PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Sree Pranavi G

Compiler Design

Unit 4: Code Optimization

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will learn about -

- **Optimization techniques/methods**
- **Types of optimization**

Code Optimization

- Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

- The replacement of one sequence of instruction by a faster sequence of instruction that does the same thing is called “code optimization” or “code improvement”.

Compiler Design

Criteria for Optimization

Good code optimization,

- **Must be Semantics Preserving.**
- **Must Speed up programs on average.**
- **Should be worth the effort - must not delay compilation process.**

Techniques/Methods of Optimization

- **Control Flow Analysis** - Identifies loops in the flow graph of a program since loops are usually good candidates for improvement.

  A control flow graph shows how instructions in the program are sequenced.

- **Data Flow Analysis** - Collects information about the way variables are used in a program.

  Determines how data flows through the program by examining the use of variables.

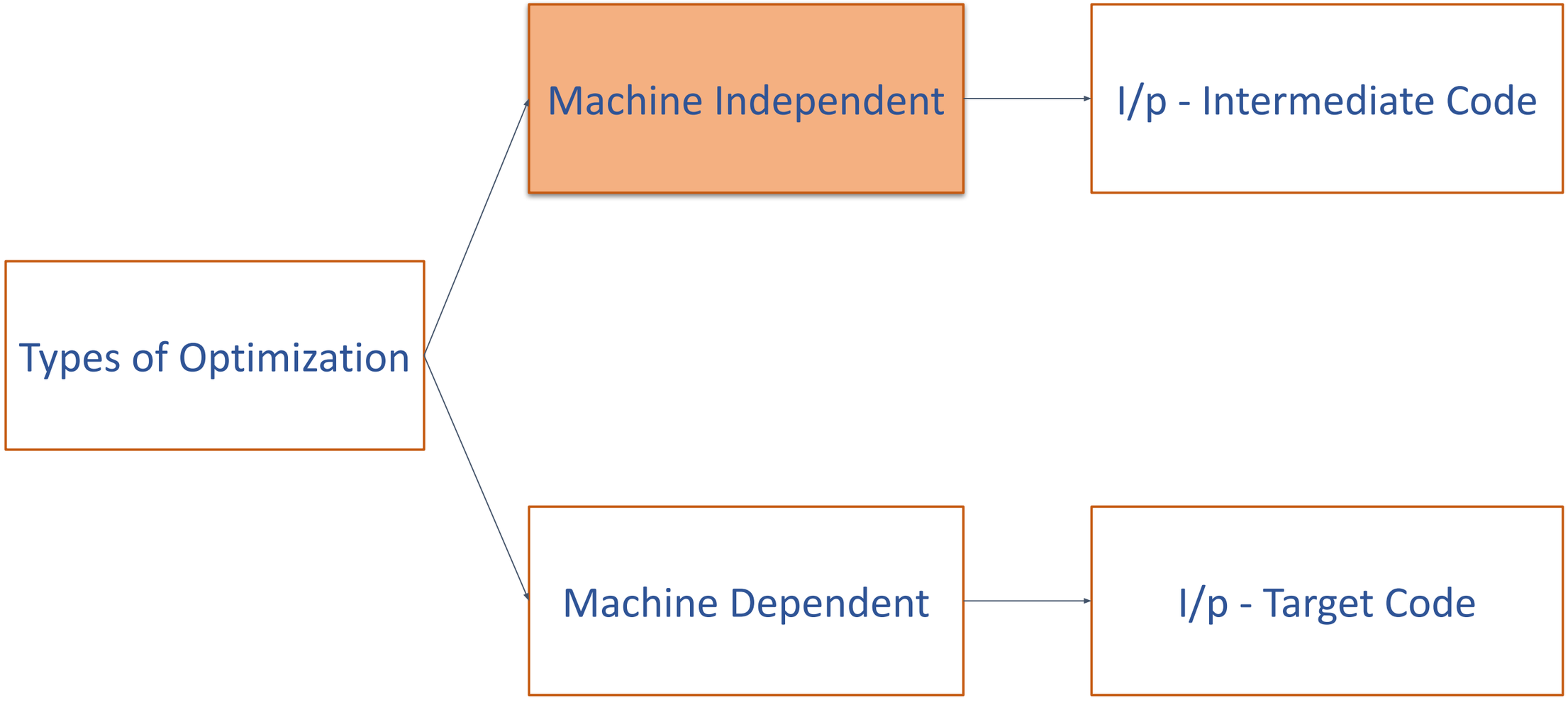
Techniques/Methods of Optimization

- **Note - Dramatic improvements are usually obtained by improving the source code.**

The programmer is always responsible in finding the best possible data structures and algorithms for solving a problem.

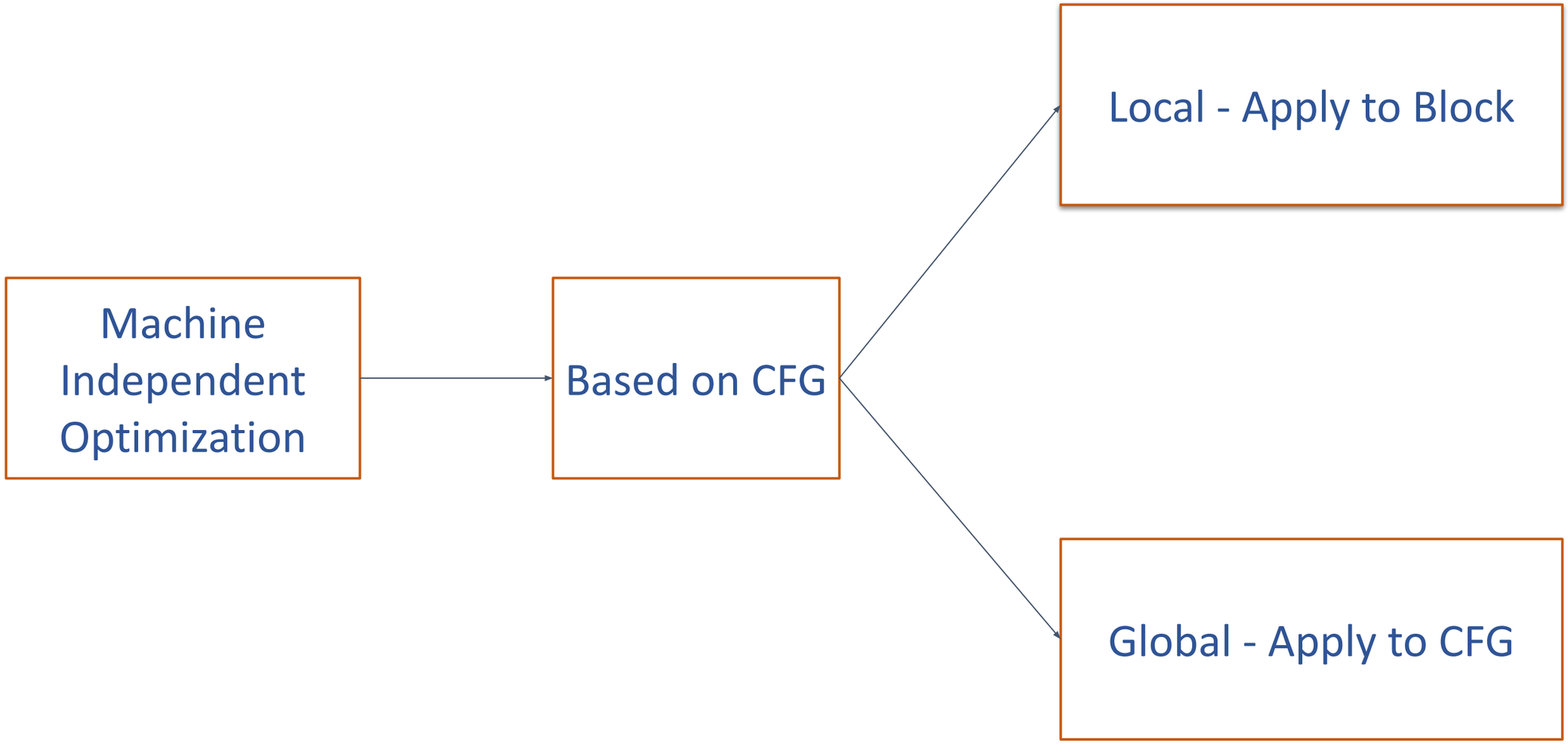
Compiler Design

Types of Optimization



Compiler Design

Types of Optimization



Compiler Design

Different Machine Independent Optimizations

- Constant Folding
- Constant Propagation
- Common Subexpression Elimination(CSE)
- Copy propagation
- Dead Code Elimination(DCE)
- Strength Reduction
- Packing temporaries
- Loop optimizations

Compiler Design

Constant Folding

- Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.

- Example : i = 30 * 20 * 10

- Example : x = 5 + 4 / 2

- Most modern compilers would not actually generate two multiply instructions and a store for this statement.

- Instead, they identify such constructs and substitute the computed values at compile time

Compiler Design

Constant Folding

- **Algebraic identities** can also be constant folded by the compiler.

- An Algebraic/ arithmetic identity is an equation that is always true regardless of the values assigned to the variables.

  Example : 0 * x = 0 ; 1 * x = x;

  even if the compiler does not know the value of x

- **Concatenation of string literals** and constant strings can be constant folded.

  Example : Code such as “dog" + “house" may be replaced with “doghouse".

Compiler Design

Constant Propagation

- **Constant propagation is the process of substituting the values of known constants in expressions at compile time.**

- **An Constant assigned to a variable is substituted when the variable is encountered during compile time. This is usually called Constant propagation.**

Example :

int a = 10; int b = 45 - a / 2; return b * (200 / a + 2);

Propagating a yields,

int a = 10; int b = 45 - 10 / 2; return b * (200 / 10 + 2);

Compiler Design

Constant Propagation

Example :

int a = 10; int b = 45 - a / 2; return b * (200 / a + 2);

Propagating a yields,

int a = 10; int b = 45 - 10 / 2; return b * (200 / 10 + 2);

Fold the value of b,

int a = 10; int b = 40; return b * (200 / 10 + 2);

Propagating b yields,

int a = 10; int b = 40; return 40 * (200 / 10 + 2);

constant fold the result,

int a = 10; int b = 40; return 880;

a, b becomes dead code, hence finally we get

return 880;

use gcc -O1 -fdump-tree-optimized prog.c to view the output of optimized file.

Compiler Design

Constant Folding & Propagation

Example :

x = 10

a = (x * x / 2) + 1

x =10

a = (10 * 10 / 2) + 1

a = (20 / 2) + 1

a = 11

//Here value of x is propagated and then constant folded.

Compiler Design

Constant Folding & Propagation

Example :

```
int a = 30;
int b = 9 - (a / 5);
int c;
c = b * 4;
if (c > 10) {
c = c - 10;
}
return c * (60 / a);
```

```
int a = 30;
int b = 3;
int c;
c = 12;

if (true) {
c = 2;
}
return c * 2;
```

CSE

- Common subexpression elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and replacing them with a single variable holding the computed value.

- An occurrence of E is called a common subexpression, if E is previously computed and the values in E have not changed since the previous computation.

- Relies on data flow analysis.

Example :

```
a = b * c + g;
d = b * c * e;
```

```
t1 = b * c;
a = t1 + g;
d = t1 * e;
```

Example :

t6 = 4 * i → Retain

x = a[t6]

t7 = 4 * i → Eliminate

t8 = 4 * j → Retain

t9 = a[t8]

a[t7] = t9

t10 = 4 * j → Eliminate

a[t10] = x

gotoB2

t6 = 4 * i

x = a[t6]

t8 = 4 * j

t9 =a[t8]

a[t6] =t9

a[t8] = x

goto B2

Compiler Design

Copy Propagation

- Copy propagation is the process of replacing the occurrences of targets of direct assignments with their values.

- Copy propagation transformation is to use v for u, whenever possible after copy statement u = v.

- Useful "clean up" optimization frequently used after other optimizations have already been run.

Compiler Design

Copy Propagation

Example:

– y = x

z = 3 + y

Copy propagation would yield,

z = 3 + x

Compiler Design

Copy Propagation

Example :

```
p = a + c
q =b
r = q * q
```

```
p = a + c
q =b
r = b * b
```

```
p = a + c
r = b * b
```

Compiler Design

Dead Code Elimination

- Dead code is a code that is never executed or that does nothing useful.

- Such code that is unreachable or that does not affect the program can be eliminated.

Example :

```
int fun()
{ int a = 20;  //deadcode
int b = 40;
int c = b * b;
return c;
a = a +b;      //unreachable }
```

```
int fun()
{ int b = 40;
int c = b * b;
return c; }
```

Dead Code Elimination

Example :

```
int global;
void f () {
int i;
i = 1;
global = 1;
global = 2;
return;
global = 3;
}
```

```
int global;
void f ()
{ global = 2;
return;
}
```

Compiler Design

Different Machine Independent Optimizations

- *Constant Folding*
- *Constant Propagation*
- *Common Subexpression Elimination(CSE)*
- *Copy propagation*
- *Dead Code Elimination(DCE)*
- **Strength Reduction**
- **Packing temporaries**
- **Loop optimizations**

function-preserving / semantics-preserving transformations

Compiler Design

Strength Reduction

- The process of replacing an expensive operation by a cheaper one is known as Strength Reduction.

Expensive	Cheaper
x ^ 2	x * x
x * 2	x + x
x * 2	x << 1
x / 2	x * 0.5
x / 2	x >> 1

Compiler Design

Packing Temporaries

- Replace distinct temporaries by a single one if, they are not simultaneously live.

Example :

t1 = a + a

t2 = t1 + b

c = t2 * t2

t1 = a + a

t2 = t1 + b

c = t1 * t1

not correct semantically

Compiler Design

Loop Optimizations

- **Loop optimization is the process of increasing execution speed and reducing the overheads associated of loops.**

- **It plays an important role in improving cache performance and making effective use of parallel processing capabilities.**

- **Most execution time of a scientific program is spent on loops; many compiler optimization techniques have been developed to make them faster.**

Loop Optimizations

Types

- Loop Invariant Detection
- Loop Unrolling or loop unwinding
- Induction Variable Detection
- Code Motion

Compiler Design

Loop Invariant Detection

- A loop invariant is a variable whose value is constant for the duration of the loop.

Example :

```
a = 5
for(i = 0; i < 5;i++)
{
x[i] = a * i;
}
// ‘a’ is the loop invariant
```

```
a = 5
b =0
for(i= 0; i < 5;i++)
{
x[i] = b;
b = b + a; }
```

Code Motion

- A modification that decreases the amount of code in a loop can be termed as Code motion.

Example :

Evaluation of limit-2 is a loop-invariant computation in the following while-statement,

```
while (i <= limit-2)
```

```
t = limit-2
while ( i <= t )
```

Now, the computation of limit - 2 is performed once, before we enter the loop. Previously, there would be n + 1 calculations of limit - 2 if we iterated the body of the loop n times.

Compiler Design

Loop Unrolling or Loop Unwinding

- Loop overhead can be reduced by reducing the number of iterations and replicating the body of the loop.

- Unrolling duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which may degrade performance by impairing the instruction pipeline.

- Completely unrolling a loop eliminates all overhead (except multiple instruction fetches & increased program load time), but requires that the number of iterations be known at compile time.

- The transformation can be undertaken manually by the programmer or by an optimizing compiler.

Compiler Design

Loop Unrolling or Loop Unwinding

Example :

```
for (i = 0; i < 100; i++)
    g ();
```

```
for (i = 0; i < 100; i+=2)
{
    g ();
    g ();
}
```

Number of iterations can be reduced from 100 to 50.

Compiler Design

Loop Unrolling or Loop Unwinding

Example :

```
for (i=0;i<10;i++)
{
    a[i]=0;
}
```

```
for (i=0;i<10;i++)
{
a[i] = 0; i++;
a[i] = 0;
}
```

Compiler Design

Induction Variable Detection

- An Induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop.

