goto L2	goto L0
	goto L3	
	L2 : t3 = 5 * i	
	t4 = t3 + j	
	t5 = 4 * t4	

Compiler Design

Exercise 5.1 - Solution

Given Statements	Three Address Code		

for (i=1; i<=10 ; i++)
for(j = 1; j <= 10; j++)
C[i][j]= A[i][j] + B[i] [j];
 	
where A, B, C are 10x10 arrays of type float
	i = 1
	L5 : t1 = i <=10
	if t1 goto L1
	goto next
	L1 : j = 1
	L 4 : t2 = j <=10
	if t2 goto L2
	goto L3 //inc i
	
	L2 : t1 = i - 1
	t2 = 10 * t1
	t3 = j - 1
	t4 = t2 + t3
	t5 = 8 * t4
	t6 = A[t5]
		t7 = i - 1
		t8 = 10 * t7
		t9 = j - 1
		t10 = t8 + t9
		t11 = 8 * t10
		t12 = B[t11]
		t13 = t6 + t12
		t14 = i - 1
		t15 = 10 * t14
		t16 = j - 1
		t17 = t15 + t16
		t18 = 8 * t17
		c[t18] = t13
			t19 = j + 1
			j = t19
			goto L4
			
			L3 : t20 = i + 1
			i = t20
			
			goto L5
			
			next :

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

Compiler Design

Preet Kanwal
Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 4: Three-Address Code

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- Data Structures for Three-Address Code
  - Quadruples
  - Triples
  - Indirect Triples
  - Example Questions

Data Structures for TAC

- Three address code is represented as a record structure with fields for operator and operands.
- These records can be stored as an array or a linked list.
- There three types of record structures -
  1. Quadruples [4 fields]
  2. Triples [3 fields]
  3. Indirect Triples [Triples + List of pointers to Triples]

Quadruples

- A Quadruple is an array type data structure with 4 fields -

op	arg1	arg2	result

where,

op - operator.

arg1, **arg2** - the two operands used.

result - the result of the expression.

Quadruples

op	arg1	arg2	result

- arg1, arg2 and result are pointers to symbol table entries.
- This means even temporaries must be placed in symbol table as they are created.
- Any unused field is left blank/NULL
- Disadvantage - Temporary names have to be entered into symbol table.

Quadruples Format - Unary Operators

The given table describes the quadruple format for unary operators -

Statement	op	arg1	arg2	result
Unary operators - arg2 is empty	op	arg1	null	arg2
Example: x=-y	-	y	null	x
Example: x=y	=	y	null	x

Quadruples Format - Functions

The given table describes the quadruple format for functions -

Statement	op	arg1	arg2	result
param operator - arg2 and result are empty	param	arg1	null	null
Example: param x	param	x	null	null
Function Call - call func_name, func_param	call	func_name	value	x
Example: call foo,3	call	foo	3	null
Example: x = call foo,3	call	foo	3	x

Quadruples Format - Jumps

The given table describes the quadruple format for jumps -

Statement	op	arg1	arg2	result
For unconditional jumps - result is label	goto	null	null	label
conditional jump Example - if x goto L	if	x	null	L
conditional jump Example - ifFalse x goto L	ifFalse	x	null	L

Quadruples Format - Labels and return

The given table describes the quadruple format for labels and return statements -

Statement	op	arg1	arg2	result
Label generation Example - L1:	Label	null	null	L1
return	return	x	null	null
return x	return	x	null	null

Quadruples Format - Array indexing

The given table describes the quadruple format for array indexing -

Statement	op	arg1	arg2	result
x[i] = y	[]=	x	i	y
	STAR	x	i	y
x = y[i]	[]=	y	i	x
	LDAR	y	i	x

Exercise 1

Write the Three-Address Code and corresponding Quadruple representation for the following code snippet -

```
if x == 0
        u = 1;
else
    u = fact(x – 1) * x;
value = u;
```

Exercise 1 - Solution

Three-Address Code -

```
if x == 0
        u = 1;
else
    u = fact(x – 1) * x;
```

```
t1 = x == 0
ifFalse t1 goto L1
u = 1
goto L2
L1:
t2 = x – 1
param t2
t3 = call fact, 1
t4 = t3 * x
u = t4
L2:
```

Exercise 1 - Solution

Quadruple -

t1 = x == 0

ifFalse t1 goto L1

u = 1

goto L2

L1:

t2 = x – 1

param t2

t3 = call fact, 1

t4 = t3 * x

u = t4

L2:

op	arg1	arg2	result
==	x	0	t1
ifFalse	t1		L1
=	1		u
goto			L2
Label			L1
-	x	1	t2
param	t2		
call	fact	1	t3
*	t3	x	t4
=	t4		u
Label			L2

- A Triple is an array type data structure with 3 fields -

op	arg1	arg2

where,

op - operator.

arg1, **arg2** - the two operands used.

Triples

- Triples are alternative ways for representing syntax tree or Directed acyclic graph.
- Triples avoid entering temporary names into symbol table.
- For a temporary, use serial number of statement computing its value.
- Problem: Code Immovability
  - No temporary variables stored in symbol table
  - All references are only to the position of statement and not location.
  - This requires the compiler to change all references to arg1 and arg2.
  - Thus, triples are not very efficient in optimizing compilers.

Triples Format - Jumps and Label

The given table describes the triple format for jumps and label -

Statement	op	arg1	arg2
Unconditional jumps	goto	(2)	
conditional jump Example - if x goto L	if	x	(2)
conditional jump Example - ifFalse x goto L	ifFalse	x	(2)
Label	Label		

Triples Format - Array indexing

The given table describes the triple format for array indexing -

Statement	Stmt no.	op	arg1	arg2
x[i] = y	(0)	[]=	x	i
	(1)	=	(0)	y
x = y[i]	(0)	=[]	y	i
	(1)	=	x	(0)

Exercise 2

Write the Triple representation for the following Three-Address Code -

t1 = -b

t2 = t1 * d

t3 = t1 + c

t4 = -b

t5 = t4 * d

t6 = t3 +t5

a = t6

Exercise 2 - Solution

t1 = -b

t2 = t1 * d

t3 = t1 + c

t4 = -b

t5 = t4 * d

t6 = t3 +t5

a = t6

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

The value of a temporary variable can be accessed by the position of the statement that computes it.

Indirect Triples

- A separate list of pointers to the triple structure (i.e, statement numbers) is maintained.
- The statements can be moved by reordering the statement list.
- The utility of indirect triples is almost the same as that of quadruples, but requires less space.

Exercise 2 (cont.)

Write the Indirect Triple representation for the following Three-Address Code -

t1 = -b

t2 = t1 * d

t3 = t1 + c

t4 = -b

t5 = t4 * d

t6 = t3 +t5

a = t6

Exercise 2 (cont.)

t1 = -b

t2 = t1 * d

t3 = t1 + c

t4 = -b

t5 = t4 * d

t6 = t3 +t5

a = t6

	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

No change in the Structure

Advantage of Indirect Triples

Suppose the code changes to -

t1 = -b

t2 = t1 * d

t3 = t1 + c

t5 = t4 * d

t6 = t3 +t5

a = t6

	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(10)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
~~(13)~~	~~-~~	~~b~~	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

No change in the Structure

Compiler Design

Exercise 3

Write the Quadruple and Triple representation for the following code snippets -

1) a = b[i] + c[j]

2) x = f(y + 1) + 2

3) X[i] = a * c + y[i] – n[j] / v

4) for(j=0; j<=10; j++)

```
{
a = a * (j* (b/c));
}
```

Compiler Design

Exercise 3 - Solutions

1) a = b[i] + c[j]

Intermediate Code -

t1 = 4 * i

t2 = b[t1]

t3 = 4 * j

t4 = c[t3]

t5 = t2 + t4

a = t5

Quadruples

op	arg1	arg2	res
*	4	i	t1
=[ ]	b	t1	t2
*	4	j	t3
=[ ]	c	t3	t4
+	t2	t4	t5
=	t5		a

Triples

Stmt No.	op	arg1	agr2
1	*	4	i
2	=[ ]	b	(1)
3	*	4	j
4	=[ ]	c	(3)
5	+	(2)	(4)
6	=	a	(5)

Compiler Design

Exercise 3 - Solutions

2) x = f(y + 1) + 2

3-addr stmt	Quadruple Format					Triple Format			Indirect Triple Format					
	op	arg1	arg2	result	Stmt#	op	arg1	arg2	Ptr	Stmt#	Stmt#	op	arg1	arg2
T1 = y + 1	+	y	1	T1	1	+	y	1	11	1	1	+	y	1
Param T1	Param	T1			2	param	(1)		12	2	2	param	<11>	
T2 = call f, 1	call	f	1	T2	3	call	f	1	13	3	3	call	f	1
T3 = T2 + 2	+	T2	2	T3	4	+	(3)	2	14	4	4	+	<13>	2
X = T3	=	T3		X	5	=	X	(4)	15	5	5	=	X	<14>

Exercise 3 - Solutions

3) X[i] = a * c + y[i] – n[j] / v

	Quadruple Format				Triple Format				Indirect Triple Format					
3-addr stmt	op	arg1	arg2	result	Stmt#	op	arg1	arg2	Ptr	Stmt#	Stmt#	op	arg1	arg2
T1 = a * c	*	a	c	T1	1	*	a	c	111	1	1	*	a	c
T2= 4 * I	*	4	I	T2	2	*	4	I	112	2	2	*	4	I
T3 = y[T2]	=[ ]	y	T2	T3	3	=[ ]	y	(2)	113	3	3	=[ ]	y	<112>
T4 = T1 + T3	+	T1	T3	T4	4	+	(1)	(3)	114	4	4	+	(1)	<113>
T5 = 4 * j	*	4	j	T5	5	*	4	j	115	5	5	*	4	j
T6 = n[T5]	=[ ]	n	T5	T6	6	=[ ]	n	(5)	116	6	6	=[ ]	n	<115>
T7 = T6/v	/	T6	v	T7	7	/	(6)	v	117	7	7	/	<116>	v
T8 = T4 - T7	-	T4	T7	T8	8	-	(4)	(7)	118	8	8	-	<114>	<117>
T9 = 4 * i	*	4	I	T9	9	*	4	I	119	9	9	*	4	I
X[T9] = T8	[ ]=	X	T9	T8	10	[ ]=	X	(9)	120	10	10	[ ]=	X	<119>
					11	=	(10)	(8)	121	11	11	=	<120>	<118>

4) for(j=0; j<=10; j++){ a = a * (j* (b/c));}

j = 0
L1:
t1 = j <= 10
ifFalse t1 goto L2
t2 = b / c
t3 = j * t2
t4 = a * t3
a = t4
t5 = j + 1
j = t5
goto L1
L2 :

Quadruples

op	arg1	arg2	res
=	0		j
Label			L1
<=	j	10	t1
ifFalse	t1		L2
/	b	c	t2
*	j	t2	t3
*	a	t3	t4
=	t4		a
+	j	1	t5
=	t5		j
goto			L1
Label			L2

Triples

Stmt No.	op	arg1	agr2
1	=	j	0
2	Label		
3	<=	j	10
4	ifFalse	(3)	(12)
5	/	b	c
6	*	j	(5)
7	*	a	(6)
8	=	a	(7)
9	+	j	1
10	=	j	(9)
11	goto	(2)	
12	Label		

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 4: Static Single-Assignment(SSA)

Preet Kanwal
Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- Static Single-Assignment (SSA) Form
- ϕ-function
- ϕ-function Examples

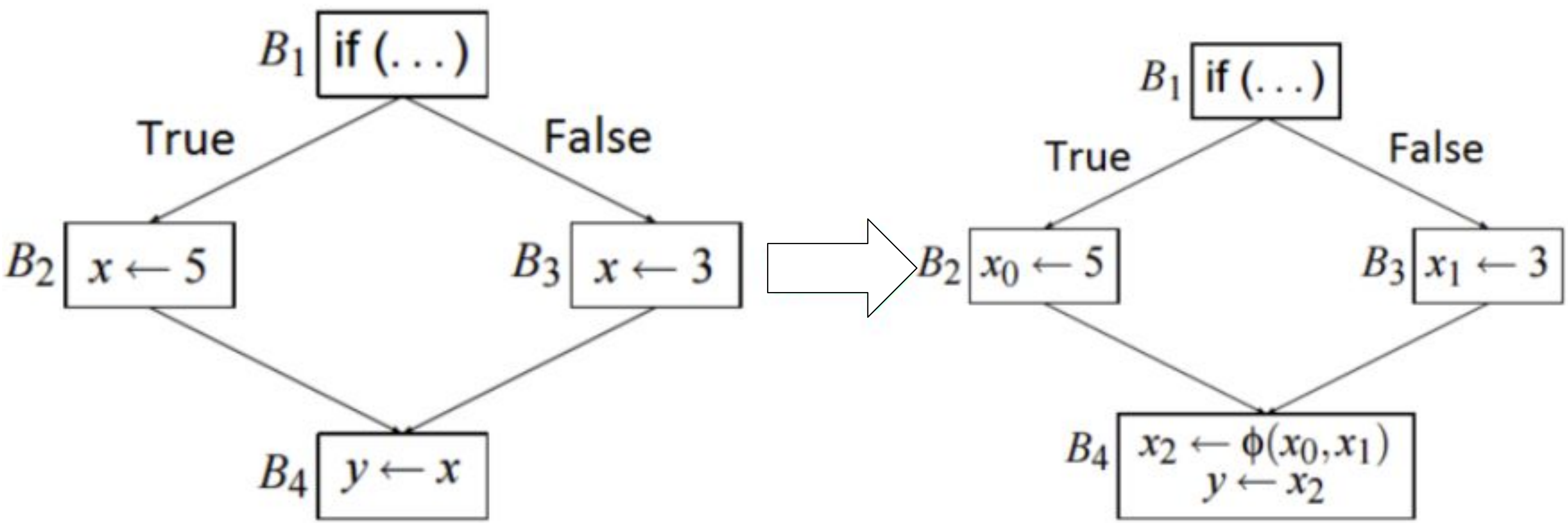
Static Single Assignment (SSA) Form

- Each variable is assigned exactly once but may be used multiple times.
- Existing variables in the original IR are split into versions:
- New version of variable is typically indicated by the original name with a subscript, so that every definition gets its own version.
- SSA is an intermediate form widely used by modern optimizing compilers.