- A variable whose value varies regularly according to the loop control variable.

Example :

L : i = i + 1

t1 = 4 * i

t2 = a[t1]

if t2 < v goto L

Initialize : t1 = 4 * i

L : t1 = t1 + 4

t2 = a[t1]

if t2 < v goto L

Compiler Design

Induction Variable Detection

Example :

```
for (i=0;i<10;i+=2)
{
x=i*3;
a[i]=y-x;
}
```

```
x = -6;
for (i=0; i<10; i+=2)
{
x = x + 6;
a[i]=y-x;
}
```

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 : Sree Pranavi G

Compiler Design

Unit 4: Code Optimization - Local

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

In this lecture, you will learn about -

- **Block Optimization**
- **Local optimization using DAG**

Local Optimization

Example 1

Apply local optimization on the following block.

$a = x^2$

b =3

c =x

d = c* c

e= b * 2

f= a+ d

g = e* f

Local Optimization

Contd.

```
a = x^2
b =3
c =x
d = c* c
e= b * 2
f= a+ d
g = e* f
```

Strength Reduction

```
a = x * x
b =3
c =x
d = c* c
e= b << 1
f= a+ d
g = e* f
```

Local Optimization

Contd.

```
a = x * x
b =3
c =x
d = c* c
e= b << 1
f= a+ d
g = e* f
```

Const. Propagation

```
a = x * x
b =3
c =x
d = c* c
e= 3 << 1
f= a+ d
g = e* f
```

Local Optimization

Contd.

```
a = x * x
b =3
c =x
d = x * x
e =6
f = a +d
g = e * f
```

Const. Propagation

```
a = x * x
b =3
c =x
d = x * x
e =6
f = a +d
g=6* f
```

Local Optimization

Contd.

```
a = x * x
b =3
c =x
d = x * x
e =6
f = a +d
g=6* f
```

CSE

```
a = x * x
b =3
c =x


e =6
f = a +a
g=6 * f
```

Local Optimization

Contd.

```
a = x * x
b =3
c =x
e =6
f = a +a
g=6 * f
```

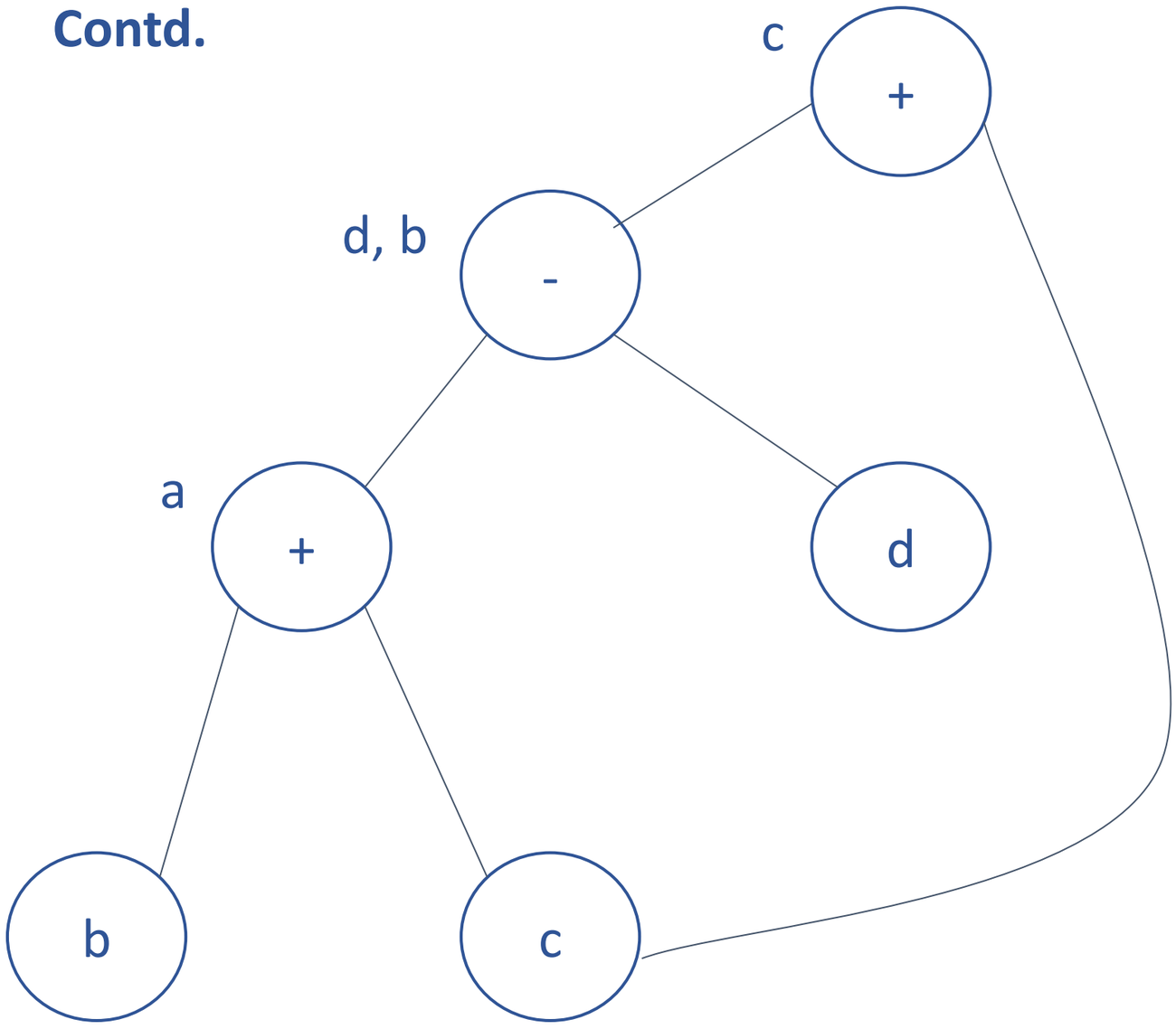
Dead Code Elim.

```
a = x * x
f = a + a
g = 6 * f
```

Compiler Design

Local Optimization using DAG

Contd.



a = b + c

b = a – d

c = b + c

d = b

Local Optimization using DAG

Example

Apply local optimization on the following block using DAG.

a = b + c

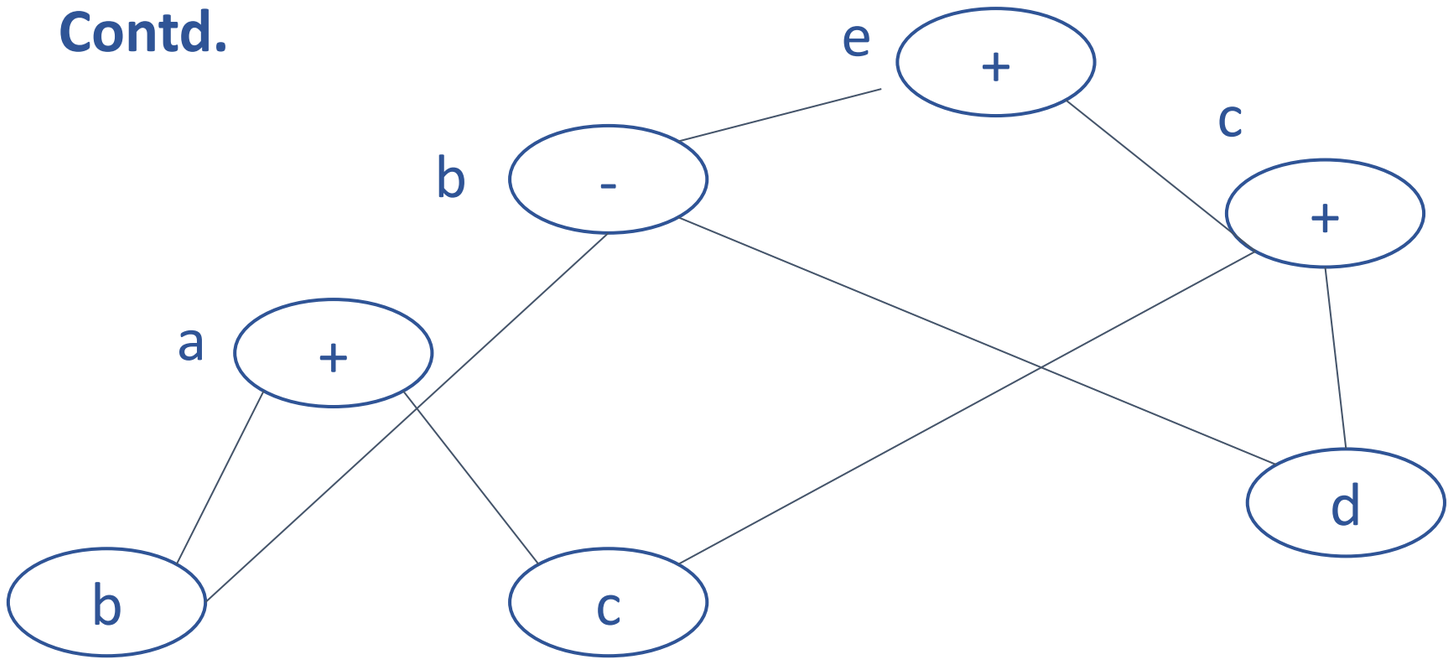
b = b – d

c = c + d

e = b + c

Local Optimization

Contd.



a = b + c

b = b – d

c = c + d

e = b + c

Note : DAG does not exhibit any CSE in this case.

However if we apply algebraic identities to DAG we can optimize the code as,

e = b – d + c + d

e = b + c

value computed by a and e is the same.

Local Optimization using DAG

Example

Apply local optimization on the following block using DAG.

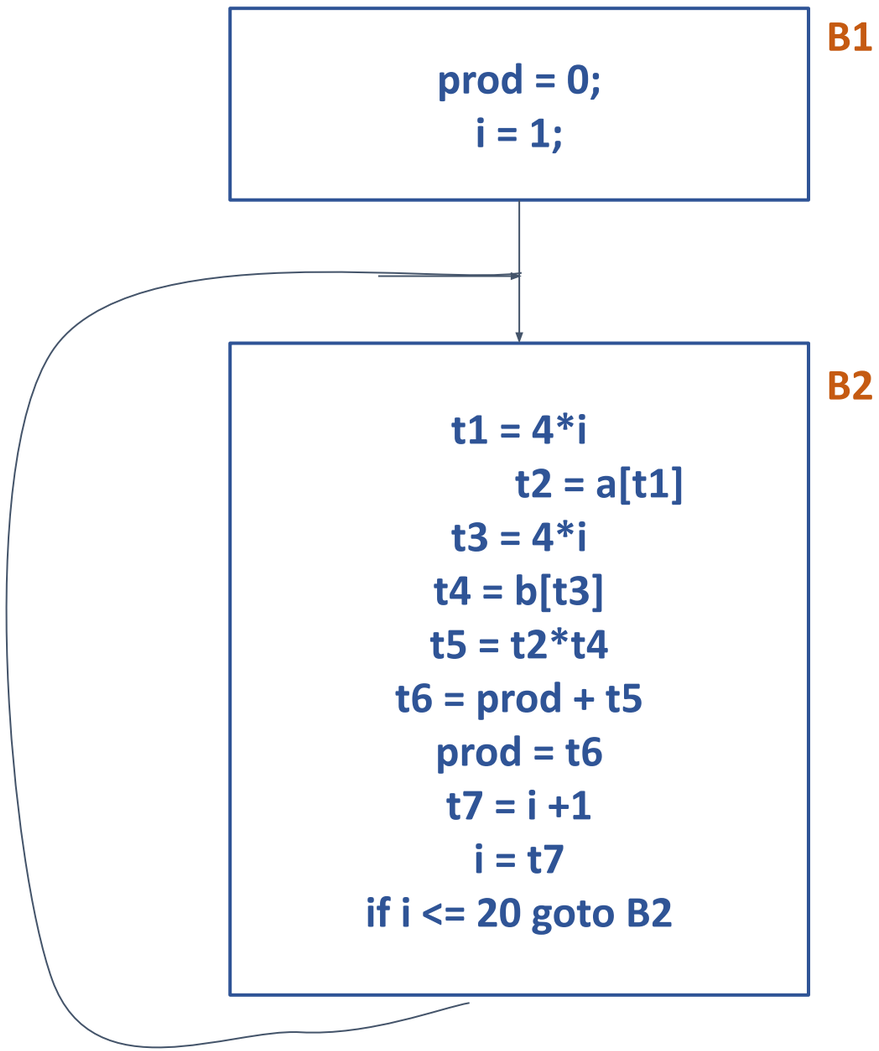
```
prod = 0;
i = 1;
do
prod = prod + a[i] * b[i];
i = i + 1;
while (i <=20);
```

TAC

```
prod = 0;
i = 1;
loop : t1 = 4*i
t2 = a[t1]
t3 = 4*i
t4 = b[t3]
t5 = t2*t4
t6 = prod + t5
prod = t6
t7 = i +1
i = t7
if i <= 20 goto loop
```

Local Optimization using DAG

Contd.

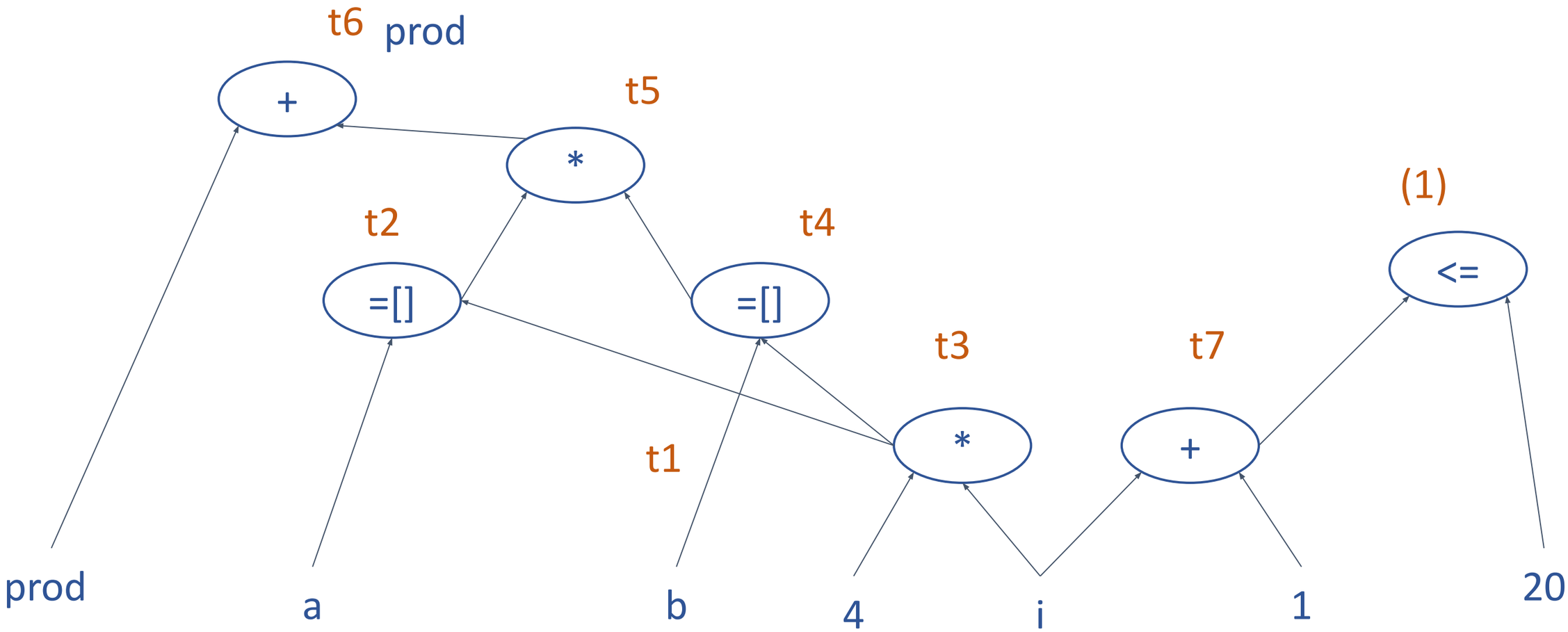


CFG

Compiler Design

Local Optimization using DAG

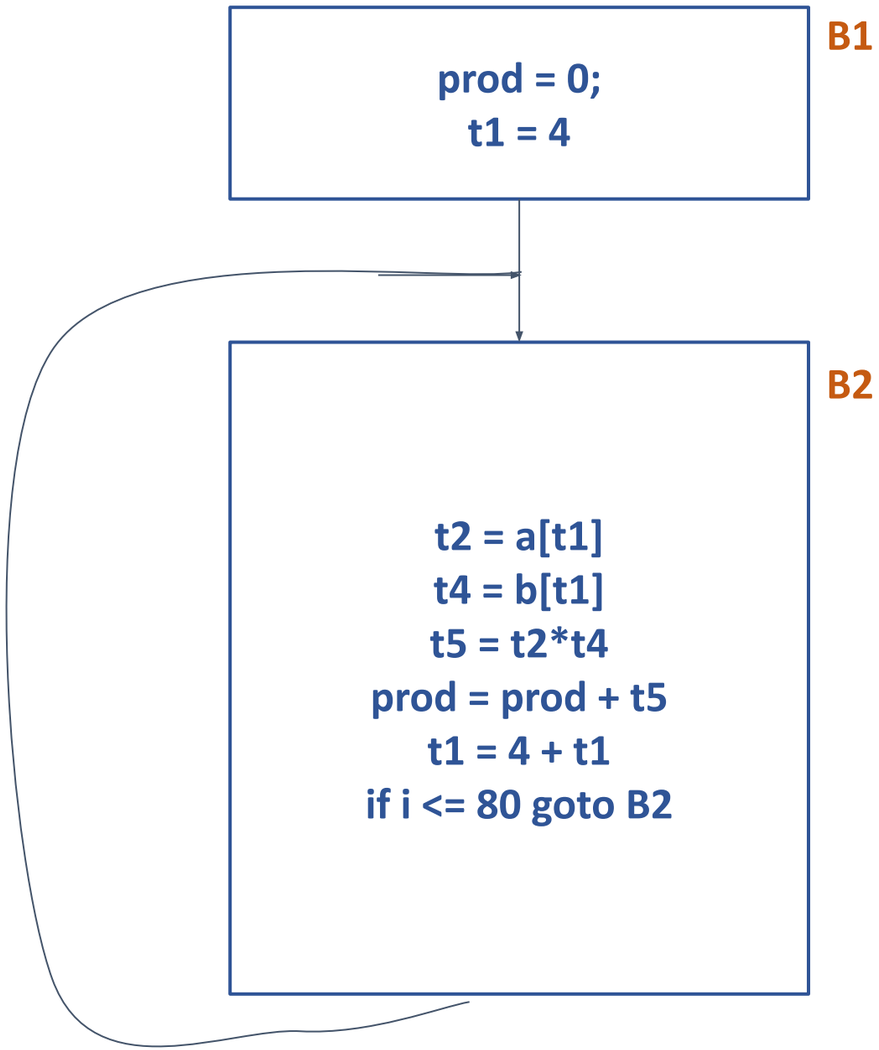
Contd.



Compiler Design

Local Optimization using DAG

Contd.



Optimized CFG

Local Optimization using DAG

Example

Apply local optimization on the following block using DAG, assume c and d are live on exit from the block.

```
a = 10
b = 4 * a
c = i * j + b
d = 15 * a * c
e = i
c = e * j + i * a
```

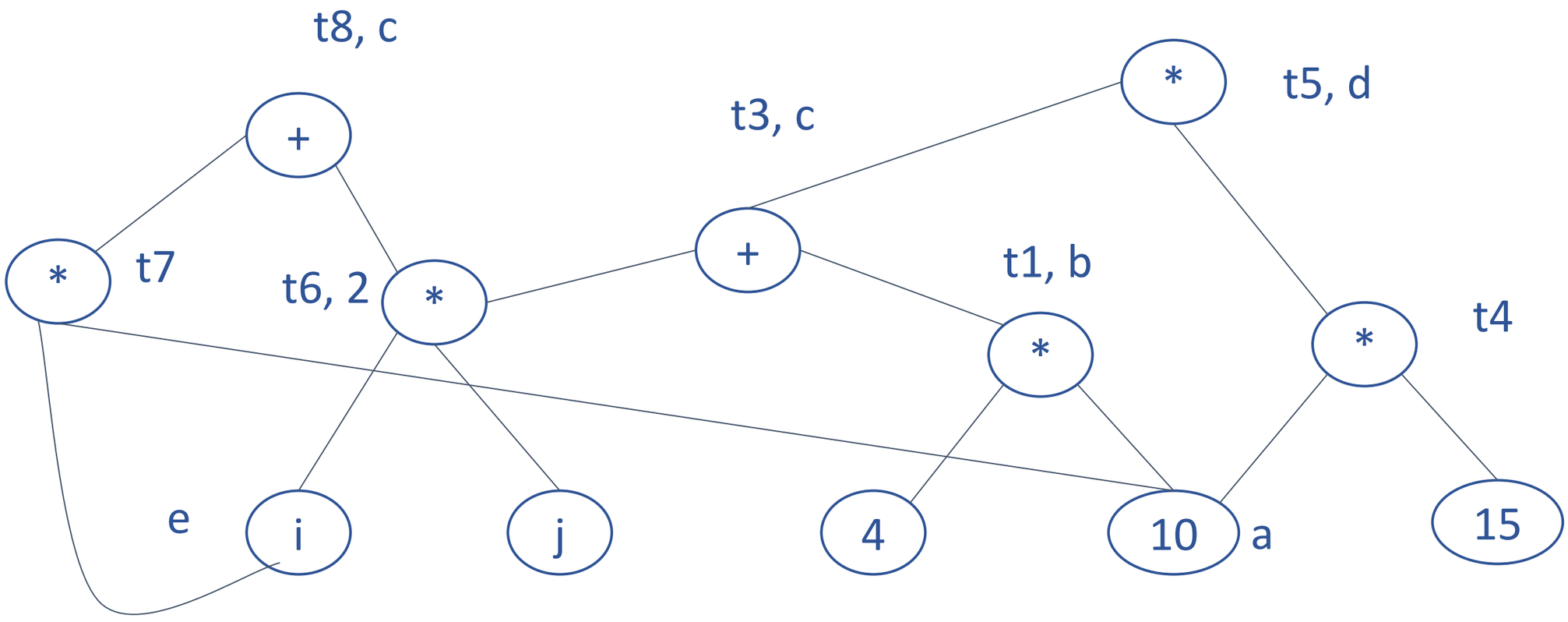
TAC

```
a = 10
t1 = 4 * a
b = t1
t2 = i * j
t3 = t2 + b
c = t3
t4 = 15 * a
t5 = t4 * c
d = t5
e = i
t6 = e * j
t7 = i * a
t8 = t6 + t7
c = t8
```

Compiler Design

Local Optimization using DAG

Contd.



Local Optimization using DAG

Contd.

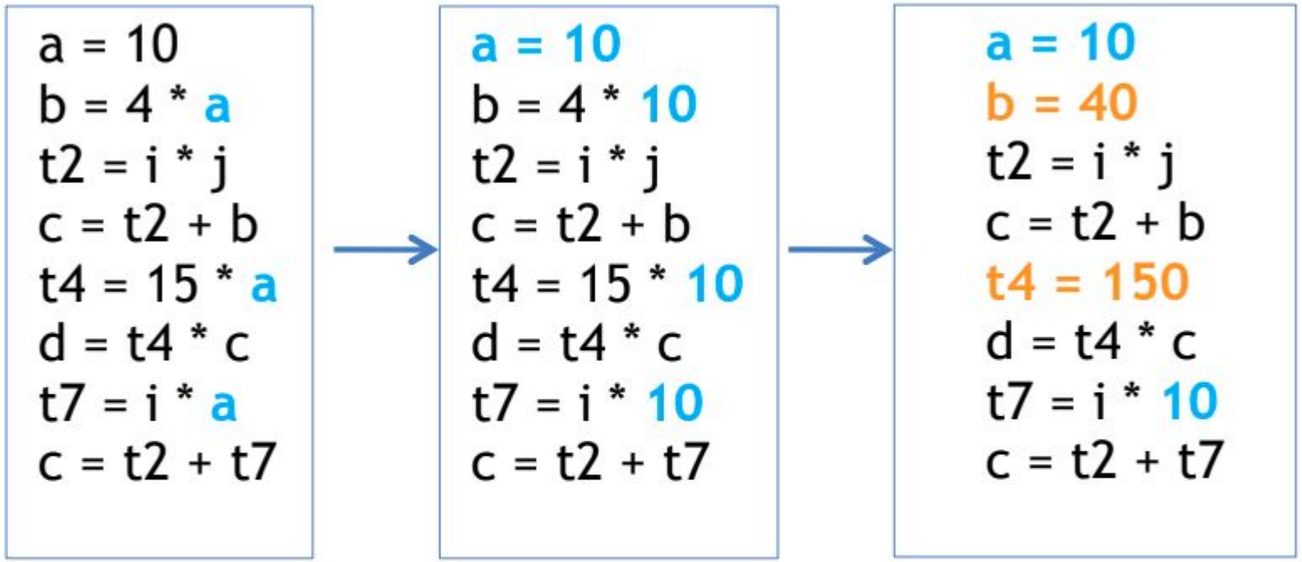
```
a = 10
t1 = 4 * a
b = t1
t2 = i * j
t3 = t2 + b
c = t3
t4 = 15 * a
t5 = t4 * c
d = t5
e = i
t6 = e * j
t7 = i * a
t8 = t6 + t7
c = t8
```

DAG

```
a = 10
b = 4 * a
t2 = i * j
c = t2 + b
t4 = 15 * a
d = t4 * c
t7 = i * a
c = t2 + t7
```

Local Optimization using DAG

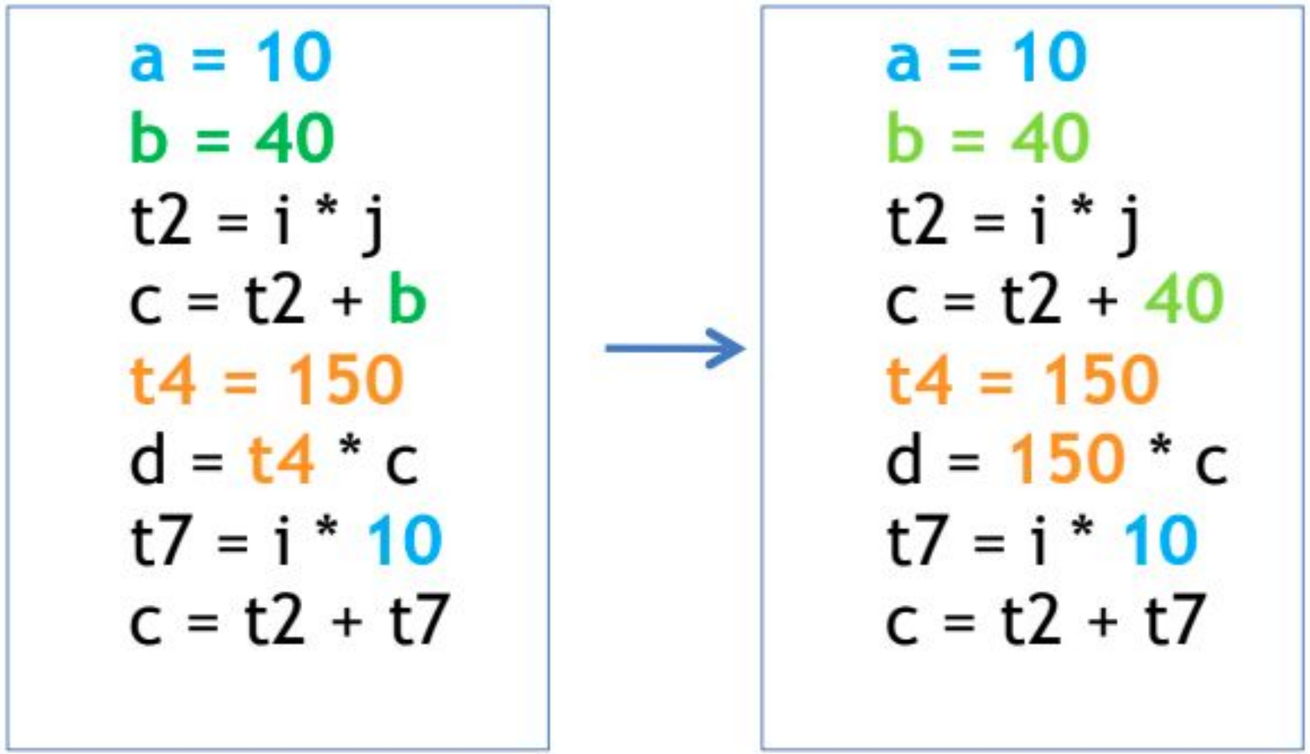
Contd.



Constant propagation - a Constant folding - b, t4

Local Optimization using DAG

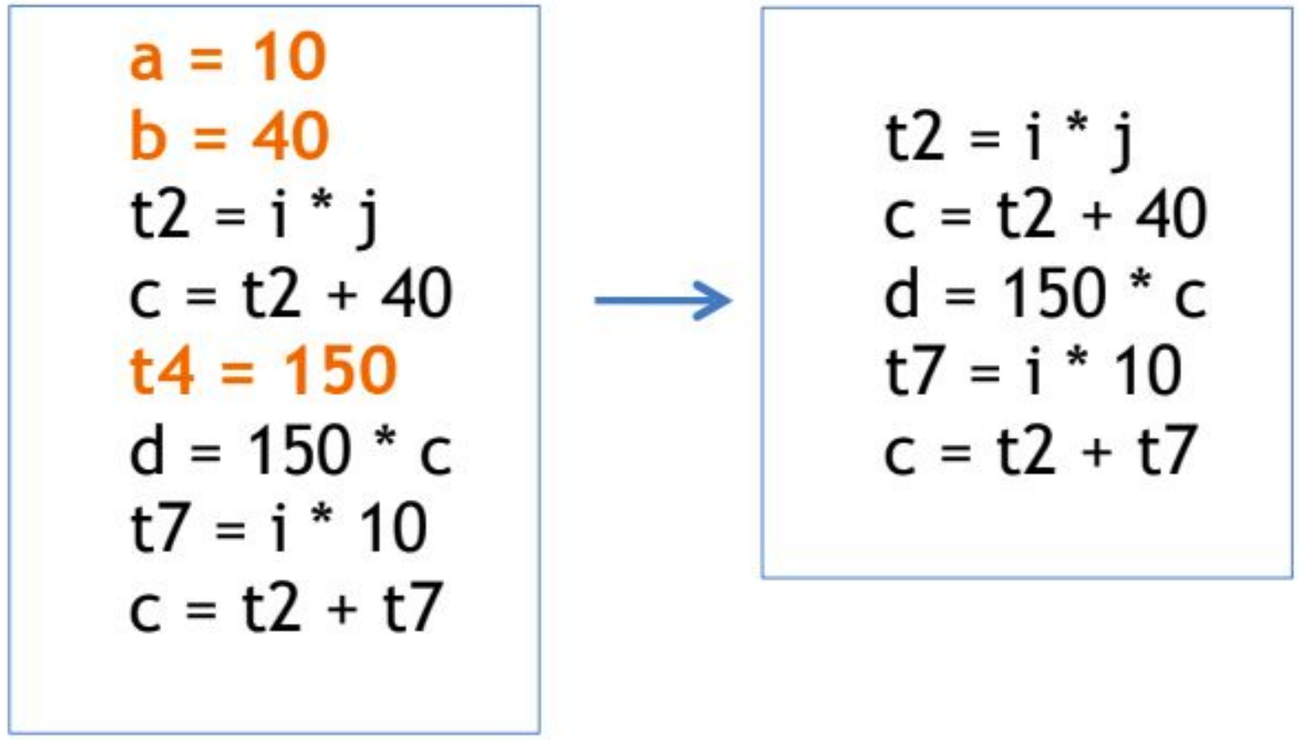
Contd.