ϕ-function

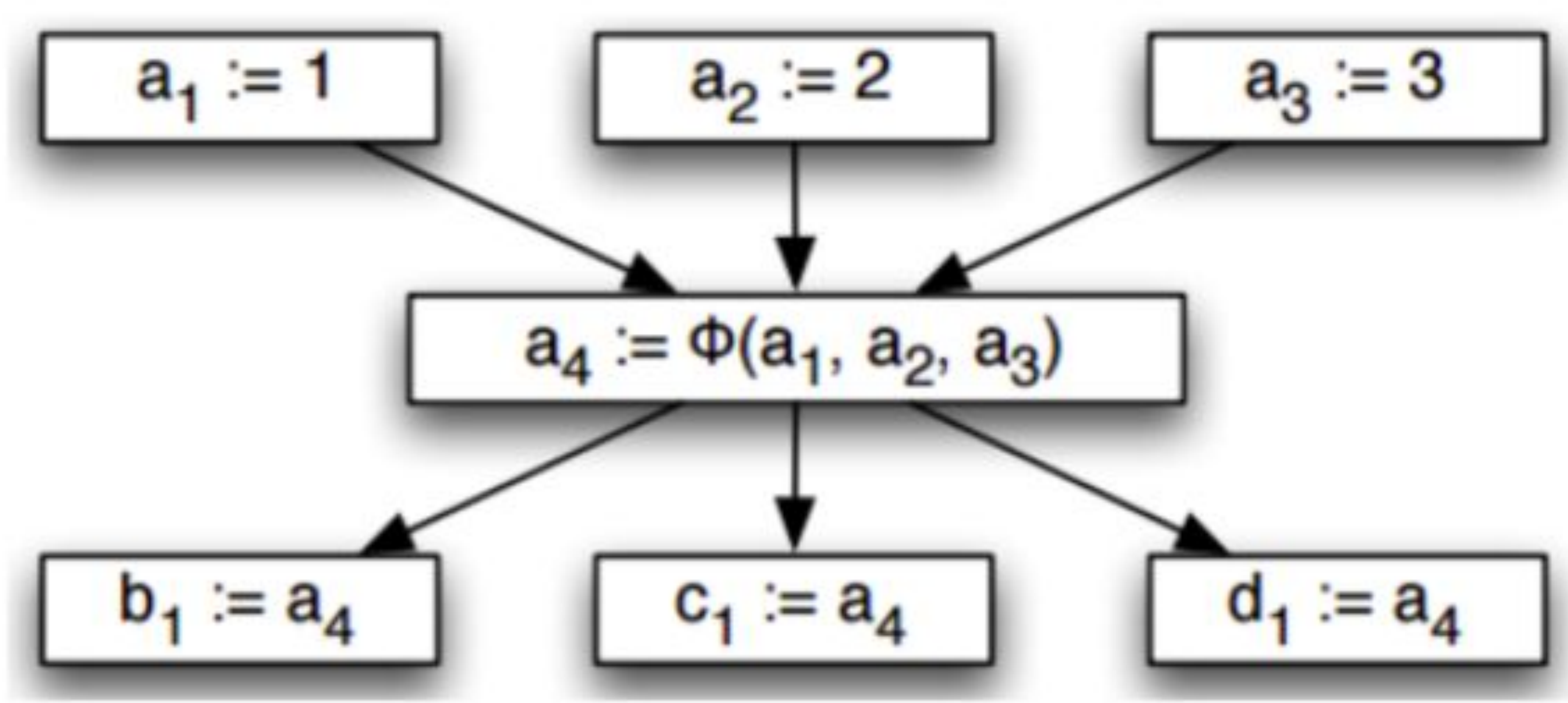
- Control flow can't be predicted in advance, so we can't always know which definition of a variable reached a particular use.
- To handle this uncertainty, we create ϕ functions.
- Notation represents natural "meet points" where values are combined.
- No. of arguments to ϕ(a1, a2 ....) is the number of incoming flow edges.
- Return value of the function corresponds to the control-flow path taken to get to the statement.

ϕ-function - Example 1

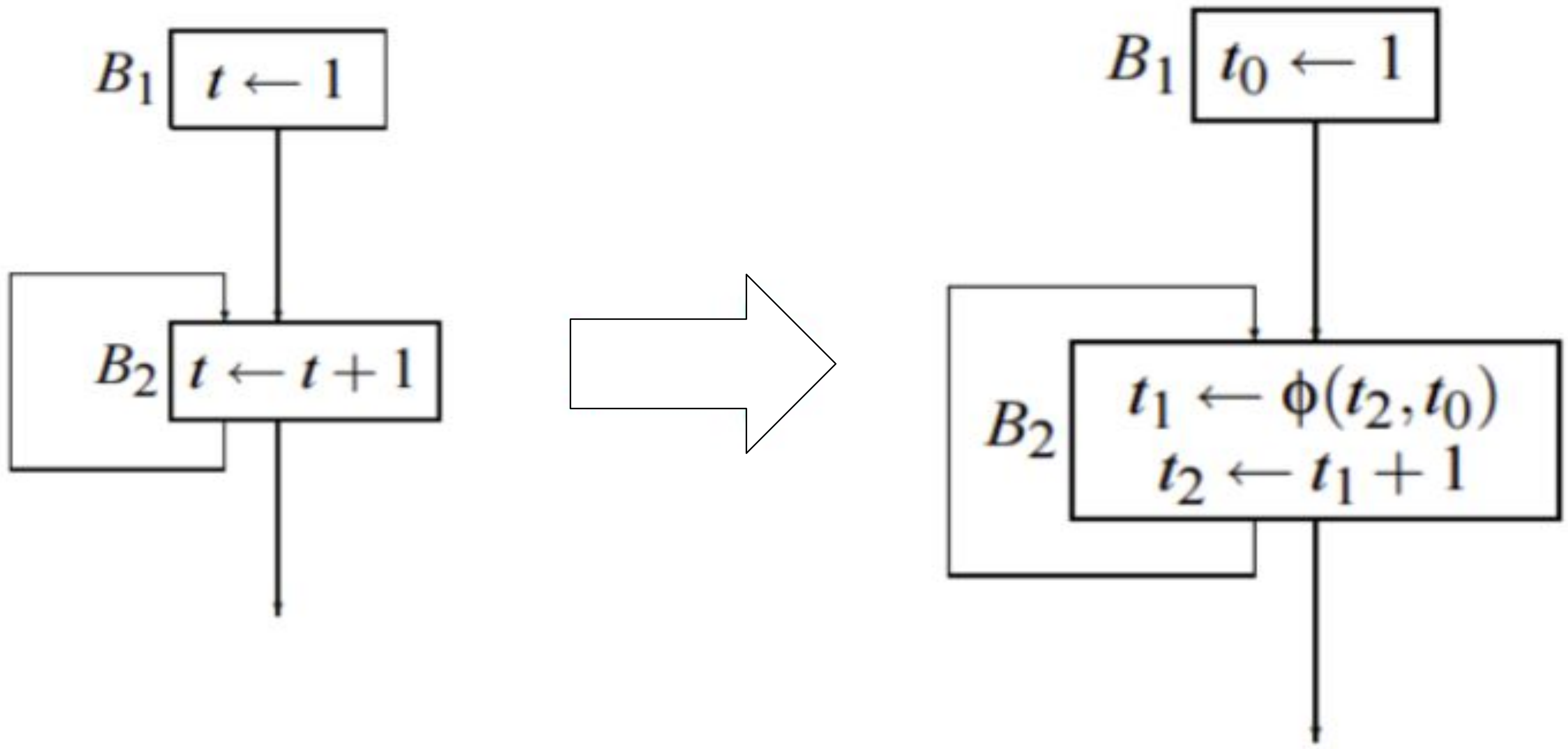


ϕ-function - Example 2

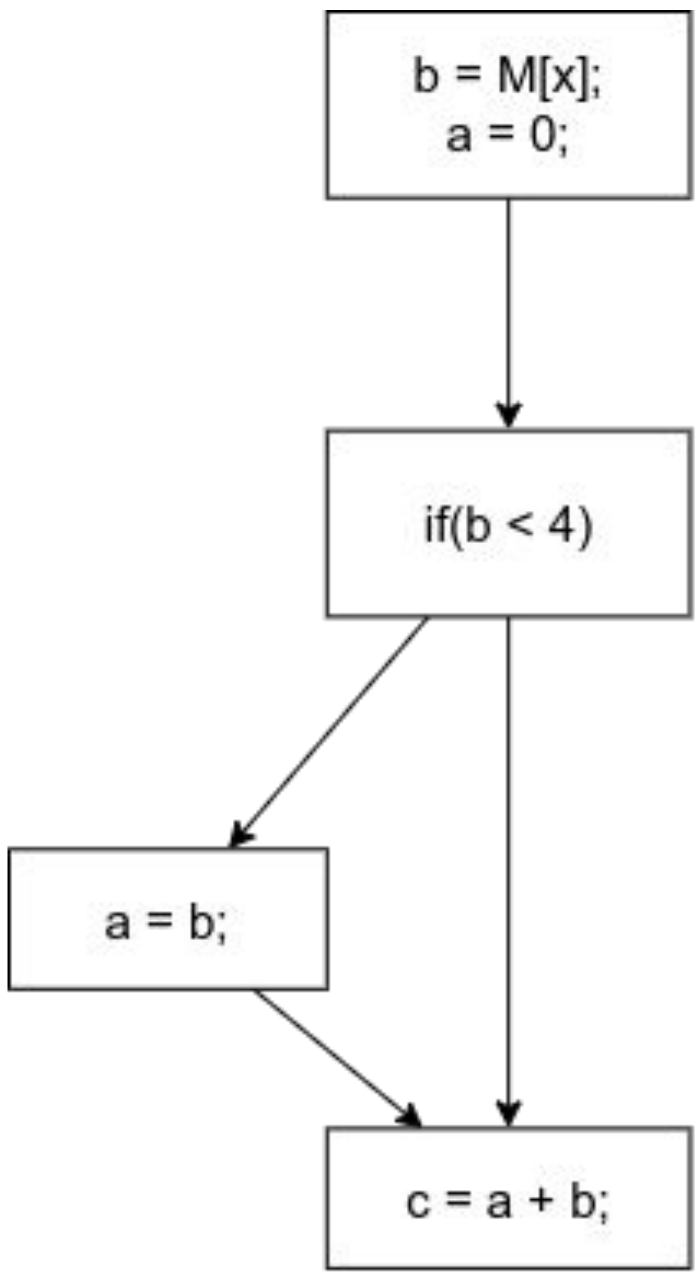
```
case (...) of
    0: a := 1;
    1: a := 2;
    2: a := 3;
end
case (...) of
    0: b := a;
    1: c := a;
    2: d := a;
end
```