Constant propagation- b, t4

Local Optimization using DAG

Contd.



Dead code elimination

Local Optimization using DAG

Contd.

```
a = 10
b = 40
t2 = i * j
c = t2 + 40
t4 = 150
d = 150 * c
t7 = i * 10
c = t2 + t7
```

DCE

```
t2 = i * j
c = t2 + 40
d = 150 * c
t7 = i * 10
c = t2 + t7
```

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 : Sree Pranavi G

Compiler Design

Unit 4: Code Optimization on CFG

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

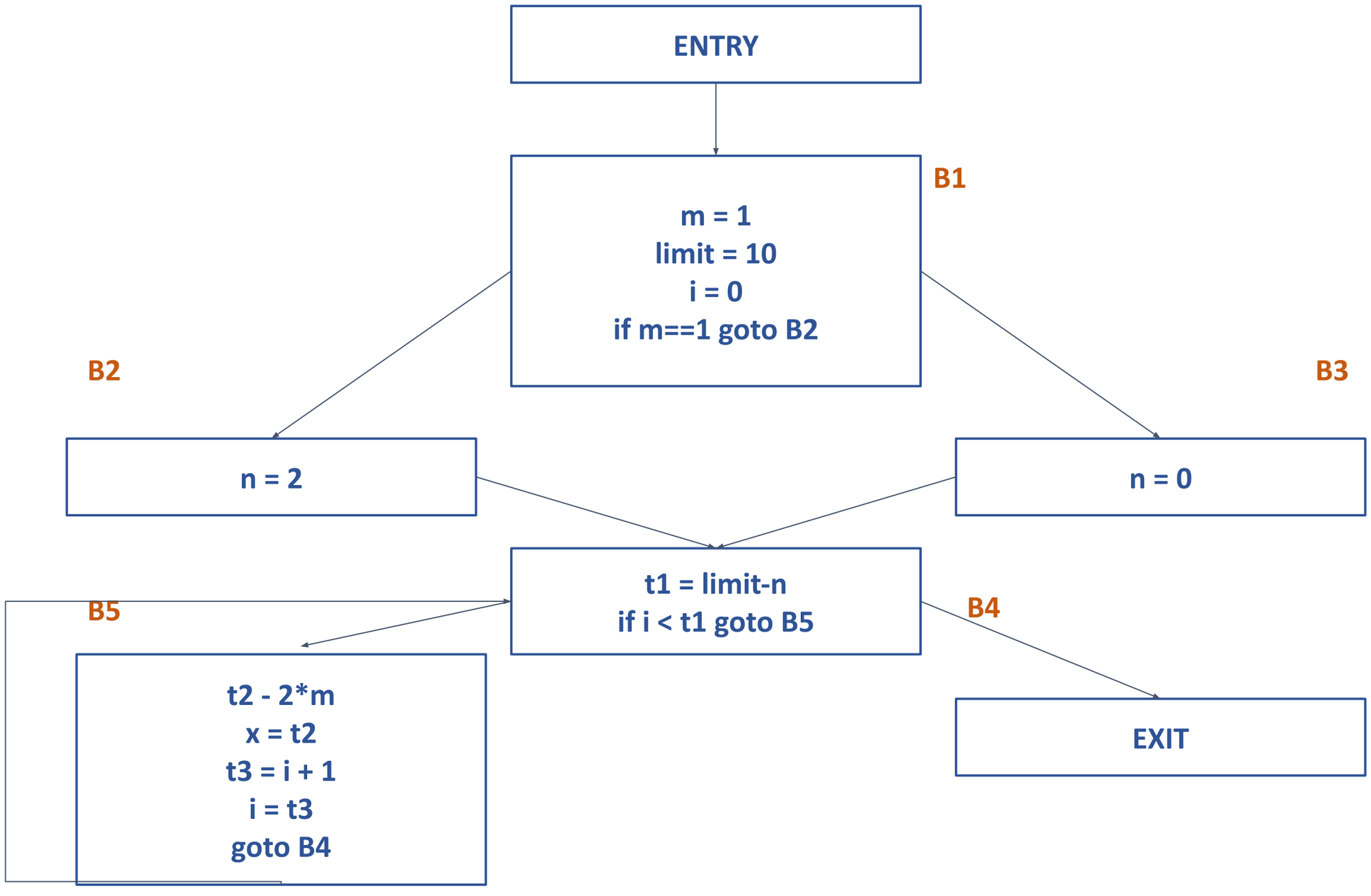
PES UNIVERSITY ONLINE

In this lecture, you will learn about -

- **Global Optimization on CFG**

Compiler Design

Optimization on CFG



Compiler Design

Optimization on CFG

Steps to optimizing CFG

1. In Block 1 , Since the value of m is known as 1, we can evaluate the condition:

   if m==1 goto B2

   making B3 unreachable and hence removing the B3 as it contains dead code.

2. Next we can move n = 2 in B2 to block B1 and remove B2.
3. Since value of limit and n is known we can propagate the value to B4, and constant fold the value t1 as 10 -2 = 8.
4. Next we can propagate the value of t1 to the statement:

   if i < t1 goto B5 ➔ if i < 8 goto B5

Optimization on CFG

Steps to optimizing CFG

5. After this as t1 is not being used anywhere else, t1 = 8 becomes a dead code and hence can be eliminated.
6. In Block B5, we must draw the Dag and the optimized code using DAG is :

   x = 2 * m

   i = i + 1

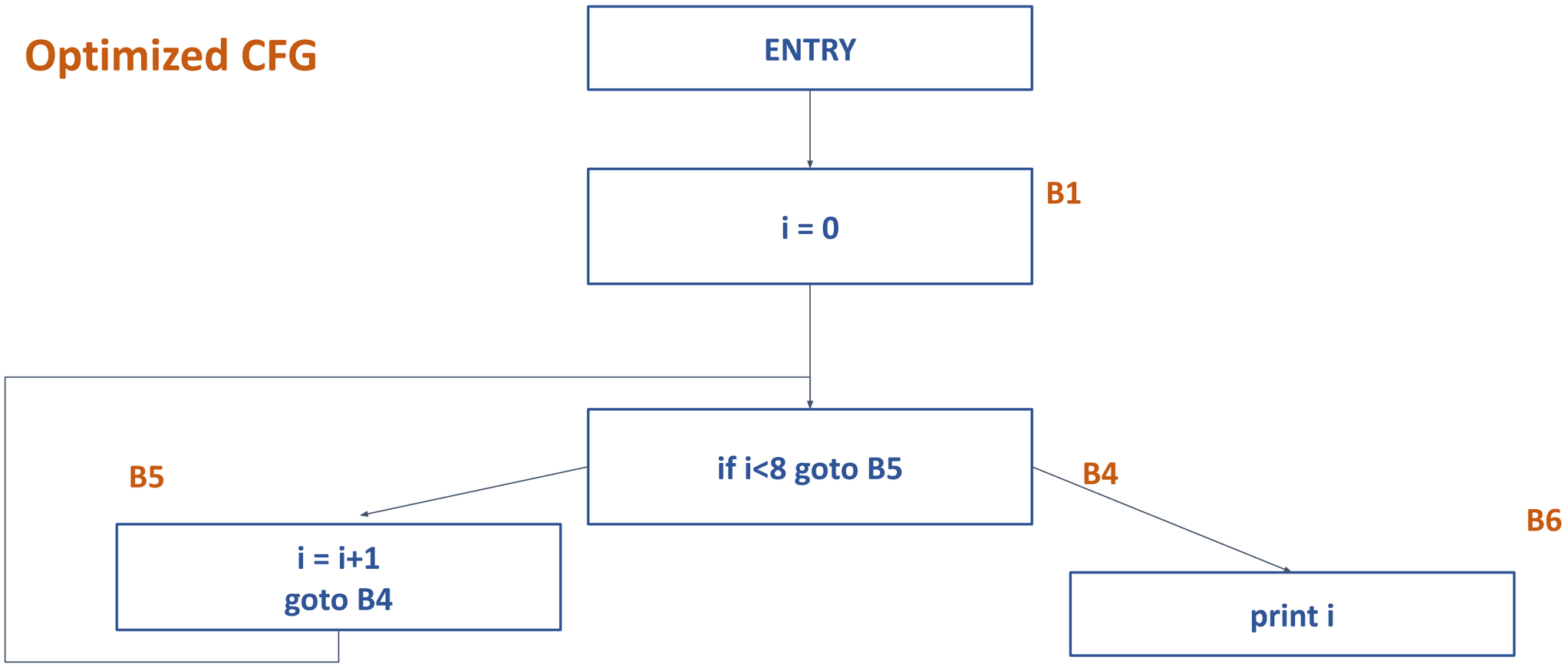
   goto B4
7. Since x = 2 * m is loop invariant we move it to Block 1.
8. If required we can constant fold x by propagating the value of m, since x is not being used anywhere we can eliminate x = 2 * m.
9. Similarly in Block 1 we can eliminate the dead code :

   limit = 0; m = 1, n = 2.

Compiler Design

Optimization on CFG

Optimized CFG



if we take into consideration loop unrolling, then Compiler will just keep block B6 and rest of the blocks will be removed

Optimization on CFG

Example 2

```
for (i = 0; i<n; i++) {
    for (j=0; j<n; j++){
        if (i%2){
            P[i][j] = 10*j + 5*i;
            Q[i][j] = 10 + 10*j;
        }
    }
}
//print(array P);
//print (array Q);
```

Example 2 - Questions

Which one of the following is false?

a) The code contains loop invariant computation

b) There is scope of common sub-expression elimination in this code

c) There is scope of strength reduction in this code

d) There is scope of dead code elimination in this code

Justify each statement which is true with proper explanation,

Optimization on CFG

Example 2 - Solution

- 5 * i and i % 2can be moved out of the inner loop - loop invariant computations, hence (a) is true.
- 10 * j is a candidate for common sub expression elimination hence (b) is true.
- 10*j can be replaced with t1 = -10 before the j loop and place

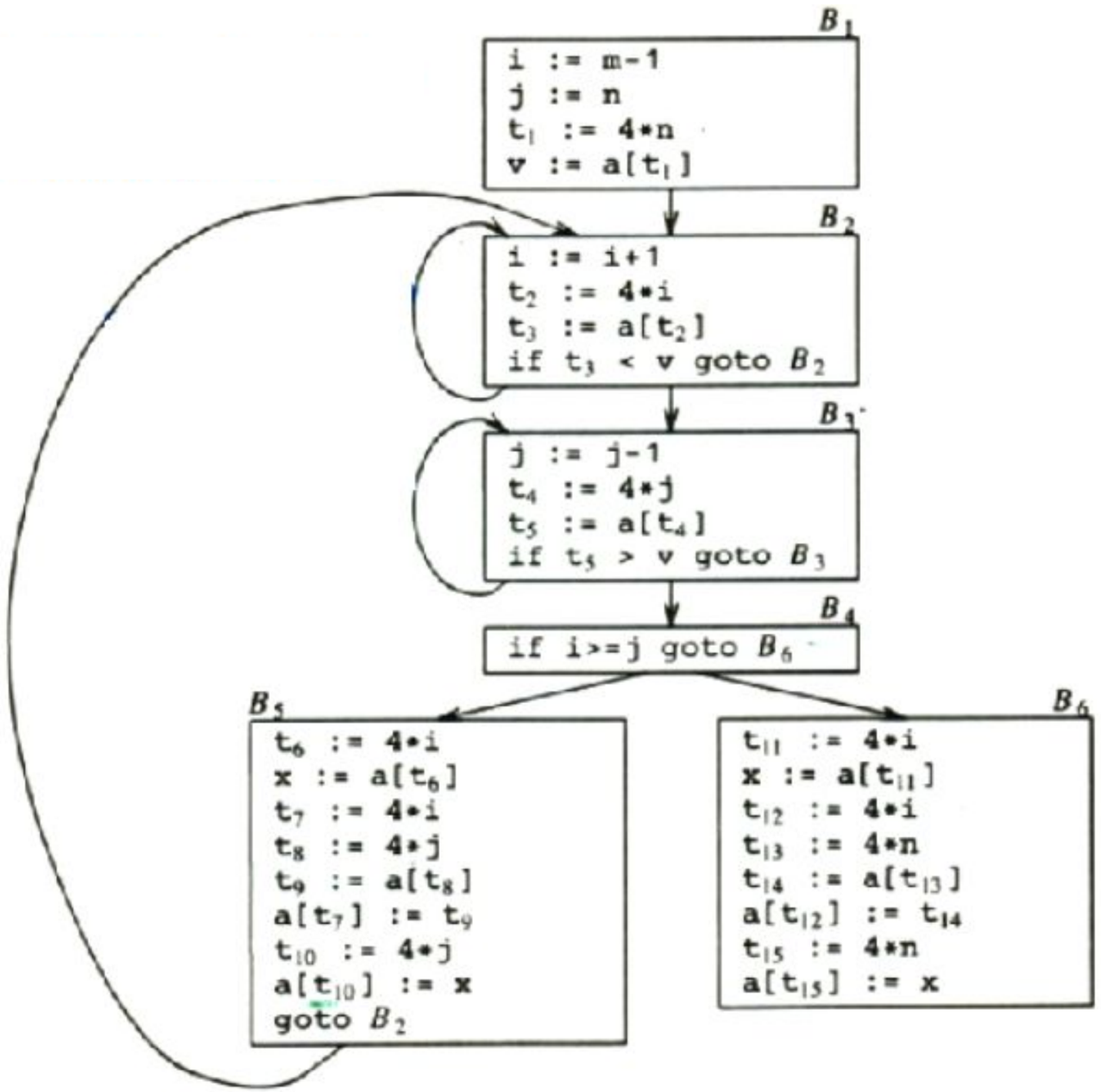
  t1 = t1 + 10 wherever 10 * j is being computed. Similarly for 5 * i.

  Hence (c) is true as there is a scope for strength reduction in this code.
- No dead code present in the given piece of code, hence (d) is false.

Compiler Design

Optimization on CFG

Example 3



Optimization on CFG

Example 3 contd.