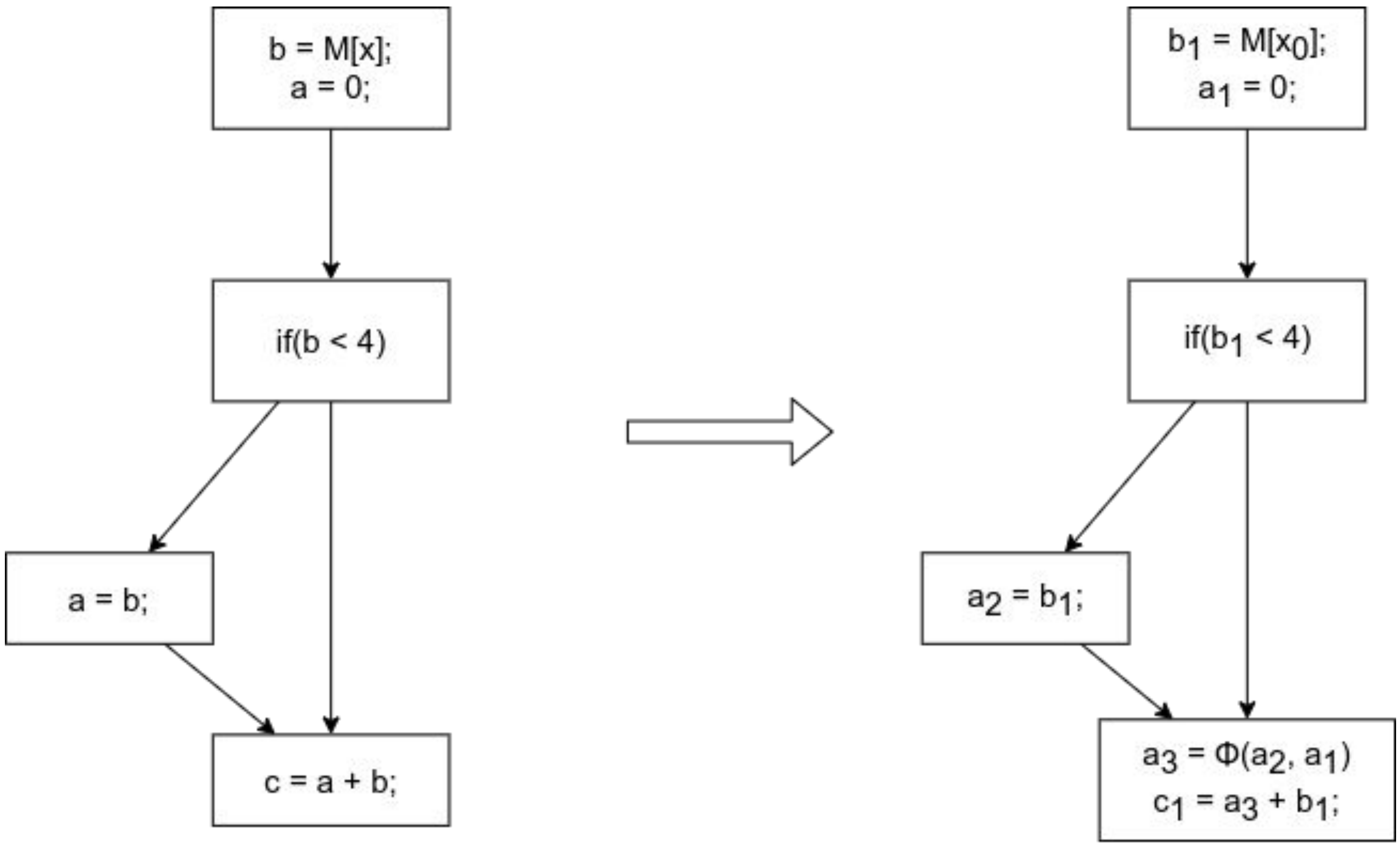
ϕ-function - Example 3



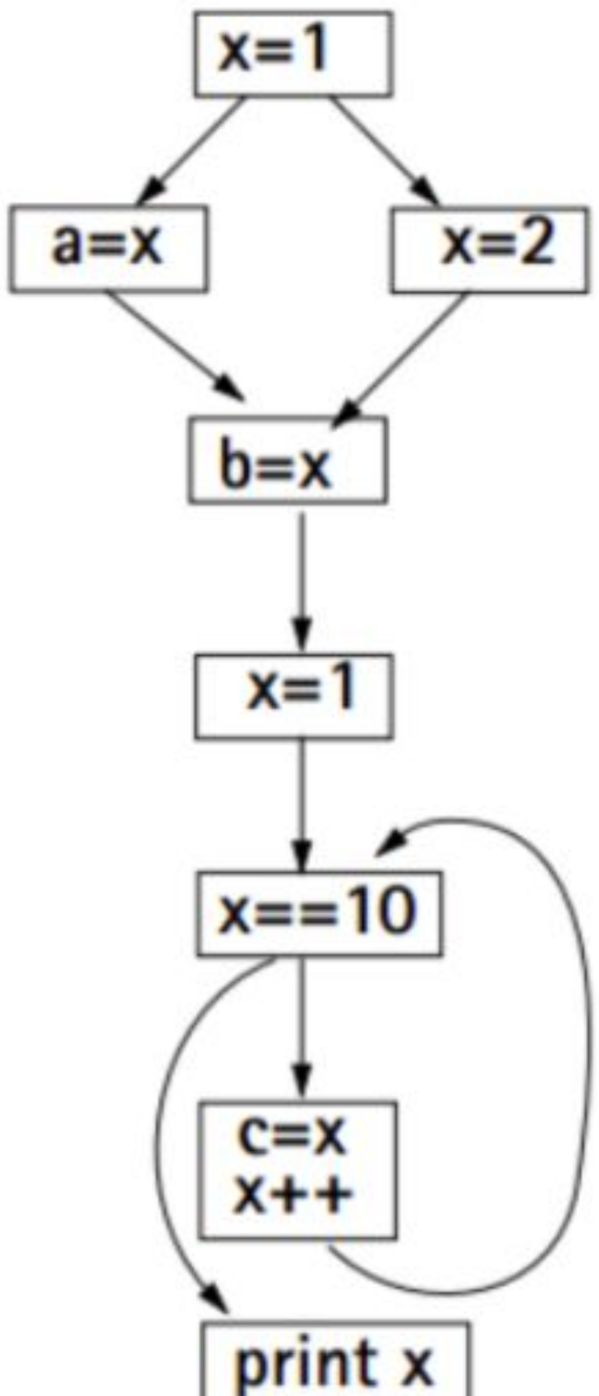
ϕ-function - Example 4



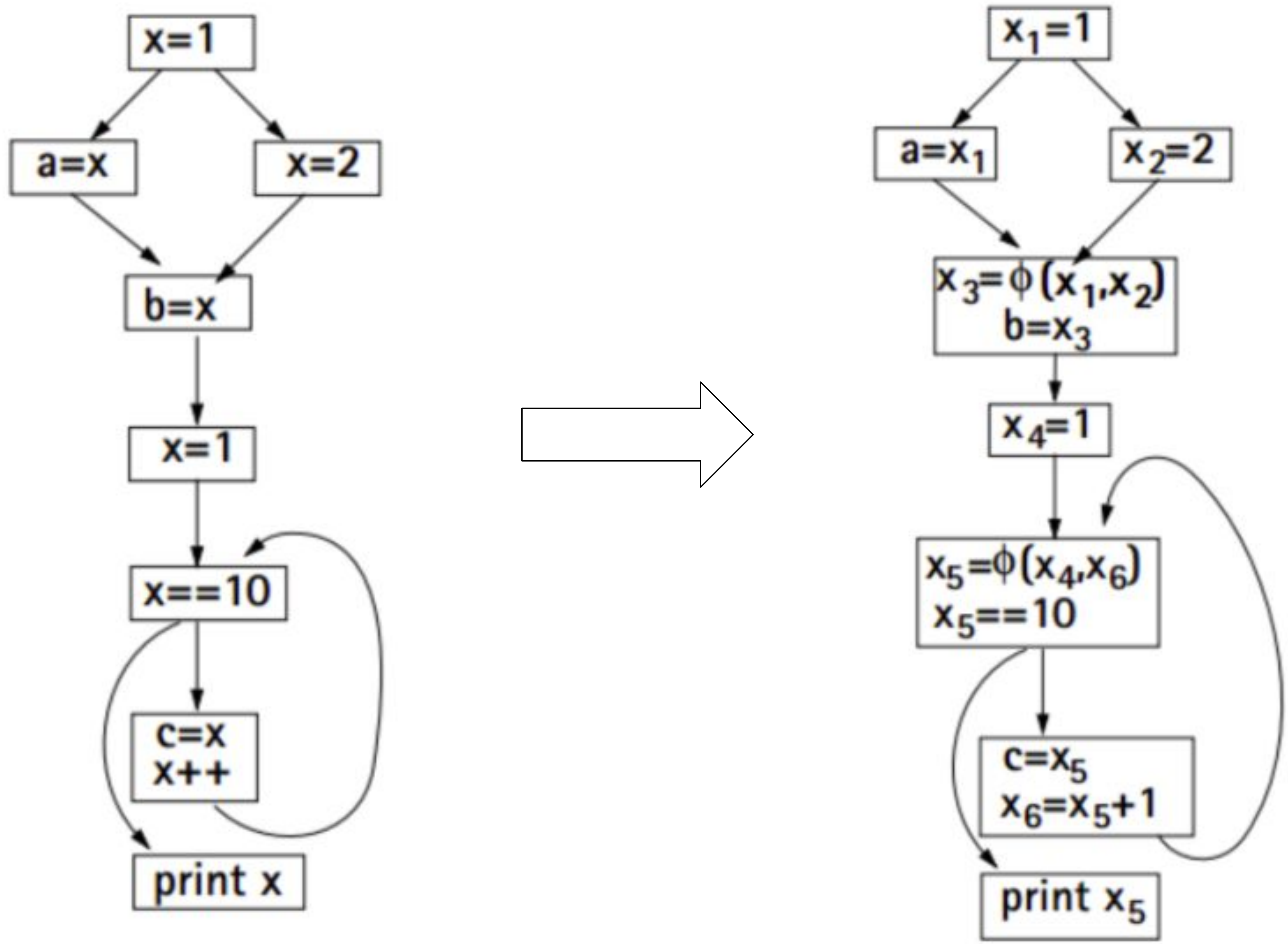
ϕ-function - Example 4



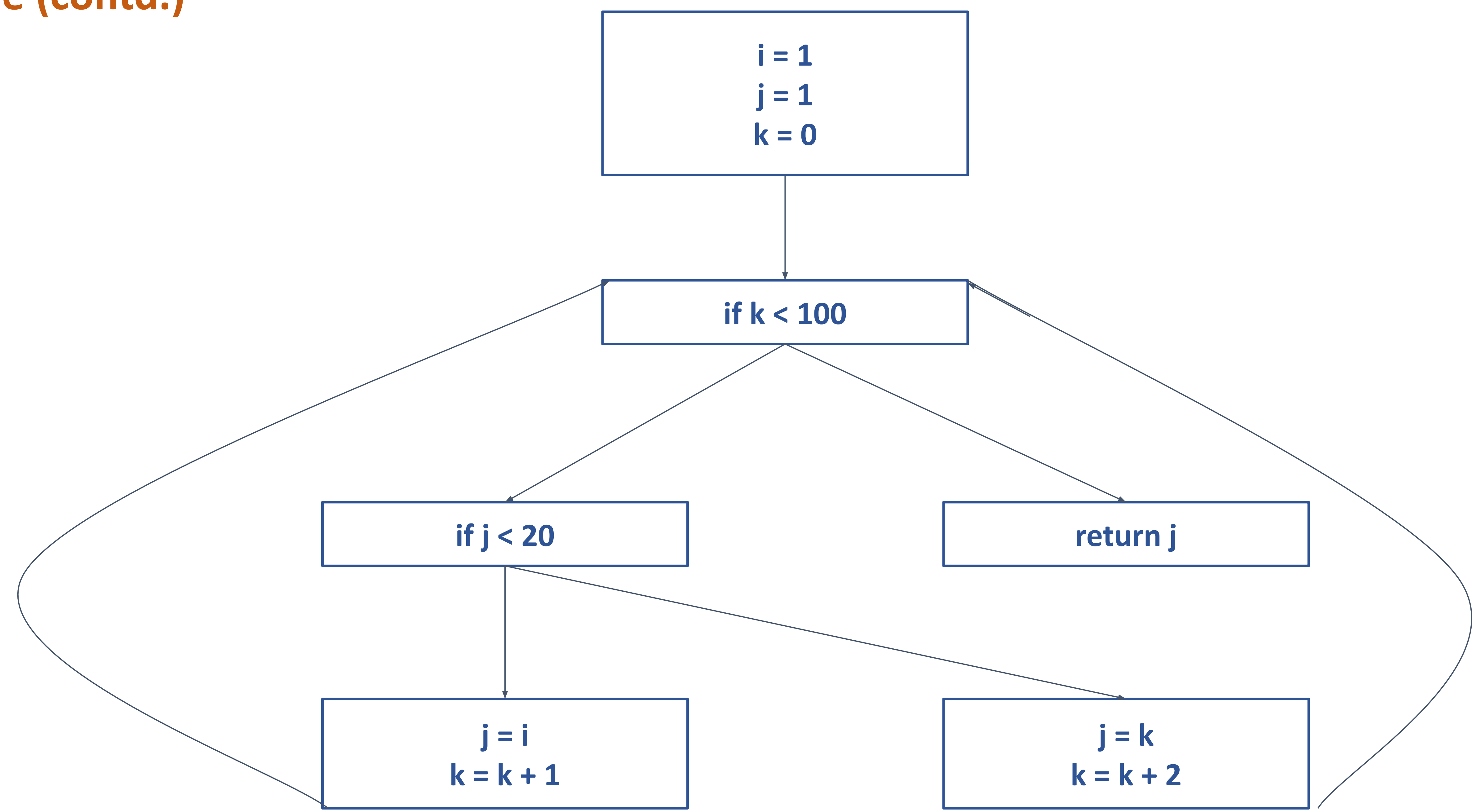
ϕ-function - Example 5



ϕ-function - Example 5



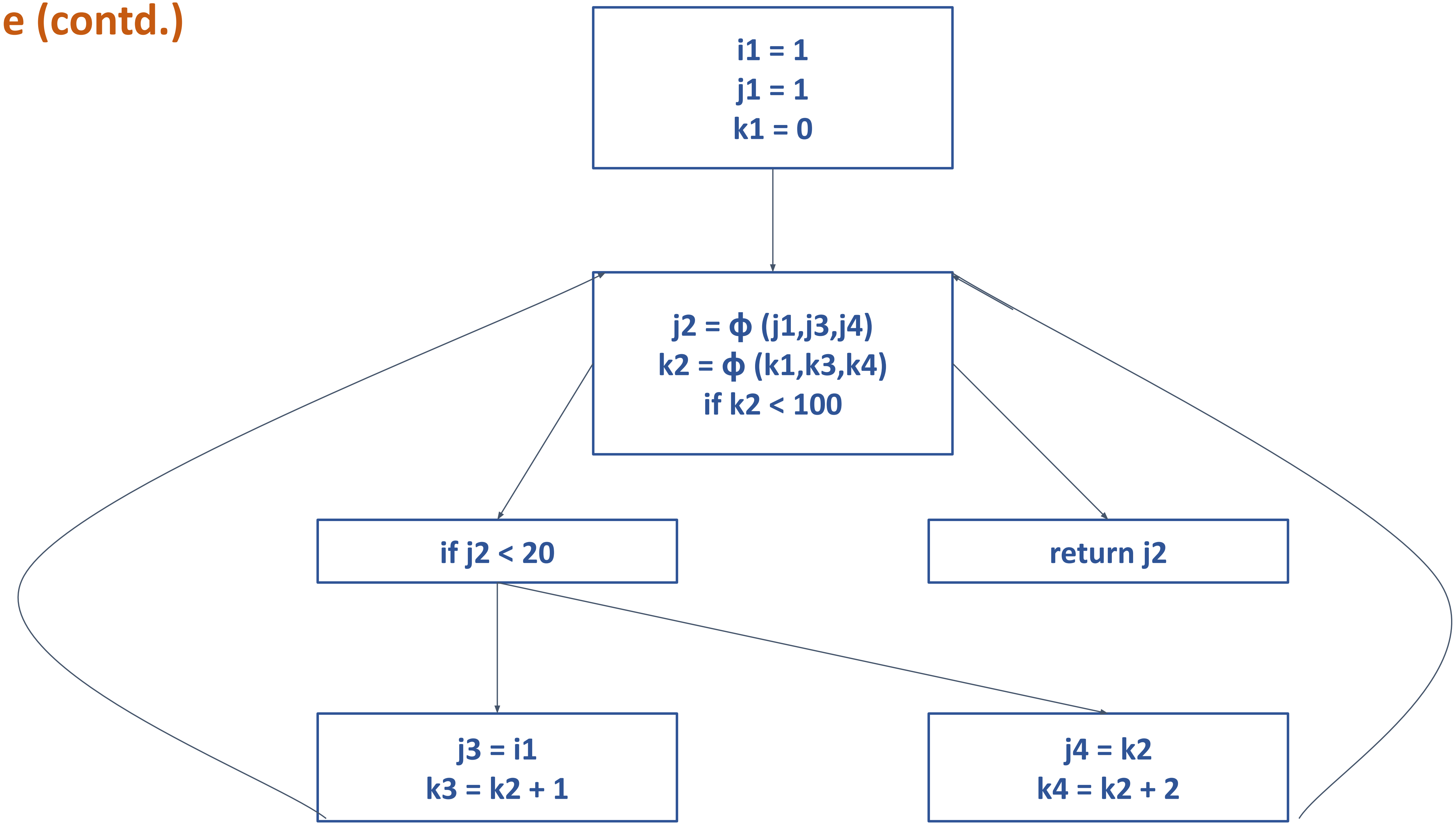
Example (contd.)



ϕ-function - Example 6

Example (contd.)

SSA



ϕ-function - Example 7

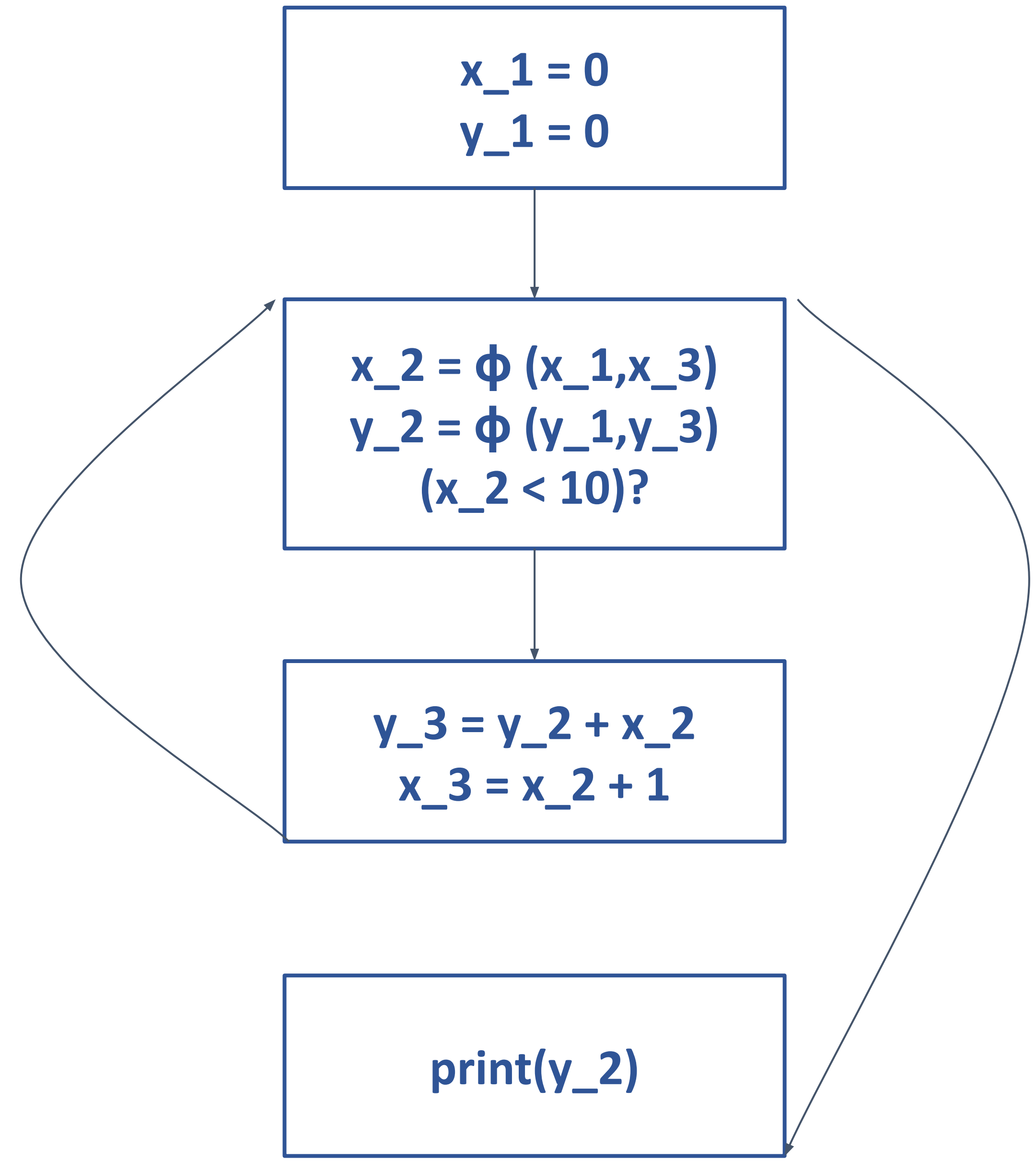
Example

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

ϕ-function - Example 7

Example

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



Note -

- Most modern production compilers use SSA form (eg. gcc, suif, llvm, hotspot etc.)
- Popular compiler optimizations (eg. constant propagation) become easier to write (and in some cases, algorithmically faster) when applied to programs in SSA form.
- Conversion to SSA form introduces a lot of assignments - compilers that do this need to have good register allocators that can eliminate most of them again (not a concern these days).

THANK YOU

Preet Kanwal
Department of Computer Science & Engineering
preetkanwal@pes.edu

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Sree Pranavi G

Compiler Design

Unit 4: Control Flow Graph Generation

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- CFG Generation
- Converting program to SSA
- Basic blocks
- Generate TAC

CFG Representation of Intermediate Code

- A flow graph is graphical representation that exhibits flow of control information.

- Helps in performing machine independent optimization.

- Benefits of code generation:
  - Better register allocation.
  - Better instruction selection.
  - Helps reduce program cost.

Rules for constructing flow graph

- The nodes of the flow graph are the basic blocks.
- Number the nodes(For example:B1,B2).
- The first basic block (i.e B1)is called initial block.
- Draw a directed edge from the initial block i.eB1to the next block following B1 i.e B2if B2 immediately follows B1.

CFG Representation of Intermediate Code

- Partition intermediate code into **basic blocks.**

- Nodes of FG : Basic blocks

- Add nodes :
  - Entry node (edge to first block)
  - Exit node (edge from last block)

- Edges of FG : indicate which blocks can follow other blocks. (determine predecessor and successor of a Block).

Basic Blocks

- A basic block consists of set of statements which are executed sequentially without branching.

- Maximal sequence of consecutive instructions such that,
  - Flow of control can only enter the basic block from the first instruction
  - Control leaves the block only at the last instruction

- Each instruction is assigned to exactly one basic block.

Rules for determining basic blocks

- Identify leaders
  - The first three address instruction in the intermediate code is a leader.
  - Any instruction that is the target of a conditional or unconditional jump is a leader.
  - Any instruction that immediately follows a conditional or unconditional jump is a leader.

- The first instruction in the basic block is a leader and the basic block ends just before another leader instruction.

Example

1. i = 1
2. j = 1
3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 0.0
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. t5 = i - 1
14. t6 = 88 * t5
15. a[t6] = 1.0
16. i = i +1
17. if i <= 10 goto (13)

Example (contd.)