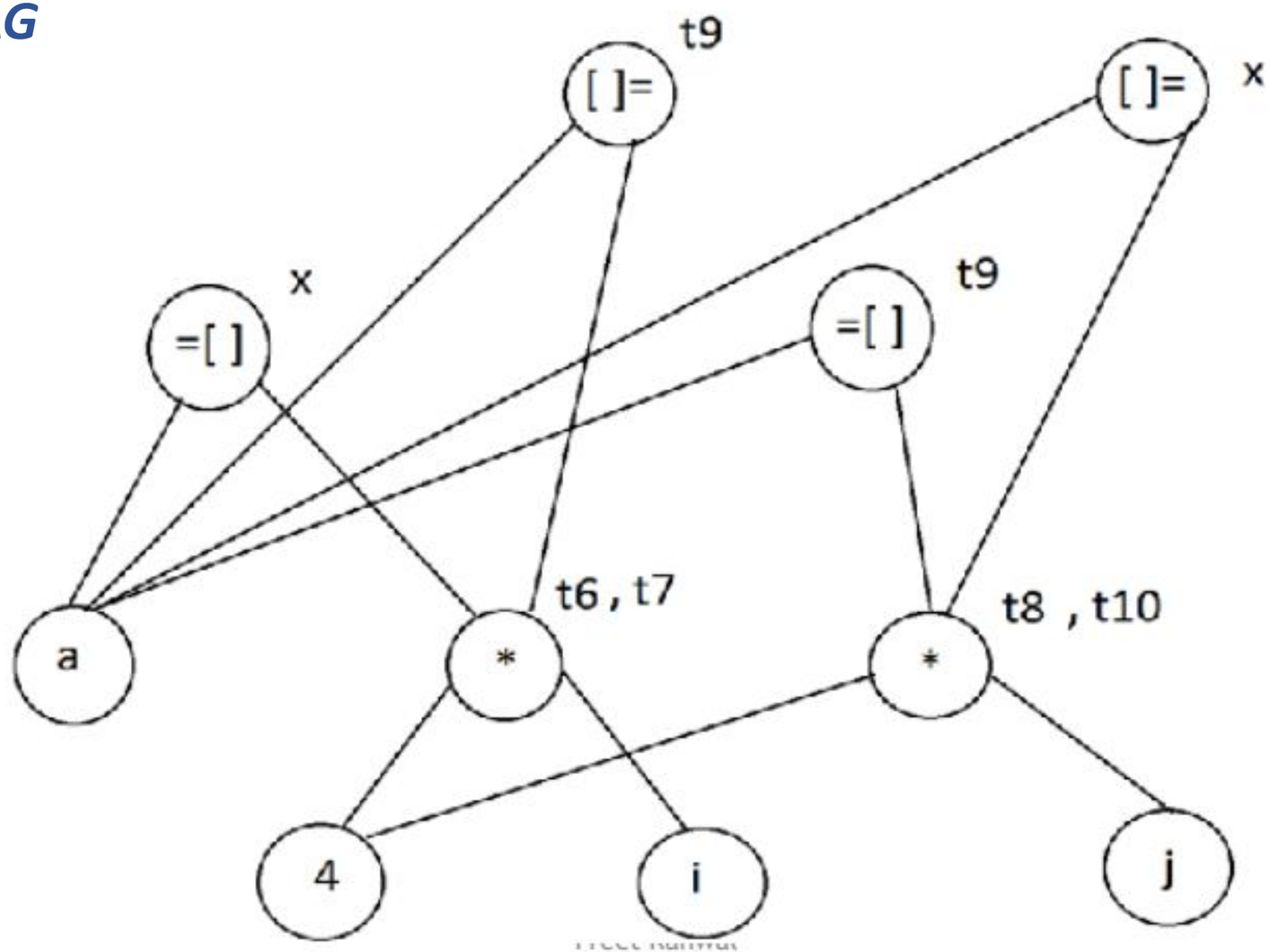
Block B5

```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
```

Optimization on CFG

Example 3 contd.

DAG



*t6 = 4 * i*

x = a[t6]

*t8 = 4 * j*

t9 = a[t8]

a[t6] = t9

a[t8] = x

goto B2

Optimization on CFG

Example 3 contd.

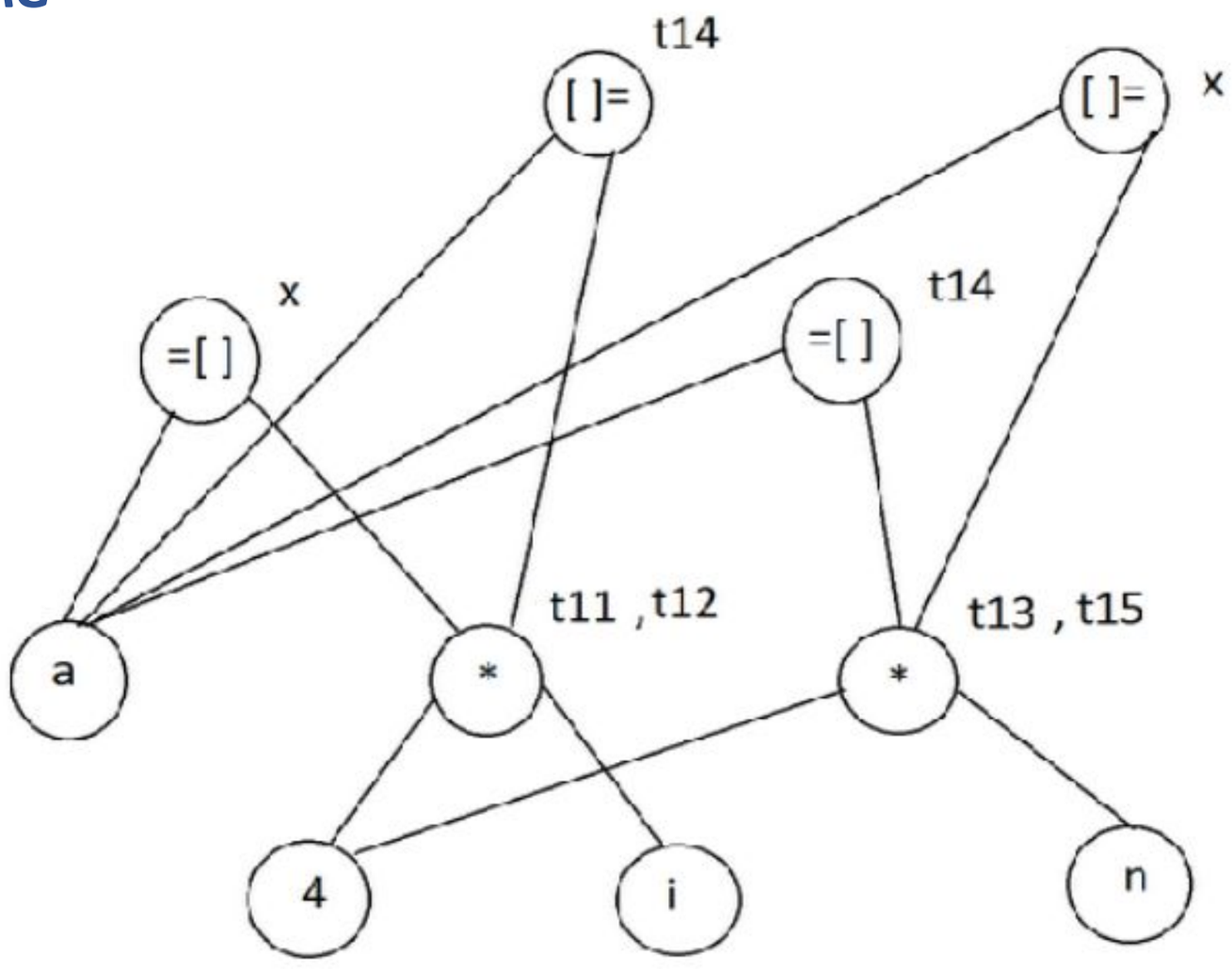
Block B6

```
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x
```

Optimization on CFG

Example 3 contd.

DAG



t11 = 4 * i

x = a[t11]

t13 = 4 * n

t14 = a[t13]

a[t11] = t14

a[t13] = x

Optimization on CFG

Example 3 contd.

Block B6

*t11 = 4 * i*

x = a[t11]

*t12 = 4 * i*

*t13 = 4 * n*

t14 = a[t13]

a[t12] = t14

*t15 = 4 * n*

a[t15] = x

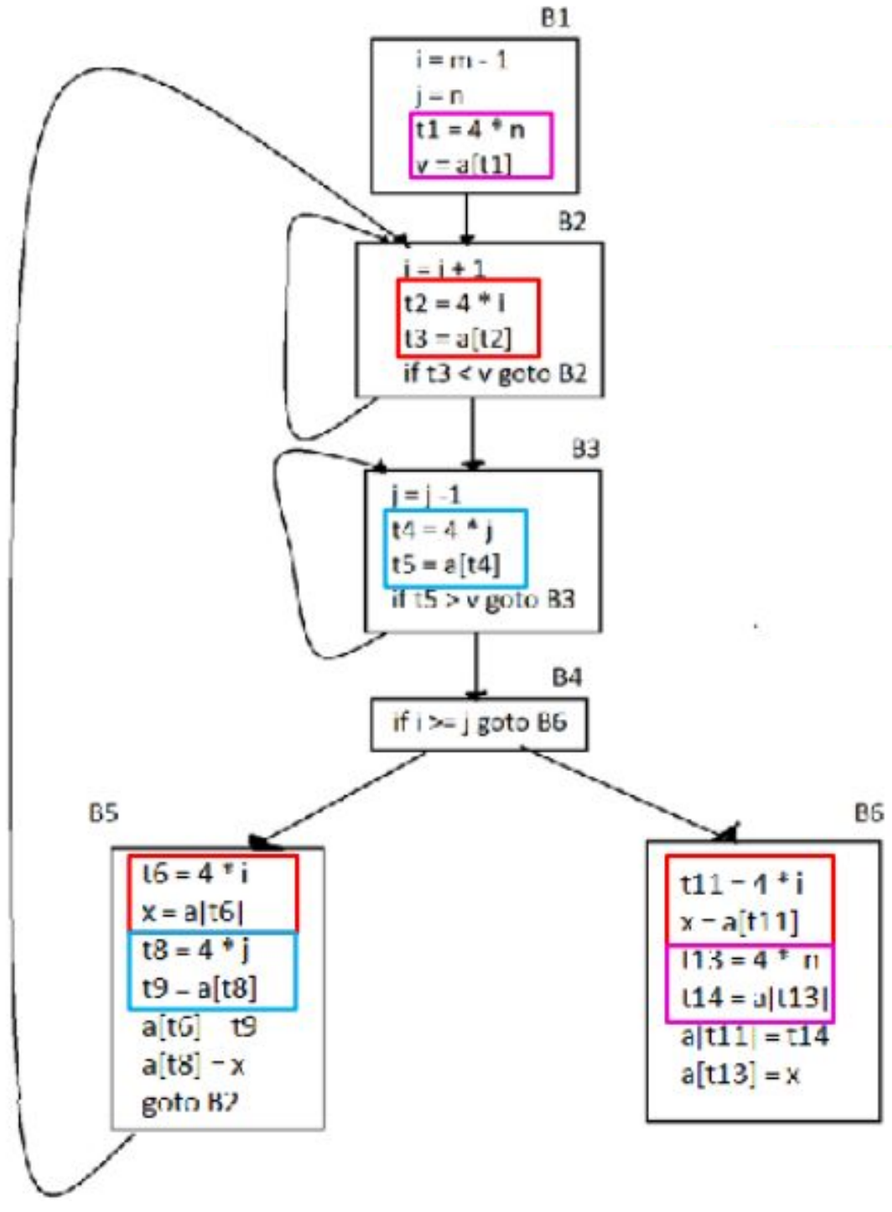
Compiler Design

Optimization on CFG

Example 3 contd.

Global optimization

CSE



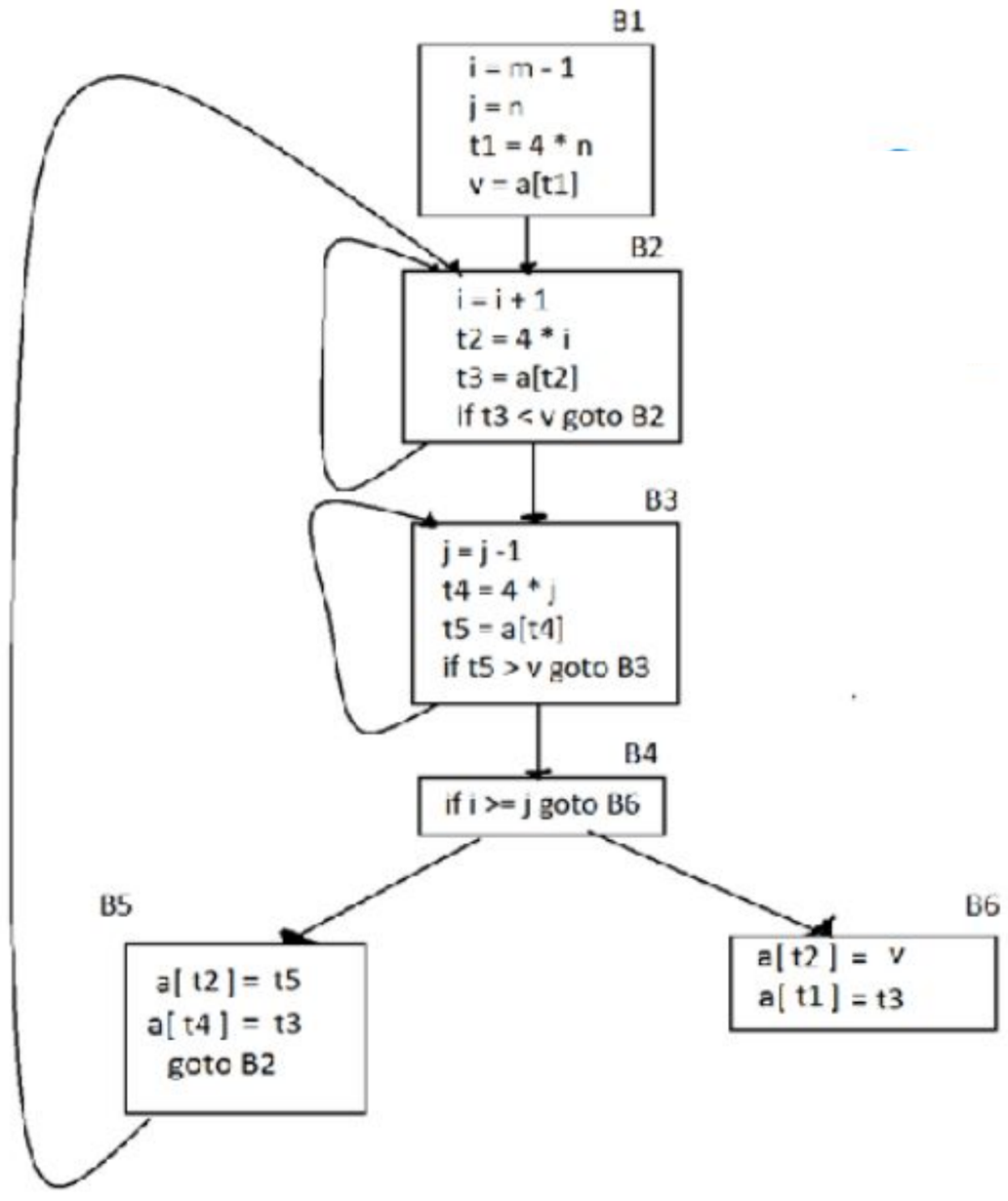
Compiler Design

Optimization on CFG

Example 3 contd.

Global optimization

Can we optimize loops?