1. i = 1
2. j = 1
3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 0.0
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. t5 = i - 1
14. t6 = 88 * t5
15. a[t6] = 1.0
16. i = i +1
17. if i <= 10 goto (13)

First instruction in IC is a leader

Any Instruction that is the Target of a conditional or unconditional jump is a Leader

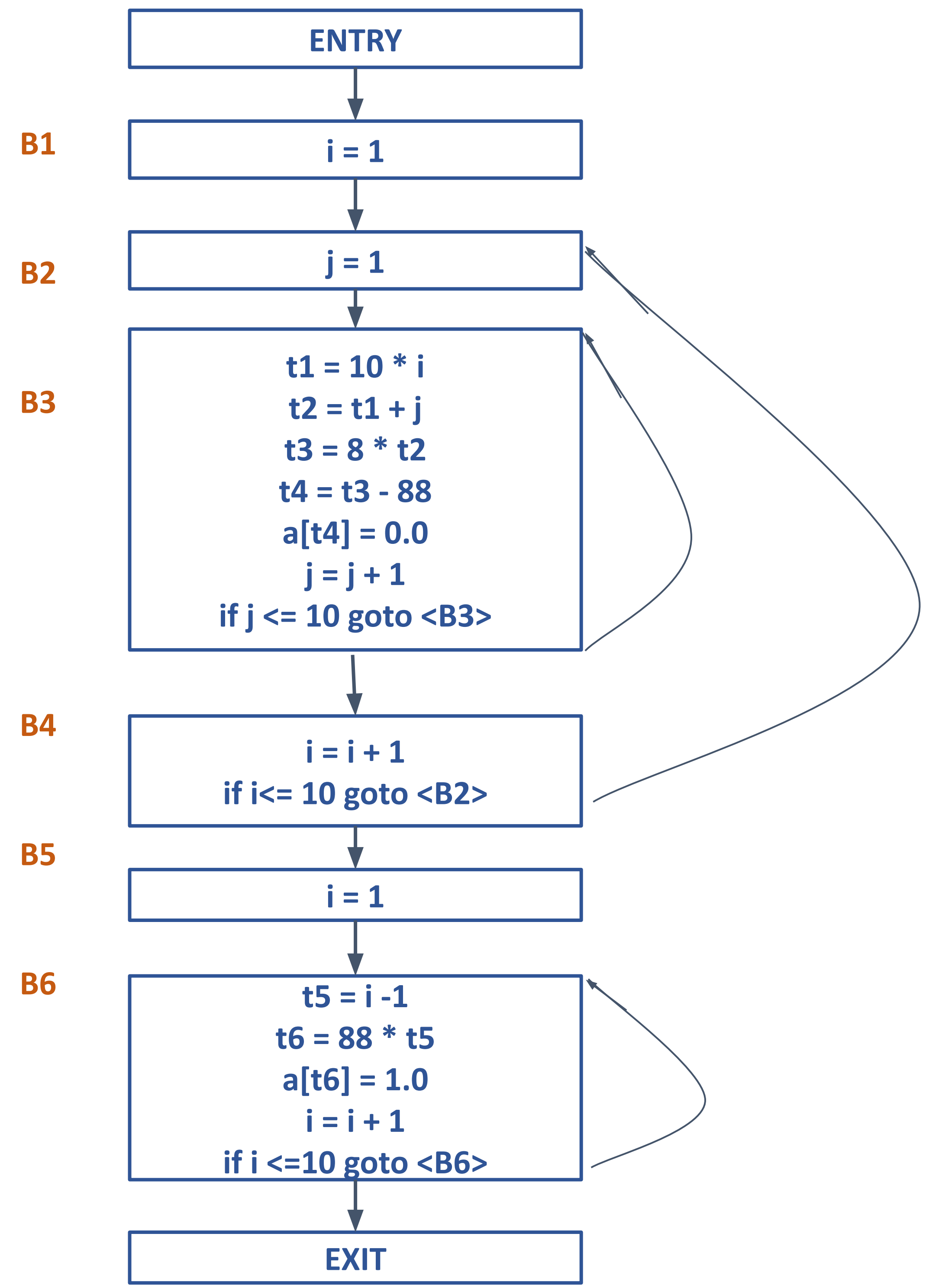
Any Instruction that follows a conditional or unconditional jump is a Leader

Example (contd.)

1. <u>i = 1</u>
2. <u>j = 1</u>
3. <u>t1 = 10 * i</u>
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 - 88
7. a[t4] = 0.0
8. j = j + 1
9. if j <= 10 goto (3)
10. <u>i = i + 1</u>
11. if i <= 10 goto (2)
12. <u>i = 1</u>
13. <u>t5 = i - 1</u>
14. t6 = 88 * t5
15. a[t6] = 1.0
16. i = i +1
17. if i <= 10 goto (13)

For each leader, its basic block consists of itself and all instructions up to but not including the next leader

Example (contd.)



PES UNIVERSITY ONLINE

Example

```
int add(n, k){
s = 0;
a = 4;
i = 0;
if(k == 0)
    b = 1;
else
    b = 2;
while(i < n) {
    s = s + a * b;
    i = i + 1;
}
return s;
}
```

Example (contd.)

ICG

```
s = 0
a = 4
i = 0
ifFalse k == 0 goto L1
b = 1
goto L2
L1 : b = 2;
```

```
L2 : ifFalse i < n goto L3
t1 = a * b
t2 = s + t1
s = t2
t3 = i + 1
i = t3
goto L2
L3 : return s;
```

Example (contd.)

CFG

<u>s = 0</u>

a = 4

i = 0

ifFalse k == 0 goto L1

<u>b = 1</u>

goto L2

<u>L1 : b = 2;</u>

<u>L2 : ifFalse i < n goto L3</u>

<u>t1 = a * b</u>

t2 = s + t1

s = t2

t3 = i + 1

i = t3

goto L2

<u>L3 : return s;</u>

Example

```
i = m – 1;
j = n;
v = a[n];
while(1)
{
do
i = i + 1;
while (a[i] <v);
```

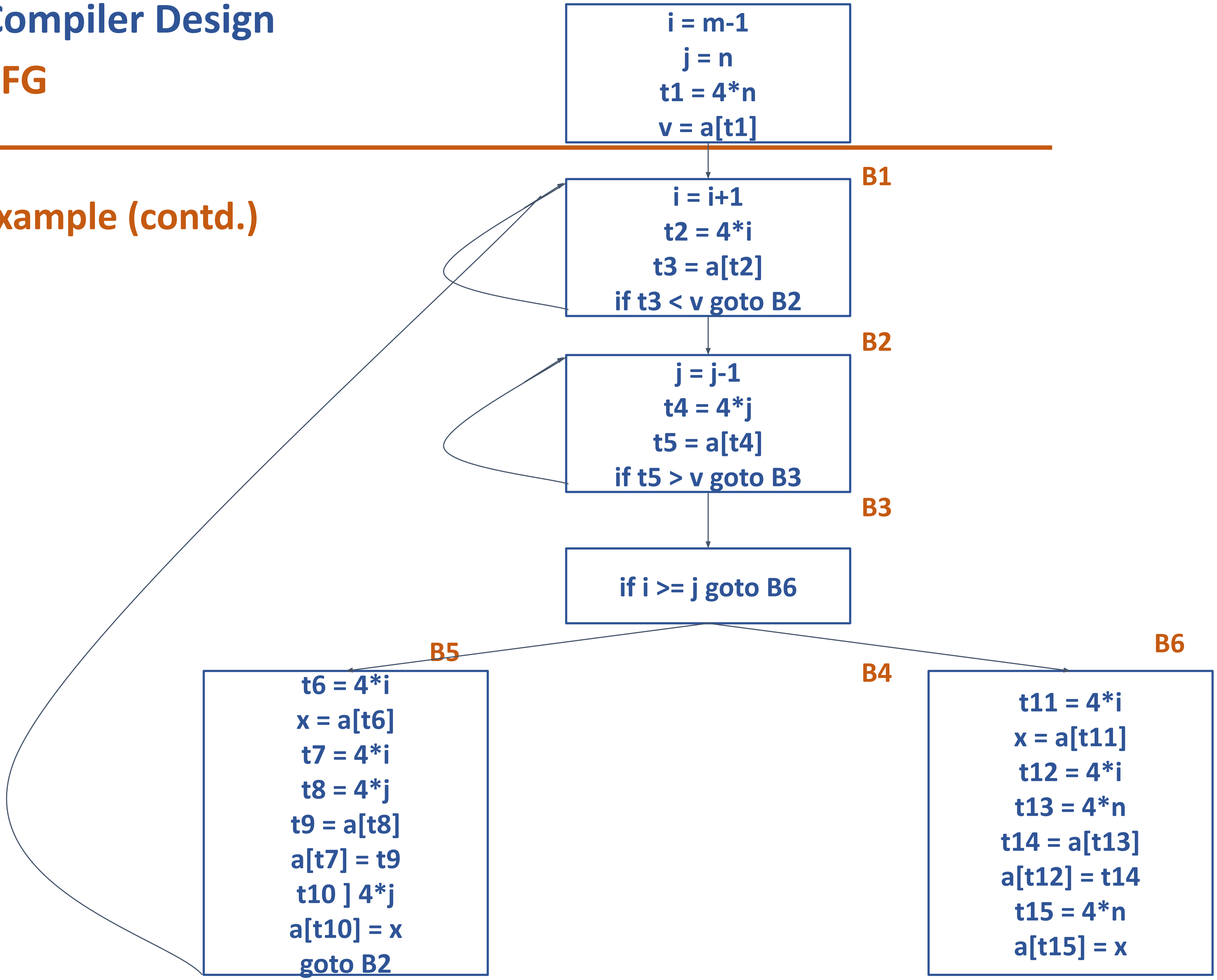
```
do
j = j – 1;
while(a[j] > v);
if(i >= j) break;
x = a[i];
a[i] = a[j];
a[j] = x
}
```

```
x = a[i];
a[i] = a[n];
a[n] = x;
```

Compiler Design

CFG

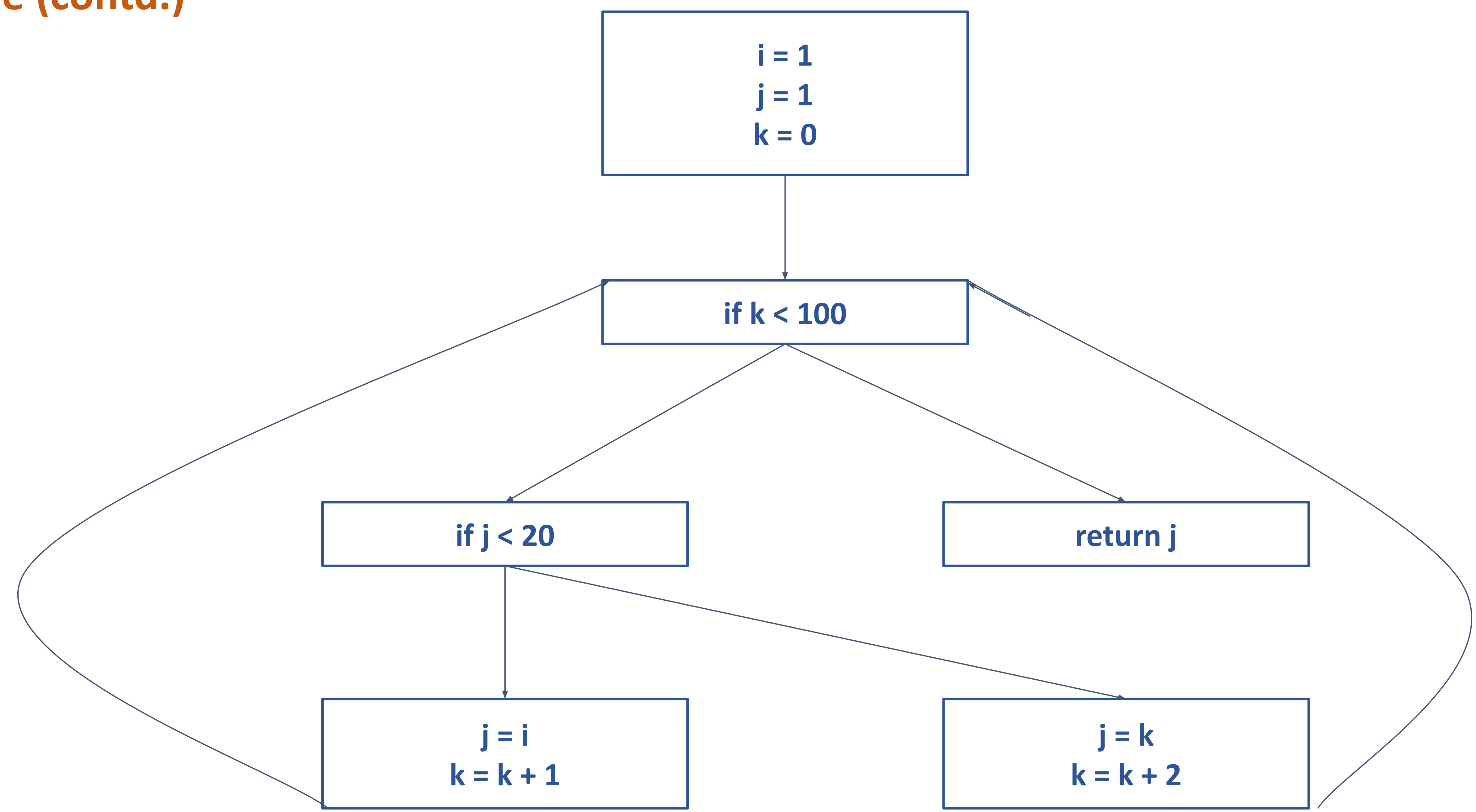
Example (contd.)



Example

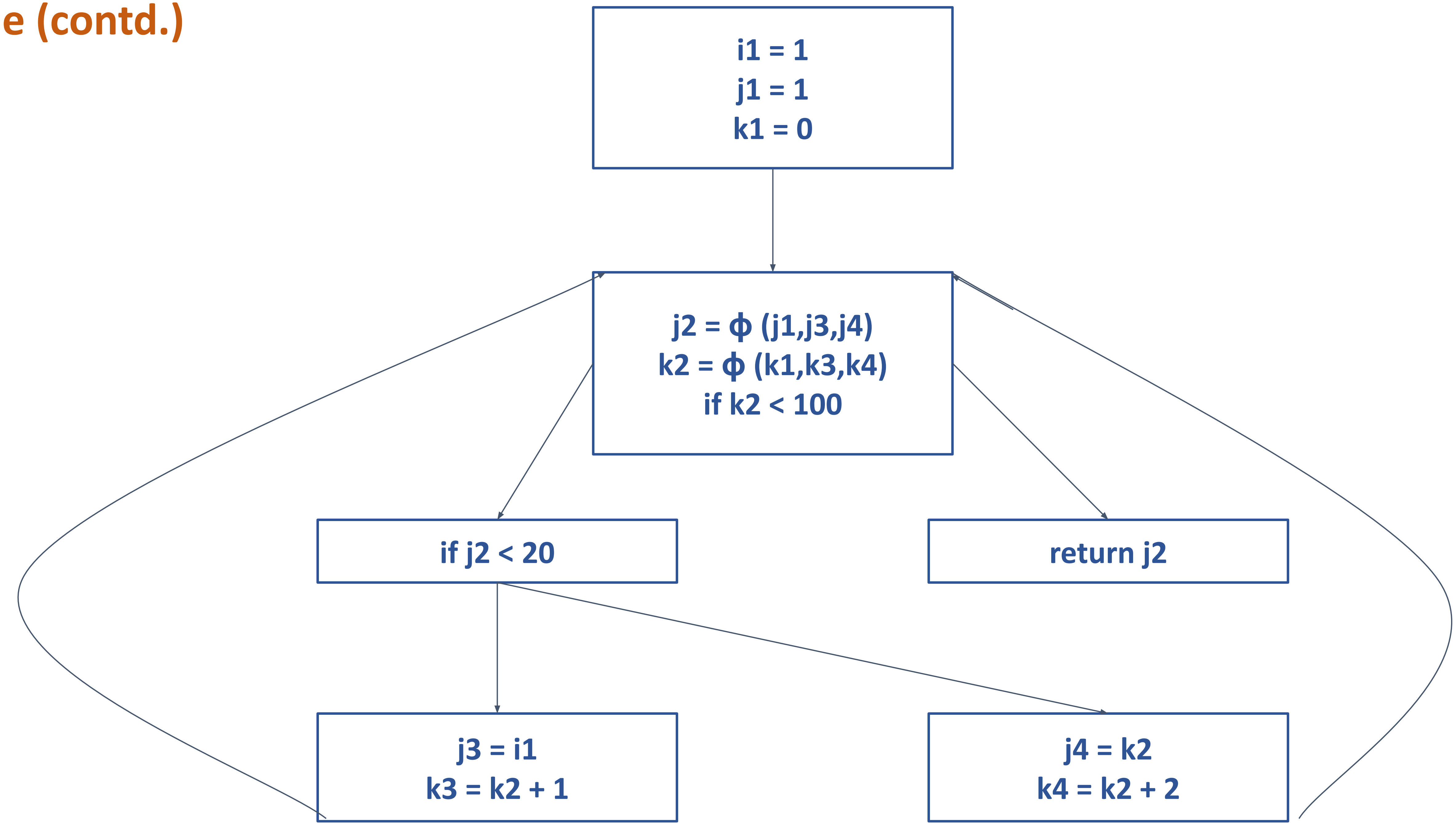
```
    i = 1
    j = 1
    k = 0
    while k < 100
        if j < 20
            j = i
            k = k + 1
        else
            j = k
            k = k + 1
        end
    end
return j
```

Example (contd.)



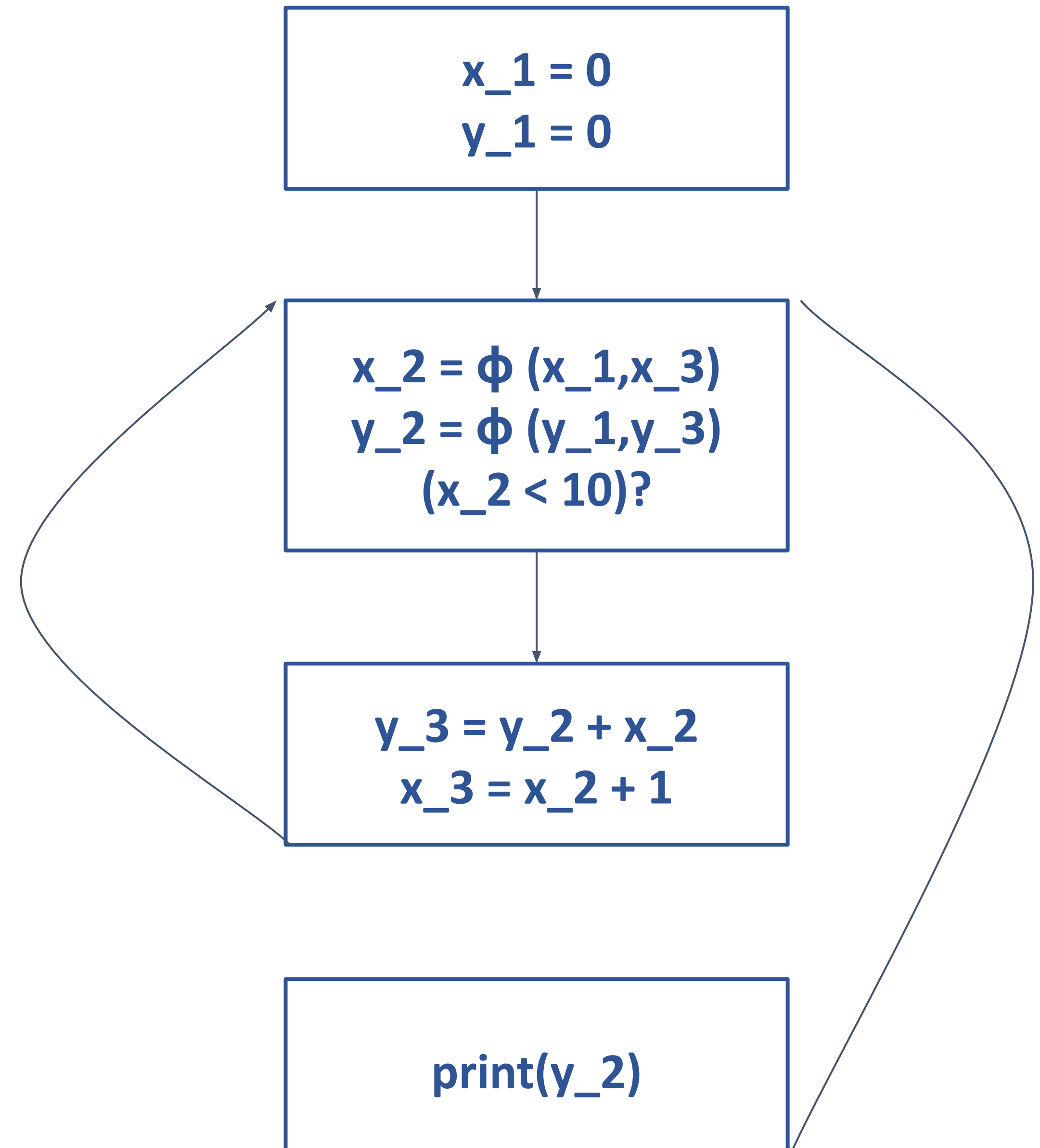
Example (contd.)

SSA



Example

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



Example

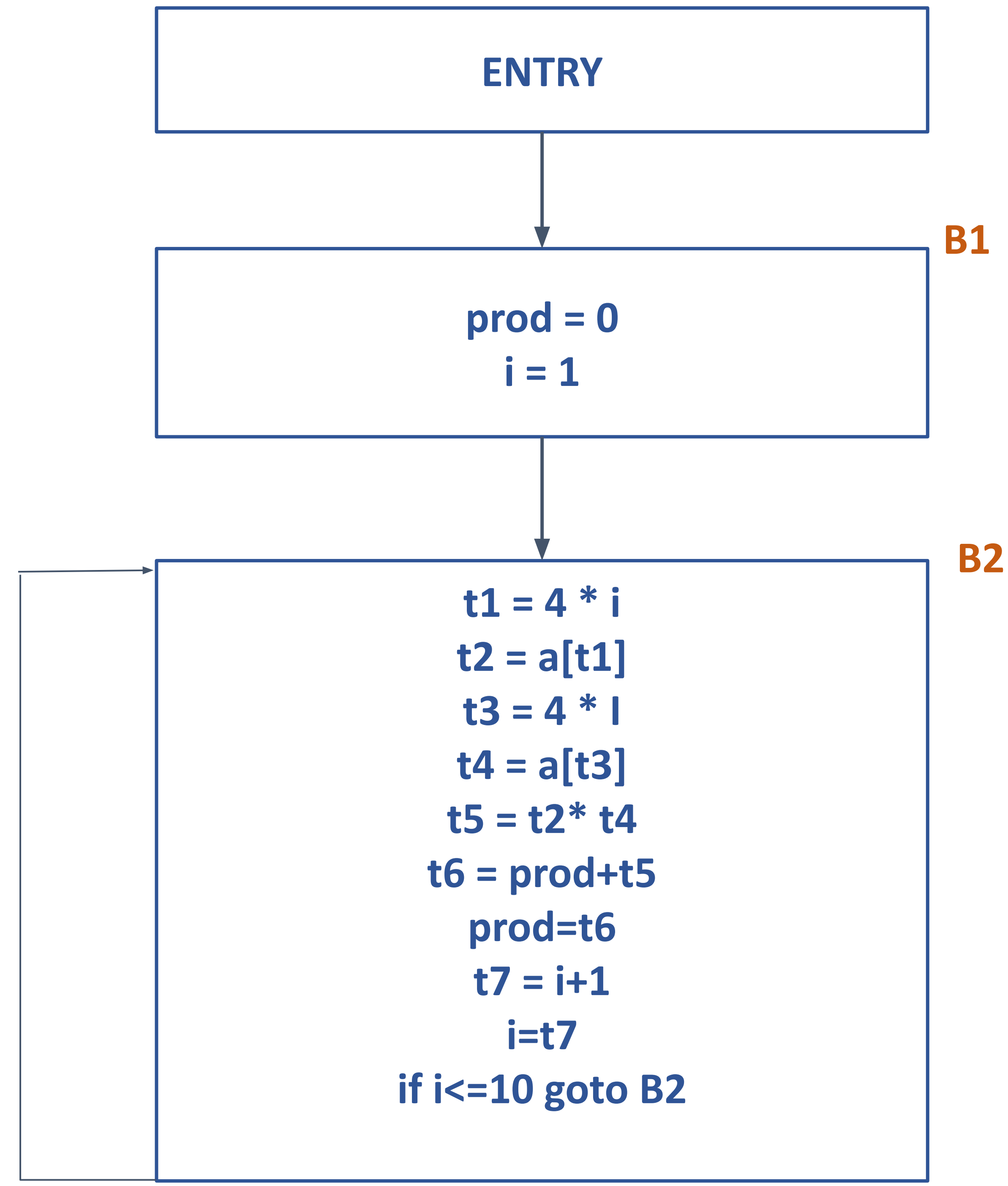
```
prod = 0
i =1
do
{

prod = prod + a[i] * b[i];
i = i +1

}
while(i <=10)
```

Example (contd.)