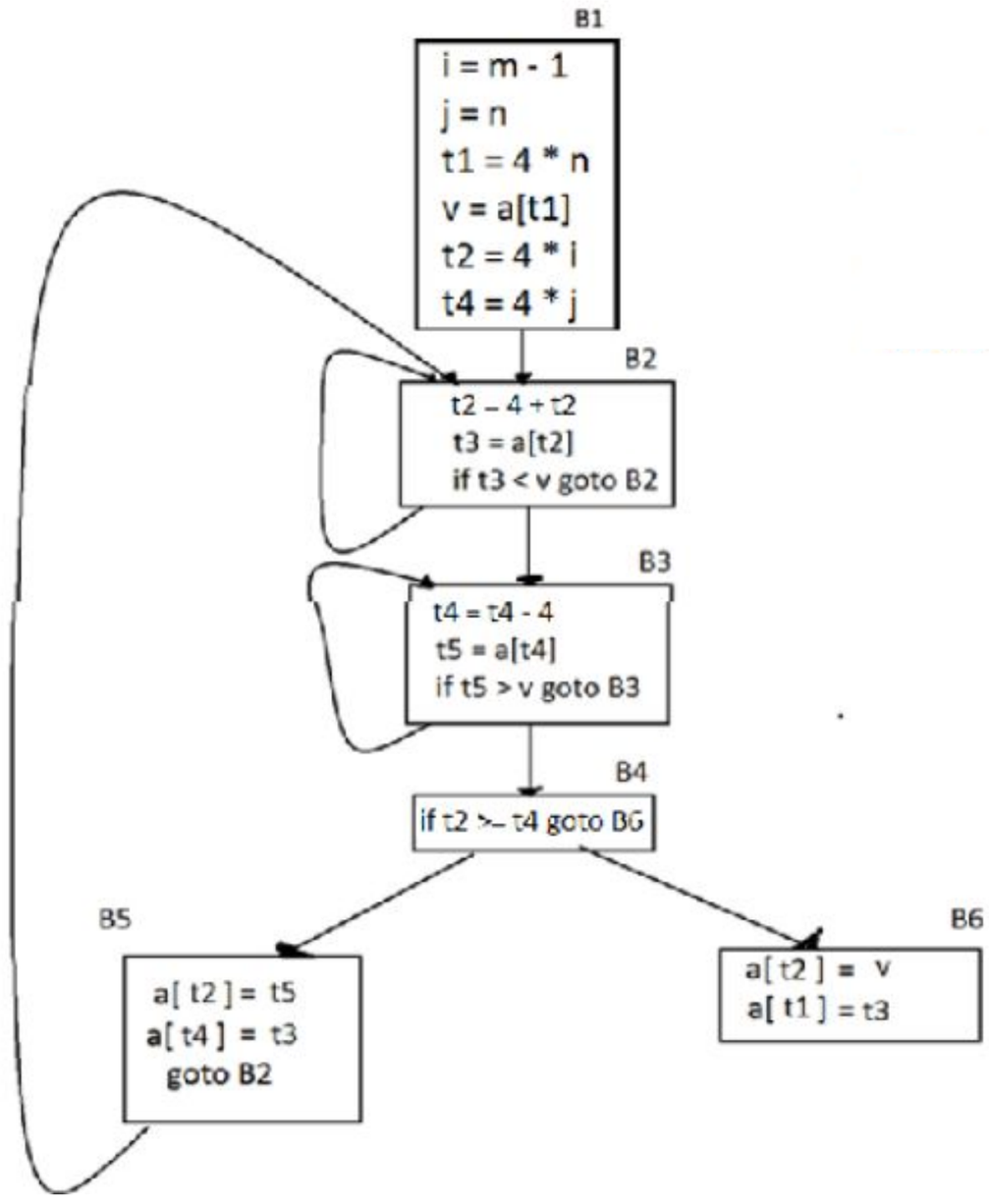


Compiler Design

Optimization on CFG

Example 3 contd.

Optimized code



Compiler Design

Optimization on CFG

Homework

Convert the following code to CFG using 3AC & optimize the CFG.

```
for (i = 0, i<n; i++) {
    for (j=0; j<n; j++){
        if (i%2){
            P = 10*j + 5*i;
            Q = 10 + 10*j;
    }
    }
}
```

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 4: Live-Variable Analysis

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Lecture Overview

PES UNIVERSITY ONLINE

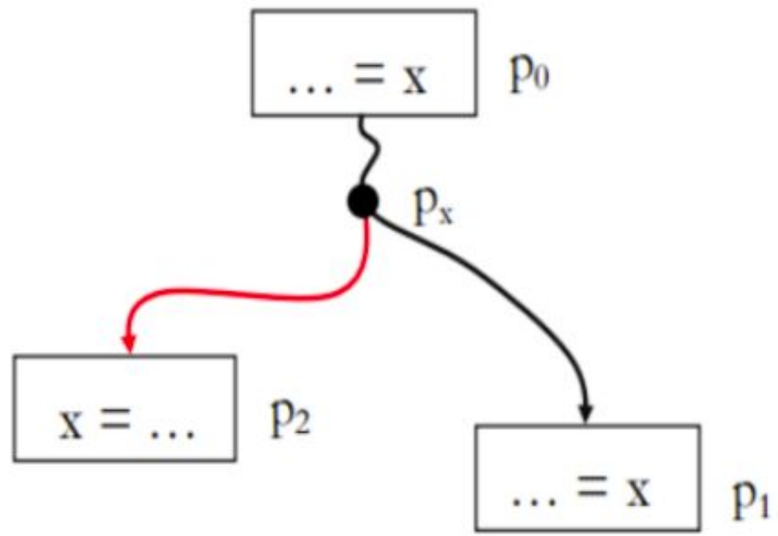
In this lecture, you will learn about -

- What is Live variable analysis?
- Liveness
- Applications
- Data flow values and equations
- Live-Variable Analysis Algorithm
- Examples

What is Live-Variable Analysis?

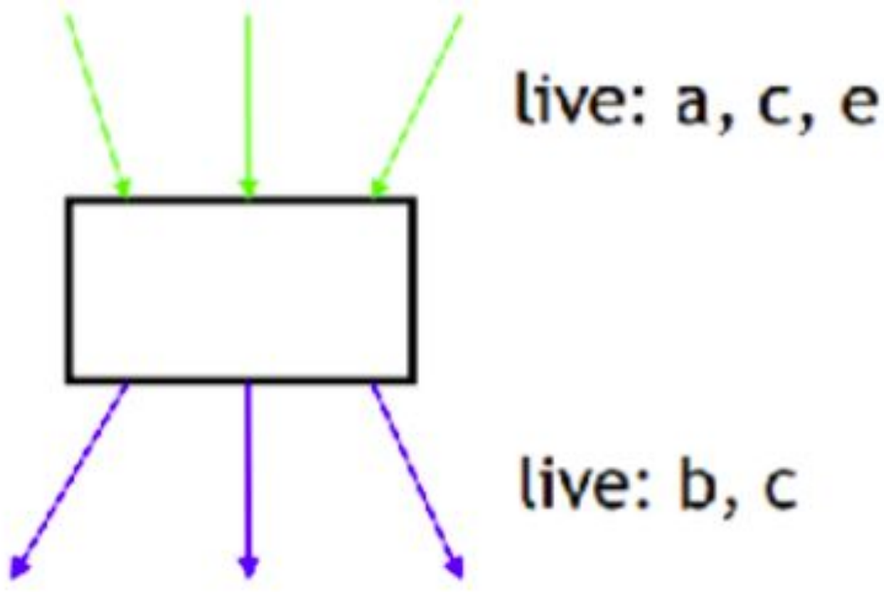
- Live variable analysis is based on the following definition -
  - For a variable **x** and program point **p** determine if the value of **x** at **p** can still be used along some path starting at **p**.
  - If yes, **x** is live at **p**.
  - If not, **x** is dead at **p**.

- In the given illustration -
  - At point p_0 the **x** variable is live if there is a path from p_0 where **x** is used.
  - Beyond p_x towards p_2 the value of **x** is no longer needed and is dead.



Liveness

- Liveness is associated with edges of control flow graph, not nodes (statements).



Applications

- **Register Allocation** - If a variable is dead at a given point **p**, we can reuse its storage, i.e, the register it occupies (if any).
- **Dead code elimination** - If variable **x** is not live after an assignment **x = ...**, then the assignment is redundant and can be deleted as dead code.

Data-flow values

Live variable Analysis algorithm requires the computation of the following data-flow values -

- **use[n] : set of variables used by Node n**
- **def [n] : set of variables defined by Node n**
- **in[n] : variables live on entry to Node n**
- **out[n] : variables live on exit from Node n**

Data-flow equations

The data-flow values are then used to compute IN and OUT sets for each basic block B using the data flow equations-

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} (IN[S])$$

$$IN[B] = use[B] \cup (OUT[B] - def[B])$$

Live-Variable Analysis Algorithm

Algorithm

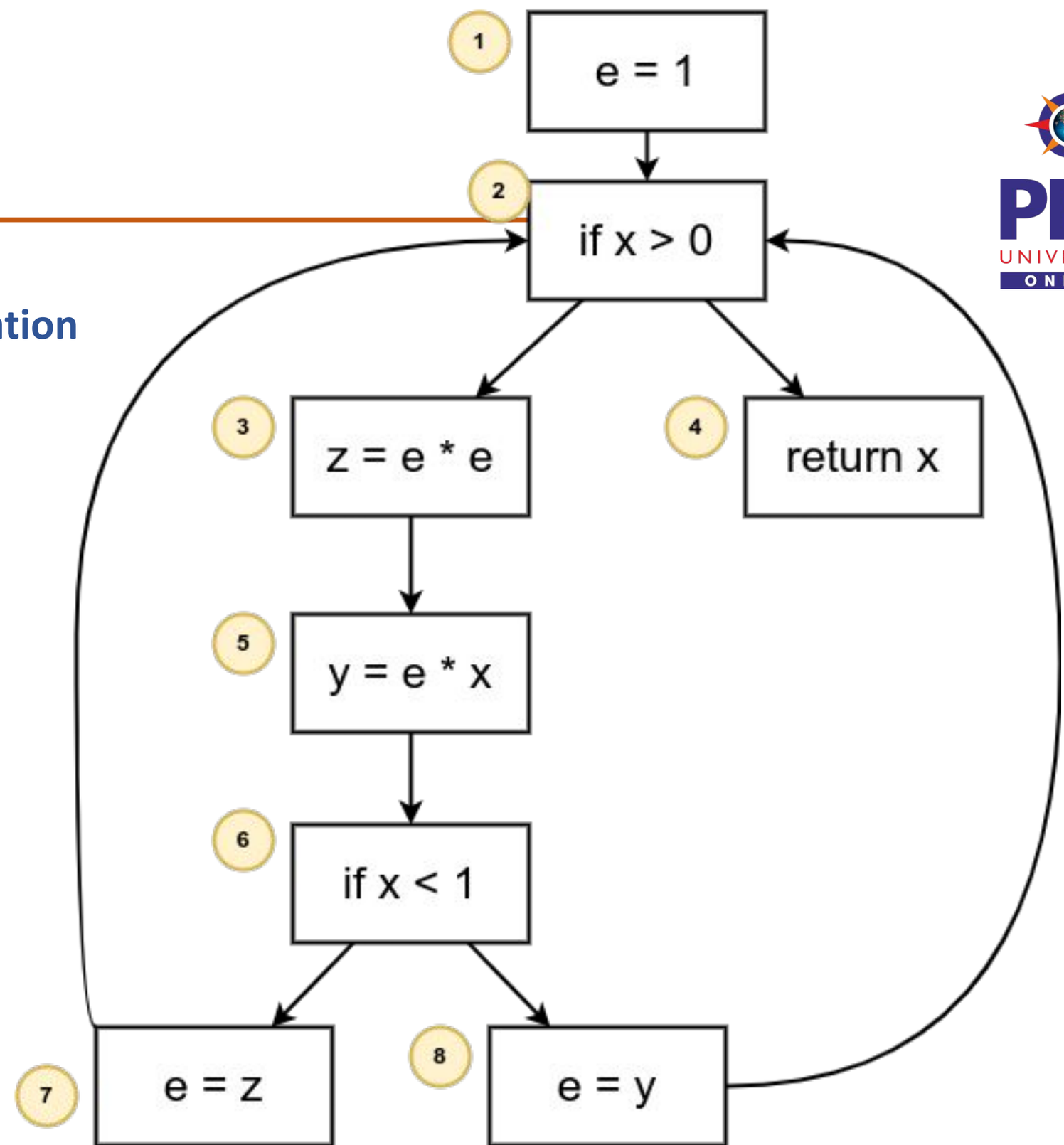
for each node n in CFG
    in[n] = ∅; out[n] = ∅ } Initialize solutions

repeat
    for each node n in CFG **in reverse topsort order**
        in'[n] = in[n] } Save current results
        out'[n] = out[n]
        out[n] = $\bigcup_{s \in succ[n]}$ in[s] } Solve data-flow equations
        in[n] = use[n] ∪ (out[n] – def[n])
until in'[n]=in[n] and out'[n]=out[n] for all n } Test for convergence

Compiler Design

Example 1

Compute Liveness Information for the following CFG -



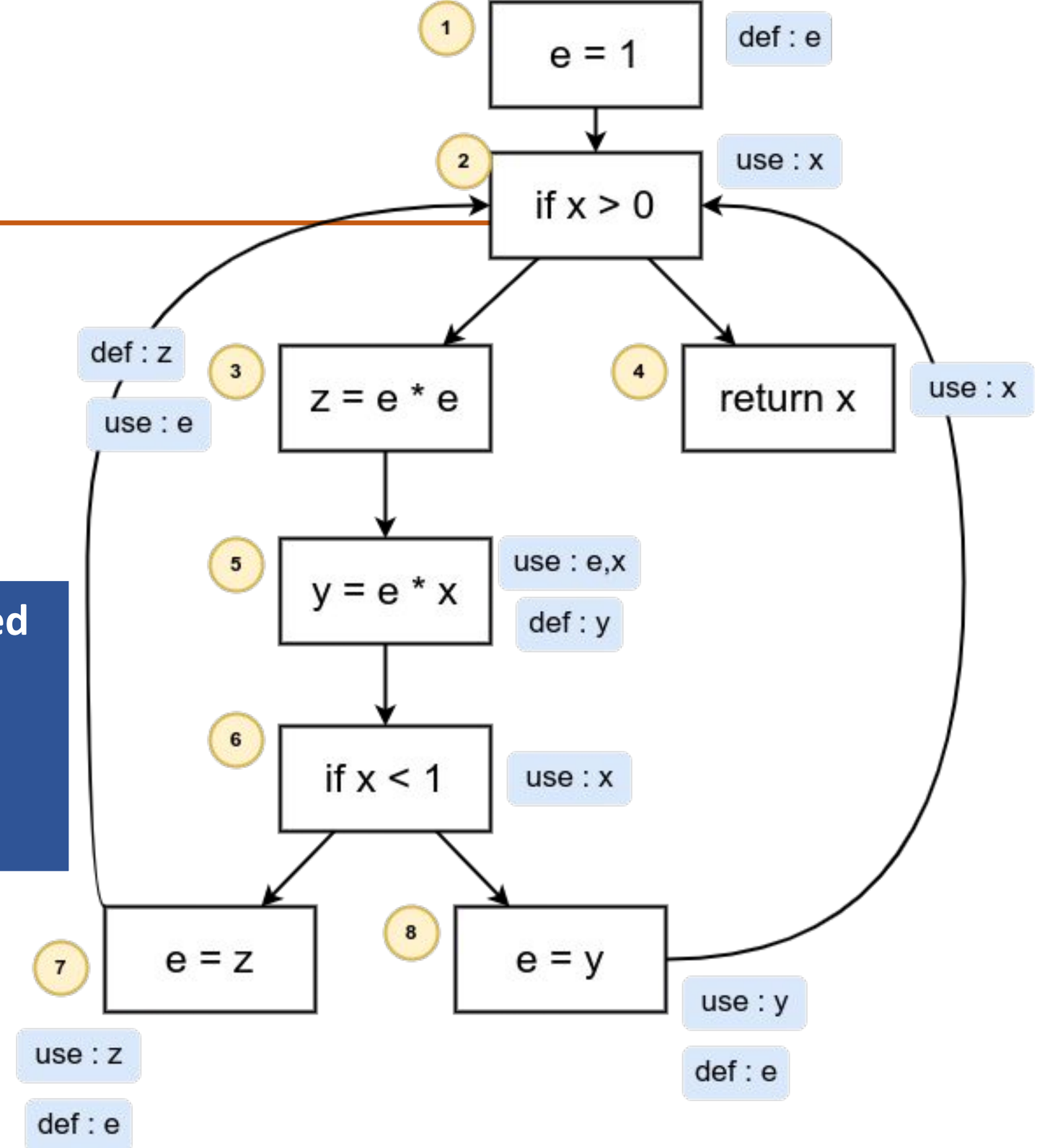
Compiler Design

Example 1 - Solution

Step 2 - Compute use and def for each basic block.