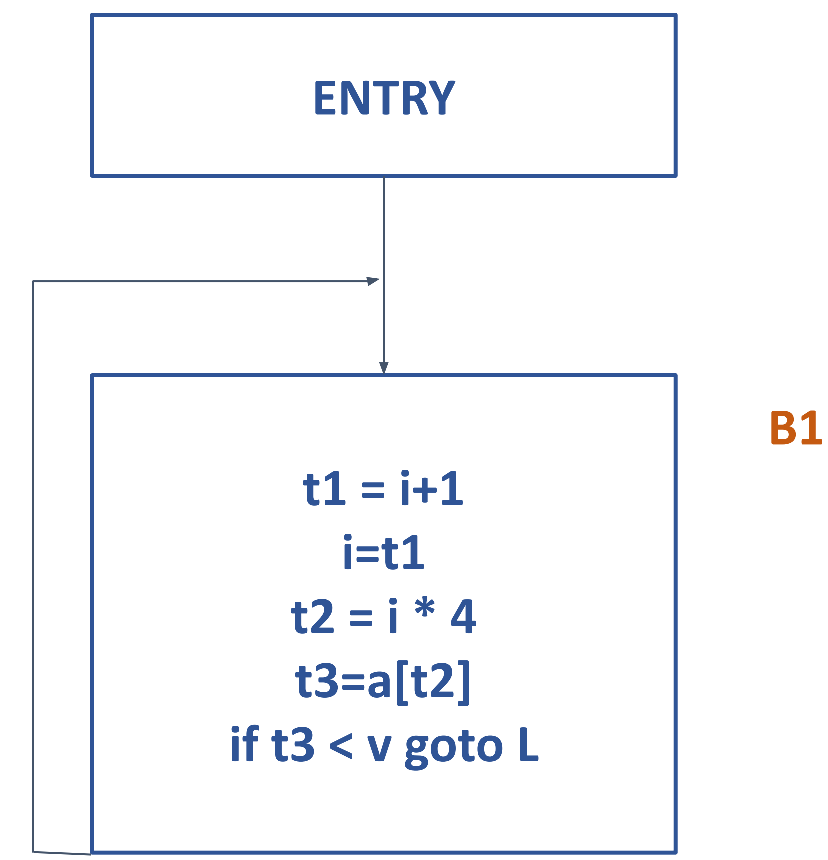
prod = 0
i = 1
L: t1 = 4 * i
t2 = a[t1]
t3 = 4 * i
t4 = a[t3]
t5 = t2* t4
t6 = prod+t5
prod=t6
t7 = i+1
i=t7
if i<=10 goto L



Example

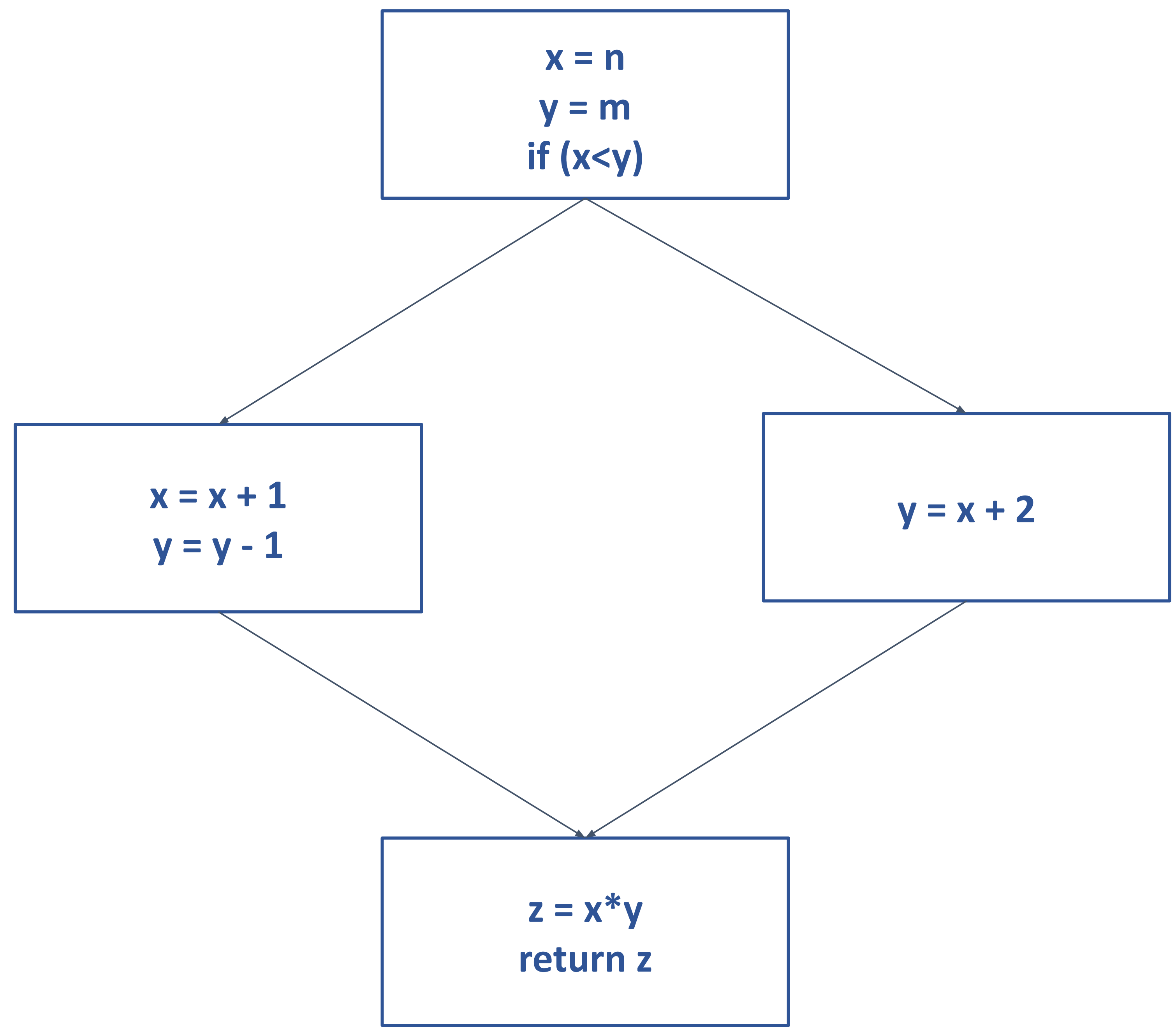
```
do
{
      i = i +1;
}
while (a[i] <v)
```

```
L : t1 = i+1
i=t1
t2 = i * 4
t3=a[t2]
if t3 < v goto L
```

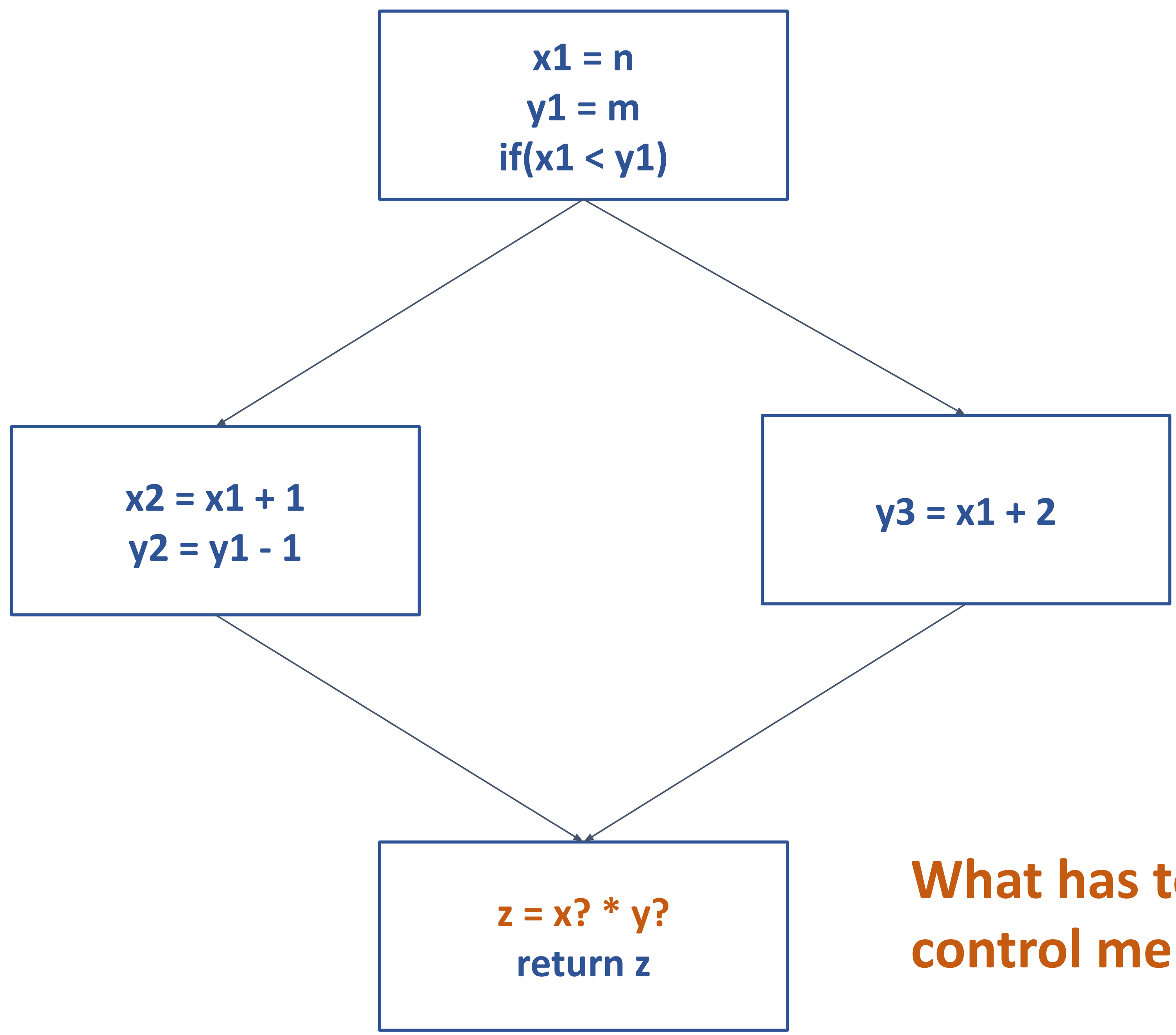


Example - *Program to SSA*

```
x = n
y = m
if(x < y)
{
    x = x + 1;
    y = y – 1;
}
else
{
    y = x + 2;
}
z = x * y;
return z;
```



Example(contd.)



What has to be done when control merges?

Example(contd.)

```
x1 = n
y1 = m
if(x1 < y1)
```

```
x2 = x1 + 1
y2 = y1 - 1
```

```
y3 = x1 + 2
```

```
x3 = φ(x1, x2)
y4 = φ(y2, y3)
z1 = x3 * y4
return z
```

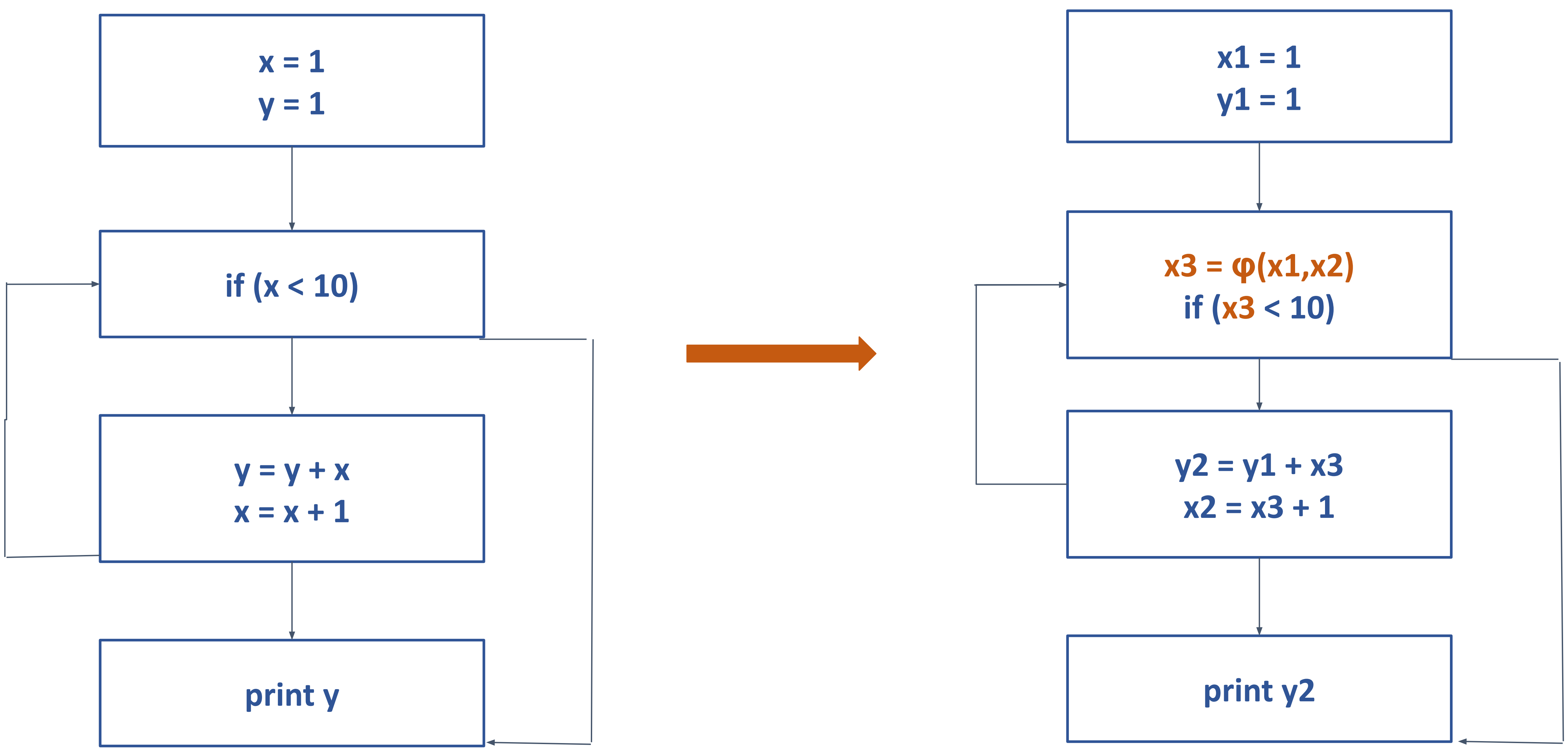
set x3 to either x1 or x2 based on which control flow edge is used to get here

set y4 to either y2 or y3 based on which control flow edge is used to get here

Compiler Design

CFG

Example - *CFG to SSA*



Two Types of Questions

1) Convert a given program to Three address code and then construct the CFG.

2) Convert a given program to SSA
   - The question could specify the student to convert the program to CFG (change the working of every loop - while or for in terms of if loop) and then SSAify it.

THANK YOU

Preet Kanwal
Department of Computer Science & Engineering
preetkanwal@pes.edu

Compiler Design

Preet Kanwal
Department of Computer Science & Engineering

Teaching Assistant : Kavya P K

Compiler Design

Unit 4: Next-Use Algorithm

Preet Kanwal

Department of Computer Science & Engineering

Lecture Overview

In this lecture, you will learn about -

- Basic Block Generation - Current Scenario
- Next-Use Information
- Next-Use Algorithm
- Examples

Basic Block Generation - Current Scenario

- Currently, in basic block generation -
  - We don’t know through which path the flow-graph has taken us to reach the current basic block.
  - We can’t assume that any variables are in registers.
  - We don’t know where we will go from this block.
  - Thus, values kept in registers must be stored back into their memory locations before the block is exited.

Requirements

- We want to keep variables in registers for as long as possible, to avoid having to reload them whenever they are needed.
- When a variable isn't needed any more we need to free the register to reuse it for other variables.
- Thus, we must know if a particular value will be used later in the basic block, i.e, its next-use.

Next-Use Information

- Goal: To know when the value of a variable will be used next.
- Next-use information must be computed within a basic block.
- In the given example -
  - Statement S5 uses the value of X computed (defined) at L1.
  - That is, X is live at S1.
  - Also at S1, we say X's next use is at S5.
  - Thus, it is a good idea to keep X in a register between S1 and S5.
- Advantage - knowing that a variable (assigned a register) is not used any further helps reassign the register to some other variable.

S1: X = ...
...
S5: Y = X

Assumptions

The following assumptions are made -

- All variables are live on exit. Thus, they will be stored back into their respective memory locations.
- Temporaries are dead on exit.

Next-use Algorithm

- It is a two-pass algorithm that computes next-use and liveness information of a basic block.
- Pass 1
  - In the first pass we scan forward over the basic block to find the end.
  - For each variable X used in the block we create fields X.live and X.next_use in the symbol table.
  - Set X.live:=FALSE; X.next use:=NONE;

Statement	X.live	Y.live	Z.live	X.next_use	Y.next_use	Z.next_use
i	*False*	*False*	*False*	-	-	-

- Pass 2
  - Scan backwards over the basic block.
  - For every tuple (i) x = y op z , do the following -
    - Copy the live/next use-info from x, y, z's symbol table entries into the tuple data for tuple (i).
    - Update x, y, z's symbol table entries as follows -

Statement	X.live	Y.live	Z.live	X.next_use	Y.next_use	Z.next_use
i	*False*	*True*	*True*	-	*i*	*i*

Example 1

- Go through the next-use algorithm for the following code

  x := y + z
  z := x * 5
  y := z - 7
  x := z + y

Statement	Symbol Table Info						Instruction Info					
	live			next-use			live			next-use		
	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
i												

Example 1 - Solution

First pass - Set live variables in each statement to False, next_use to None.

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*F*	*F*				*F*	*F*	*F*			
(2)	z := x * 5	*F*	*F*	*F*				*F*	*F*	*F*			
(3)	y := z - 7	*F*	*F*	*F*				*F*	*F*	*F*			
(4)	x := z + y	*F*	*F*	*F*				*F*	*F*	*F*			

Example 1 - Solution

PES UNIVERSITY ONLINE

Second pass - Scan backwards over the basic block - start with (4)

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*F*	*F*				*F*	*F*	*F*			
(2)	z := x * 5	*F*	*F*	*F*				*F*	*F*	*F*			
(3)	y := z - 7	*F*	*F*	*F*				*F*	*F*	*F*			
(4)	x := z + y	*F*	*T*	*T*	-	*4*	*4*	*F*	*F*	*F*			

Example 1 - Solution

Copy the live/next use-info from x, y, z's symbol table entries into the tuple data.

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*F*	*F*				*F*	*F*	*F*			
(2)	z := x * 5	*F*	*F*	*F*				*F*	*F*	*F*			
(3)	y := z - 7	*F*	*F*	*F*				*F*	*T*	*T*	-	*4*	*4*
(4)	x := z + y	*F*	*T*	*T*	-	*4*	*4*	*F*	*F*	*F*			

Example 1 - Solution

Statement (3)

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*F*	*F*				*F*	*F*	*F*			
(2)	z := x * 5	*F*	*F*	*F*				*F*	*F*	*F*			
(3)	y := z - 7	*F*	*F*	*T*	-	-	*3*	*F*	*T*	*T*	-	*4*	*4*
(4)	x := z + y	*F*	*T*	*T*	-	*4*	*4*	*F*	*F*	*F*			

Example 1 - Solution

Statement (3)

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*F*	*F*				*F*	*F*	*F*			
(2)	z := x * 5	*F*	*F*	*F*				*F*	*F*	*T*	-	-	*3*
(3)	y := z - 7	*F*	*F*	*T*	-	-	*3*	*F*	*T*	*T*	-	*4*	*4*
(4)	x := z + y	*F*	*T*	*T*	-	*4*	*4*	*F*	*F*	*F*			

Example 1 - Solution

Statement (2)

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*F*	*F*				*F*	*F*	*F*			
(2)	z := x * 5	*T*	*F*	*F*	*2*	-	-	*F*	*F*	*T*	-	-	*3*
(3)	y := z - 7	*F*	*F*	*T*	-	-	*3*	*F*	*T*	*T*	-	*4*	*4*
(4)	x := z + y	*F*	*T*	*T*	-	*4*	*4*	*F*	*F*	*F*			

Example 1 - Solution

Statement (2)

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*F*	*F*				*T*	*F*	*F*	*2*	-	-
(2)	z := x * 5	*T*	*F*	*F*	*2*	-	-	*F*	*F*	*T*	-	-	*3*
(3)	y := z - 7	*F*	*F*	*T*	-	-	*3*	*F*	*T*	*T*	-	*4*	*4*
(4)	x := z + y	*F*	*T*	*T*	-	*4*	*4*	*F*	*F*	*F*			

Example 1 - Solution

Statement (1)

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*T*	*T*	-	*1*	*1*	*T*	*F*	*F*	*2*	-	-
(2)	z := x * 5	*T*	*F*	*F*	*2*	-	-	*F*	*F*	*T*	-	-	*3*
(3)	y := z - 7	*F*	*F*	*T*	-	-	*3*	*F*	*T*	*T*	-	*4*	*4*
(4)	x := z + y	*F*	*T*	*T*	-	*4*	*4*	*F*	*F*	*F*			

Example 1 - Solution

Statement		Symbol Table Info						Instruction Info					
		live			next-use			live			next-use		
		X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
(1)	x := y + z	*F*	*T*	*T*	-	*1*	*1*	*T*	*F*	*F*	*2*	-	-
(2)	z := x * 5	*T*	*F*	*F*	*2*	-	-	*F*	*F*	*T*	-	-	*3*
(3)	y := z - 7	*F*	*F*	*T*	-	-	*3*	*F*	*T*	*T*	-	*4*	*4*
(4)	x := z + y	*F*	*T*	*T*	-	*4*	*4*	*F*	*F*	*F*	-	-	-

The data in each row reflects the state in the symbol table and in the data section of each instruction i after i has been processed.

Example 2

- **Go through the next-use algorithm for the following code -**

(1) a = a – b
(2) u = a – c
(3) v = t + u
(4) d = v + u

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V

Example 2 - Solution

First pass - Set live variables in each statement to False, next_use to None.

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
2	u = a - c	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
3	v = t+ u	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
4	d = v +u	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							

Compiler Design

Example 2 - Solution

Second pass - Scan backwards over the basic block - start with (4)

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
2	u = a - c	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
3	v = t+ u	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
4	d = v +u	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	4	4	*F*	*F*	*F*	*F*	*F*	*F*	*F*							

Compiler Design

Example 2 - Solution

Copy the live/next use-info from symbol table entries into the tuple data.

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
2	u = a - c	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
3	v = t+ u	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*
4	d = v +u	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*	*F*	*F*	*F*	*F*	*F*	*F*	*F*							

Example 2 - Solution

Statement (3)

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	F	F	F	F	F	F	F								F	F	F	F	F	F	F							
2	u = a - c	F	F	F	F	F	F	F								F	F	F	F	F	F	F							
3	v = t+ u	F	F	F	F	T	T	F	-	-	-	-	3	3	-	F	F	F	F	F	T	T	-	-	-	-	-	4	4
4	d = v +u	F	F	F	F	F	T	T	-	-	-	-	-	4	4	F	F	F	F	F	F	F							

Example 2 - Solution

Statement (3)

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
2	u = a - c	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*T*	*T*	*F*	-	-	-	-	*3*	*3*	-
3	v = t+ u	*F*	*F*	*F*	*F*	*T*	*T*	*F*	-	-	-	-	*3*	*3*	-	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*
4	d = v +u	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*	*F*	*F*	*F*	*F*	*F*	*F*	*F*							

Compiler Design

Example 2 - Solution

Statement (2)

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	*F*	*F*	*F*	*F*	*F*	*F*	*F*								*F*	*F*	*F*	*F*	*F*	*F*	*F*							
2	u = a - c	*T*	*F*	*T*	*F*	*F*	*F*	*F*	*2*	-	*2*	-	-	-	-	*F*	*F*	*F*	*F*	*T*	*T*	*F*	-	-	-	-	*3*	*3*	-
3	v = t+ u	*F*	*F*	*F*	*F*	*T*	*T*	*F*	-	-	-	-	*3*	*3*	-	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*
4	d = v +u	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*	*F*	*F*	*F*	*F*	*F*	*F*	*F*							

Example 2 - Solution

Statement (2)

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	F	F	F	F	F	F	F								T	F	T	F	F	F	F	2	-	2	-	-	-	-
2	u = a - c	T	F	T	F	F	F	F	2	-	2	-	-	-	-	F	F	F	F	T	T	F	-	-	-	-	3	3	-
3	v = t+ u	F	F	F	F	T	T	F	-	-	-	-	3	3	-	F	F	F	F	F	T	T	-	-	-	-	-	4	4
4	d = v +u	F	F	F	F	F	T	T	-	-	-	-	-	4	4	F	F	F	F	F	F	F							

Example 2 - Solution

Statement (1)

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	*T*	*T*	*F*	*F*	*F*	*F*	*F*	*1*	*1*	-	-	-	-	-	*T*	*F*	*T*	*F*	*F*	*F*	*F*	*2*	-	*2*	-	-	-	-
2	u = a - c	*T*	*F*	*T*	*F*	*F*	*F*	*F*	*2*	-	*2*	-	-	-	-	*F*	*F*	*F*	*F*	*T*	*T*	*F*	-	-	-	-	*3*	*3*	-
3	v = t+ u	*F*	*F*	*F*	*F*	*T*	*T*	*F*	-	-	-	-	*3*	*3*	-	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*
4	d = v +u	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*	*F*	*F*	*F*	*F*	*F*	*F*	*F*							

Example 2 - Solution

Finally -

Statement		Symbol Table Info														Instruction Info													
		live							next-use							live							next-use						
		A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V	A	B	C	D	T	U	V
1	a = a - b	*T*	*T*	*F*	*F*	*F*	*F*	*F*	*1*	*1*	-	-	-	-	-	*T*	*F*	*T*	*F*	*F*	*F*	*F*	*2*	-	*2*	-	-	-	-
2	u = a - c	*T*	*F*	*T*	*F*	*F*	*F*	*F*	*2*	-	*2*	-	-	-	-	*F*	*F*	*F*	*F*	*T*	*T*	*F*	-	-	-	-	*3*	*3*	-
3	v = t+ u	*F*	*F*	*F*	*F*	*T*	*T*	*F*	-	-	-	-	*3*	*3*	-	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*
4	d = v +u	*F*	*F*	*F*	*F*	*F*	*T*	*T*	-	-	-	-	-	*4*	*4*	*F*	*F*	*F*	*F*	*F*	*F*	*F*	-	-	-	-	-	-	-

Example 3 - Homework

- Go through the next-use algorithm for the following code -

(1) u:=a - b
(2) v:=c - a
(3) a:=u + v

Statement		Symbol Table Info										Instruction Info									
		live					next-use					live					next-use				
		A	B	C	U	V	A	B	C	U	V	A	B	C	U	V	A	B	C	U	V
1	u = a - b																				
2	v = c - a																				
3	a = u + v																				

THANK YOU

Preet Kanwal
Department of Computer Science & Engineering
preetkanwal@pes.edu