use[n] : set of variables used by Node n

def [n] : set of variables defined by Node n



Compiler Design

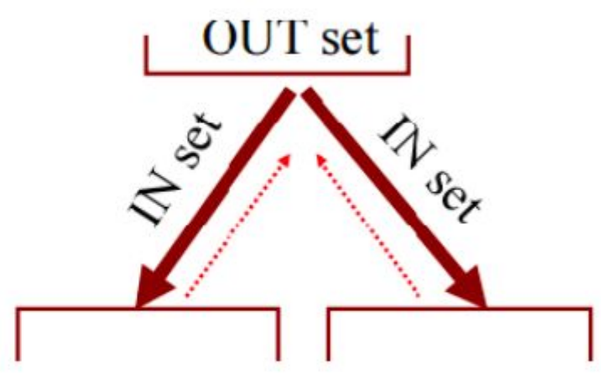
Dataflow equations

Step 3 - Compute IN and OUT for each basic block

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

OUT [B] = $\bigcup_{\text{S is a successor of B}}$ (IN[S])

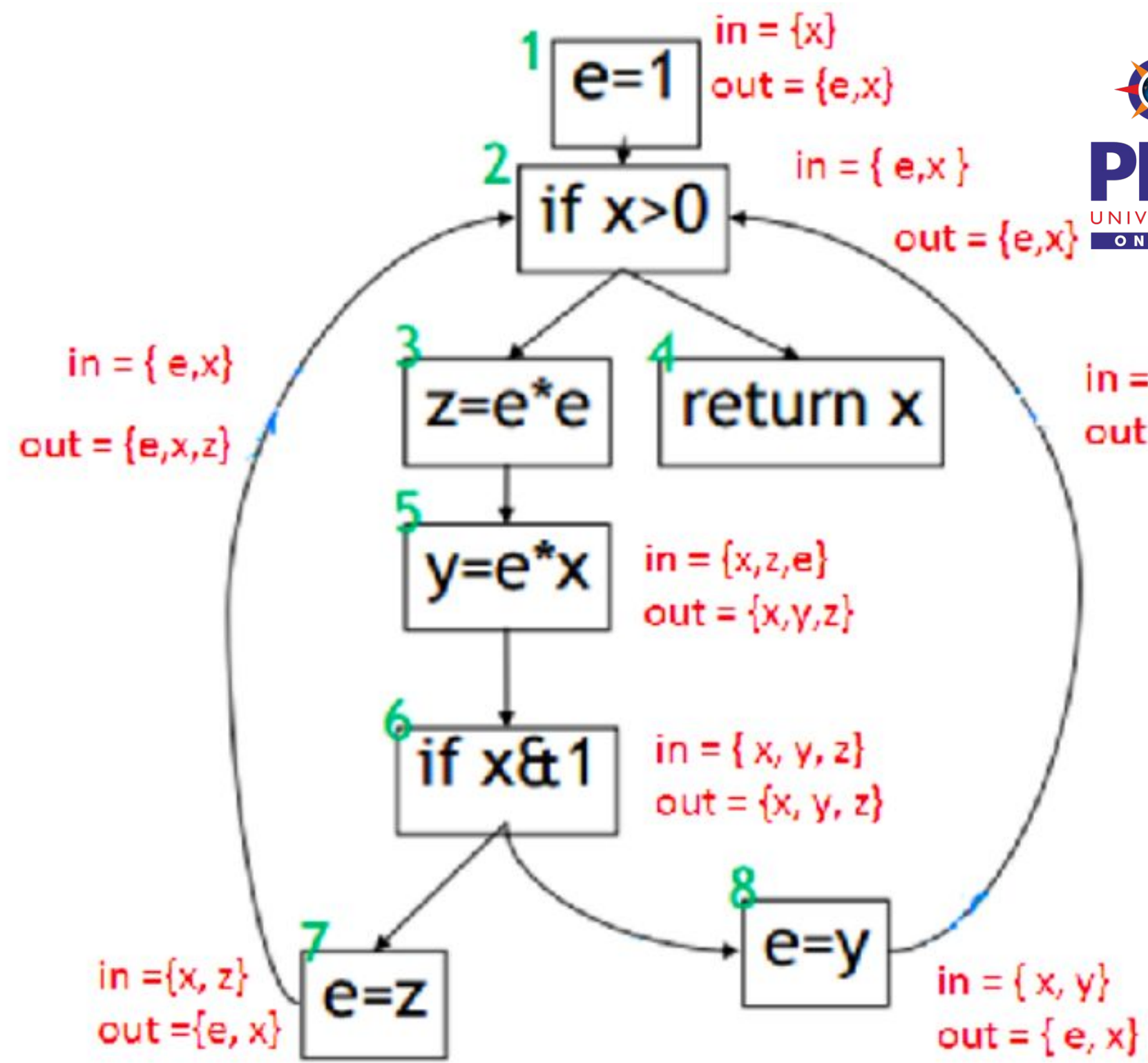
IN [B] = use[B] $\cup$ (OUT[B] - def[B])



$$IN(B) = Use(B) \cup (OUT(B) - Def(B))$$

Compiler Design

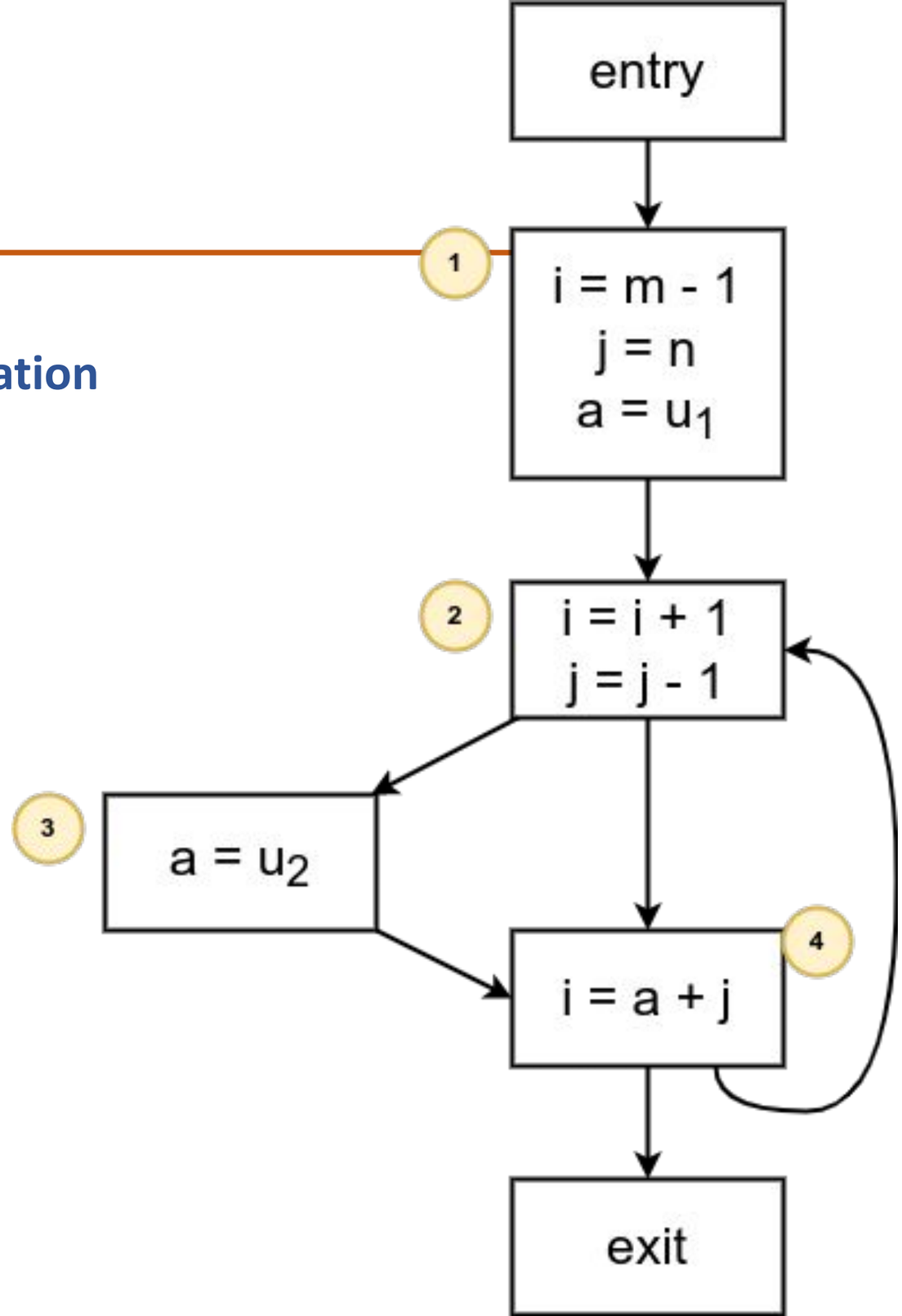
Example 1 Solution



Compiler Design

Example 2

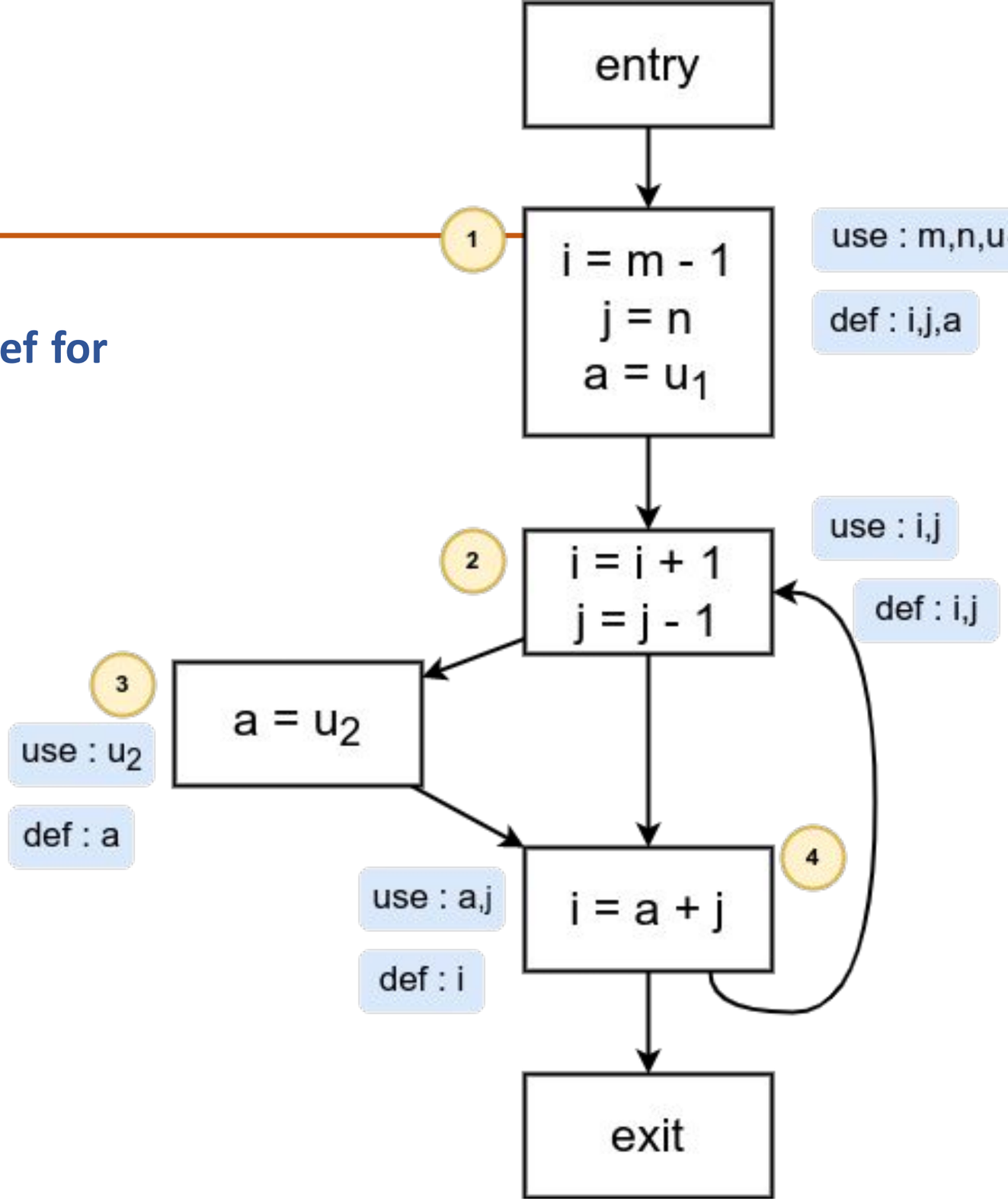
Compute Liveness Information for the following CFG -



Compiler Design

Example 2 - Solution

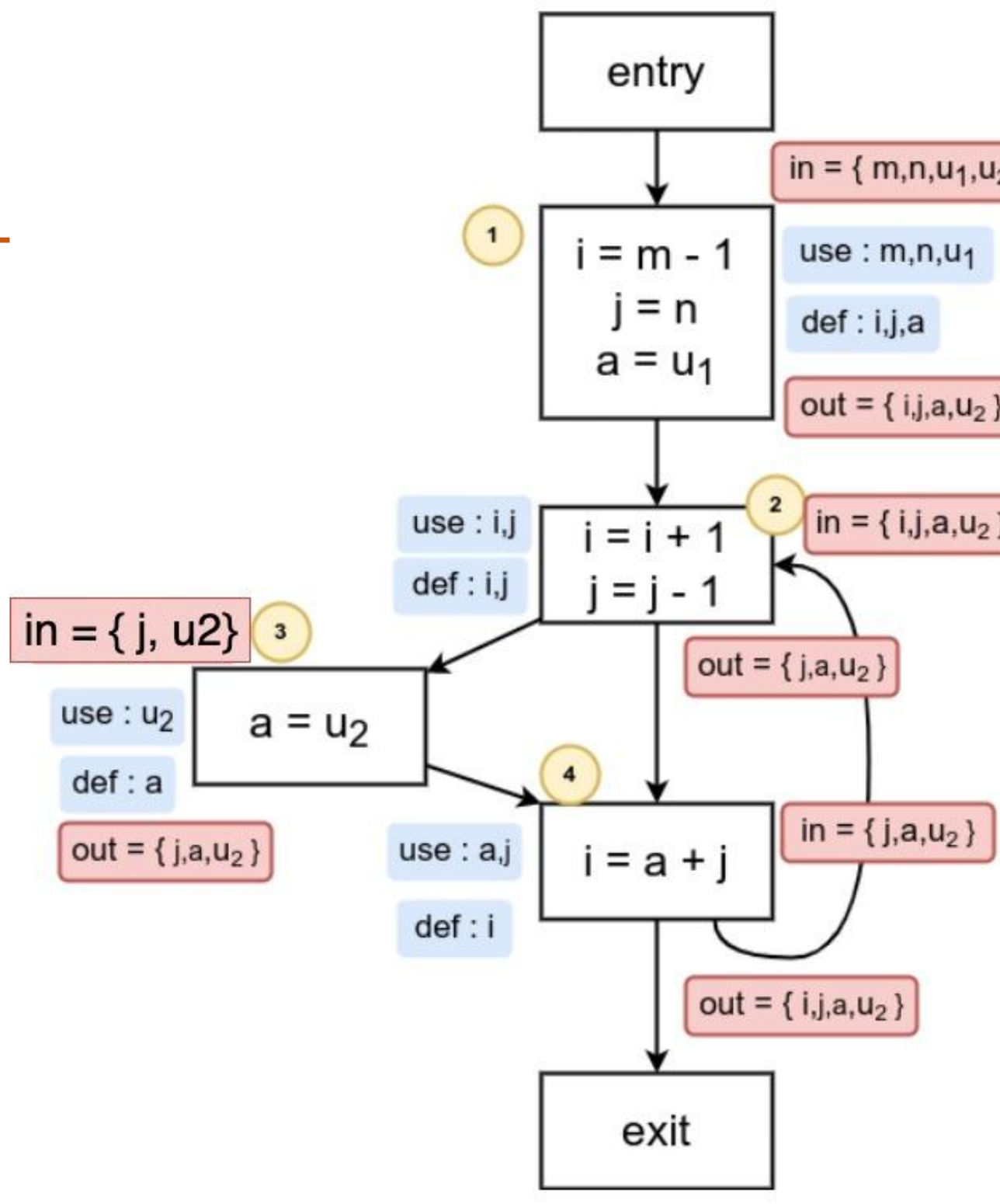
Step 2 - Compute use and def for each basic block.



Compiler Design

Example 2 - Solution

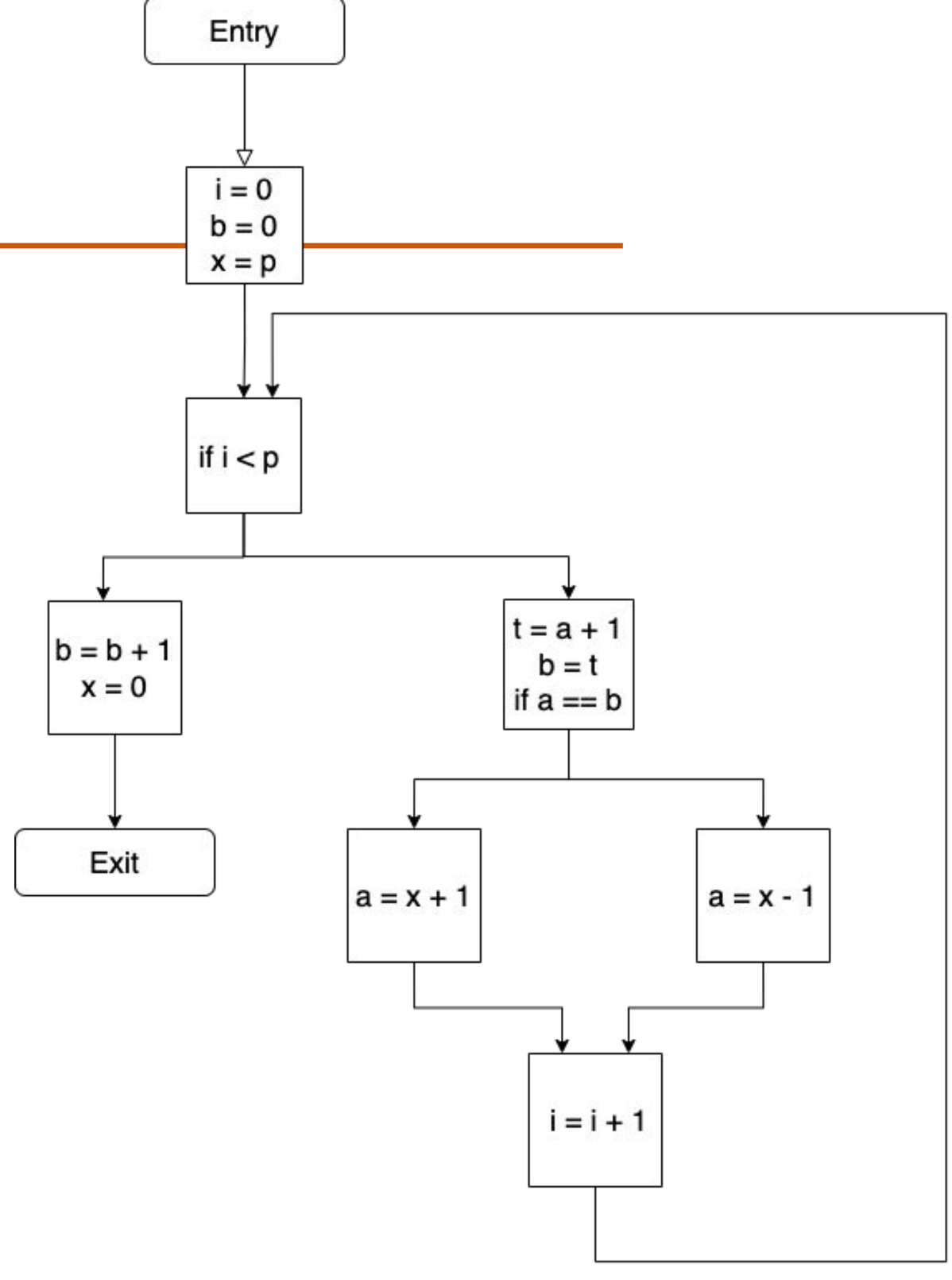
Step 3 - Compute IN and OUT for each basic block



Compiler Design

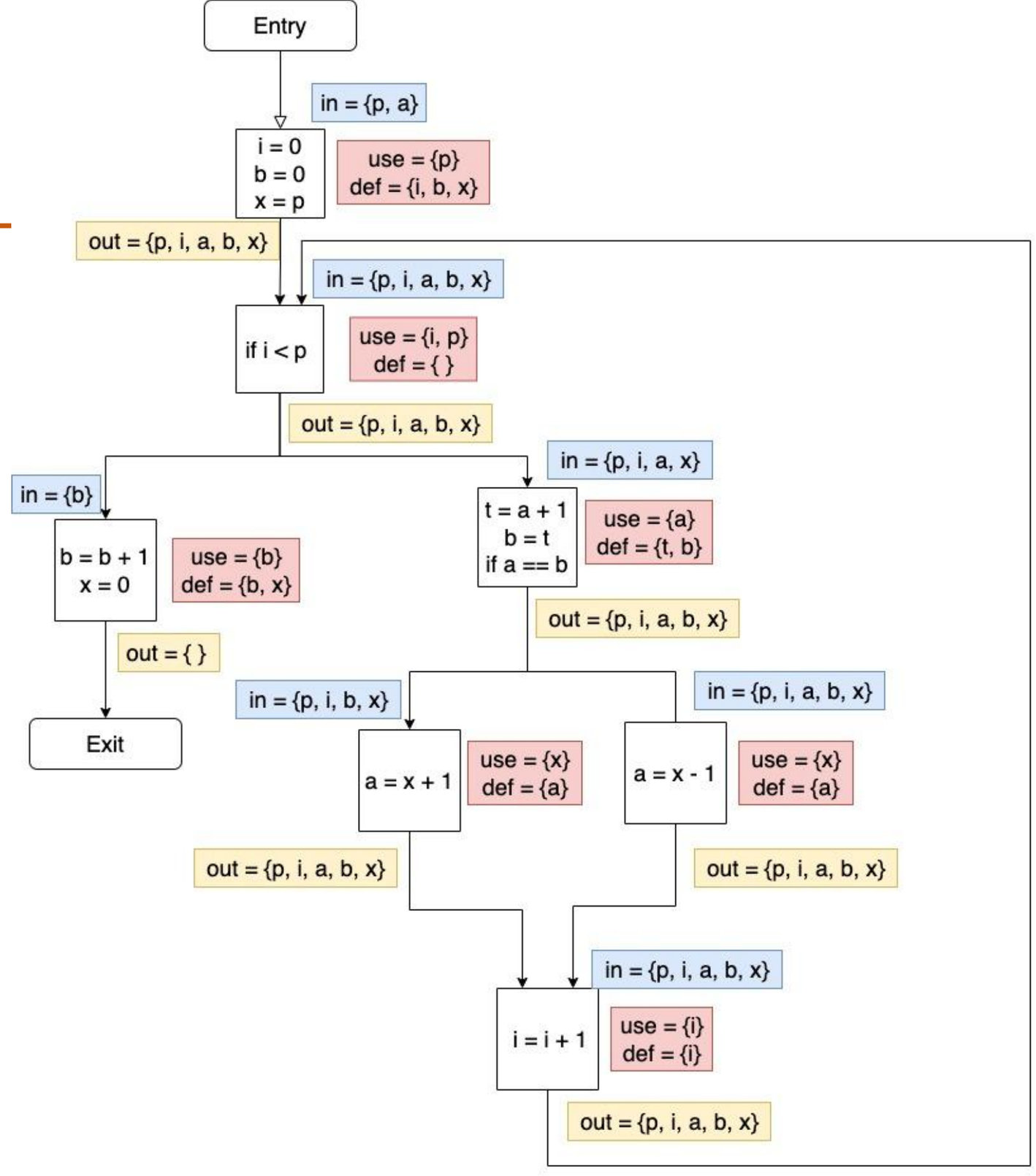
Example 3

Compute Liveness Information for the following CFG -



Compiler Design

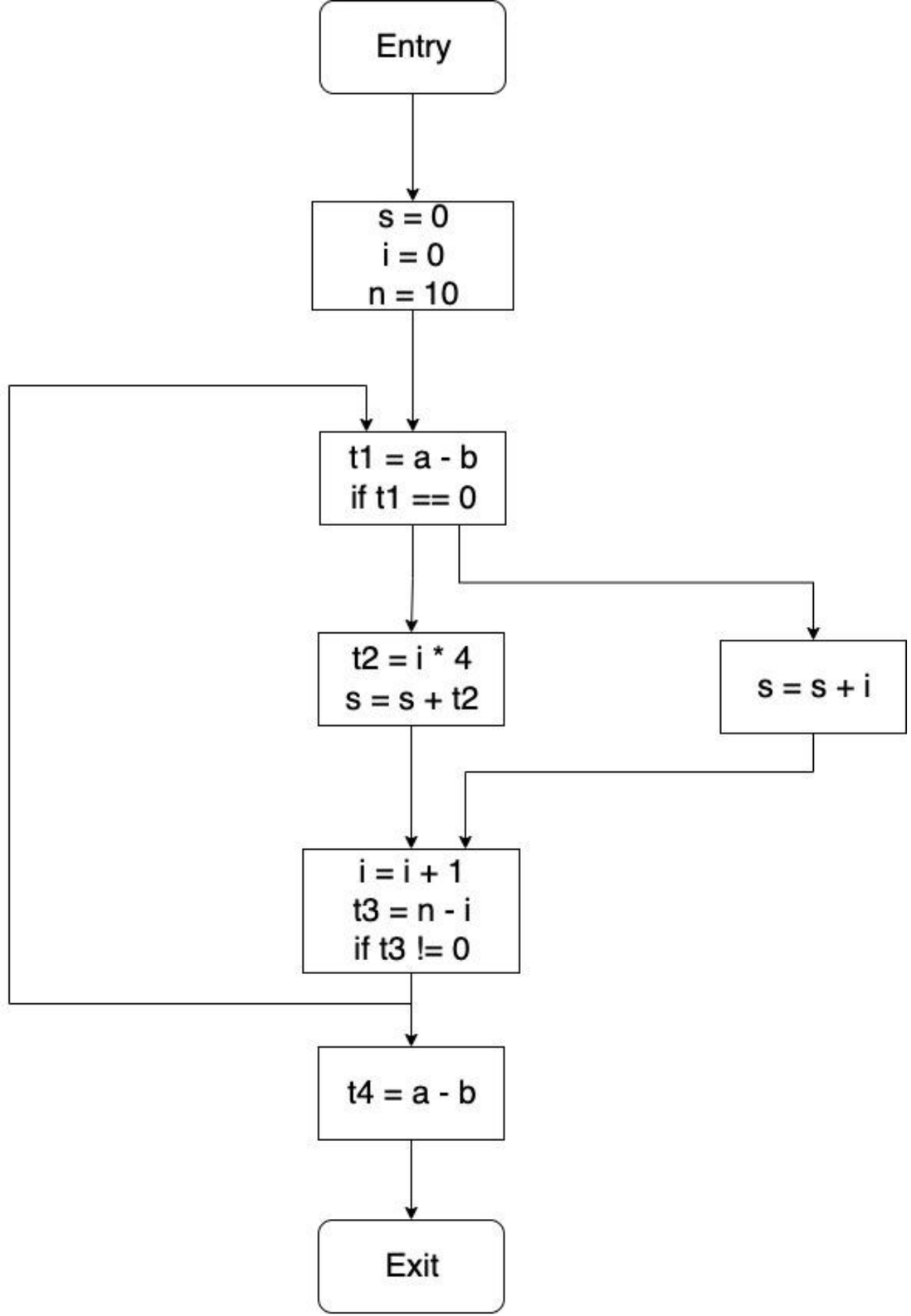
Example 3 - Solution



Compiler Design

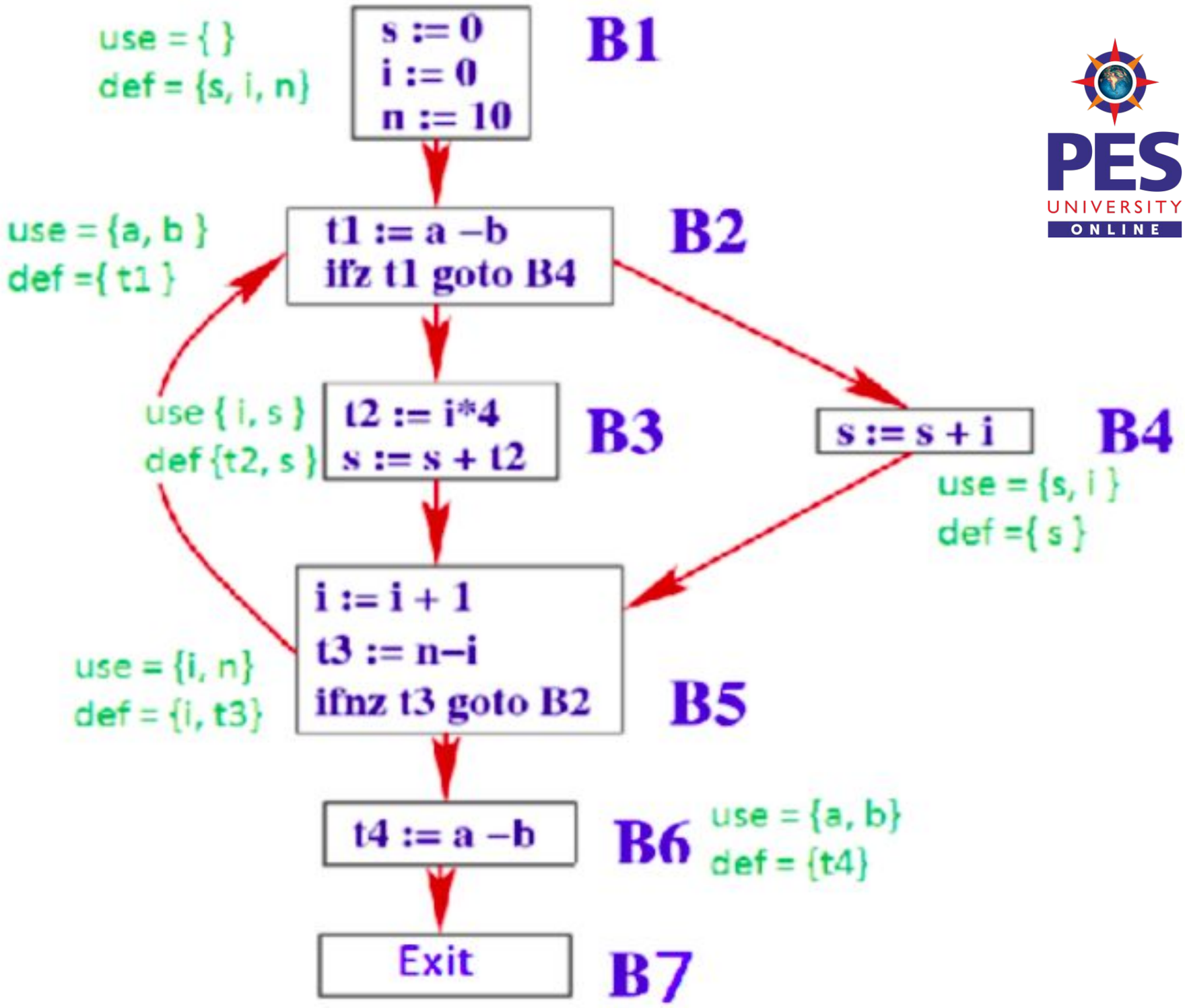
Example 4

Compute Liveness Information for the following CFG -



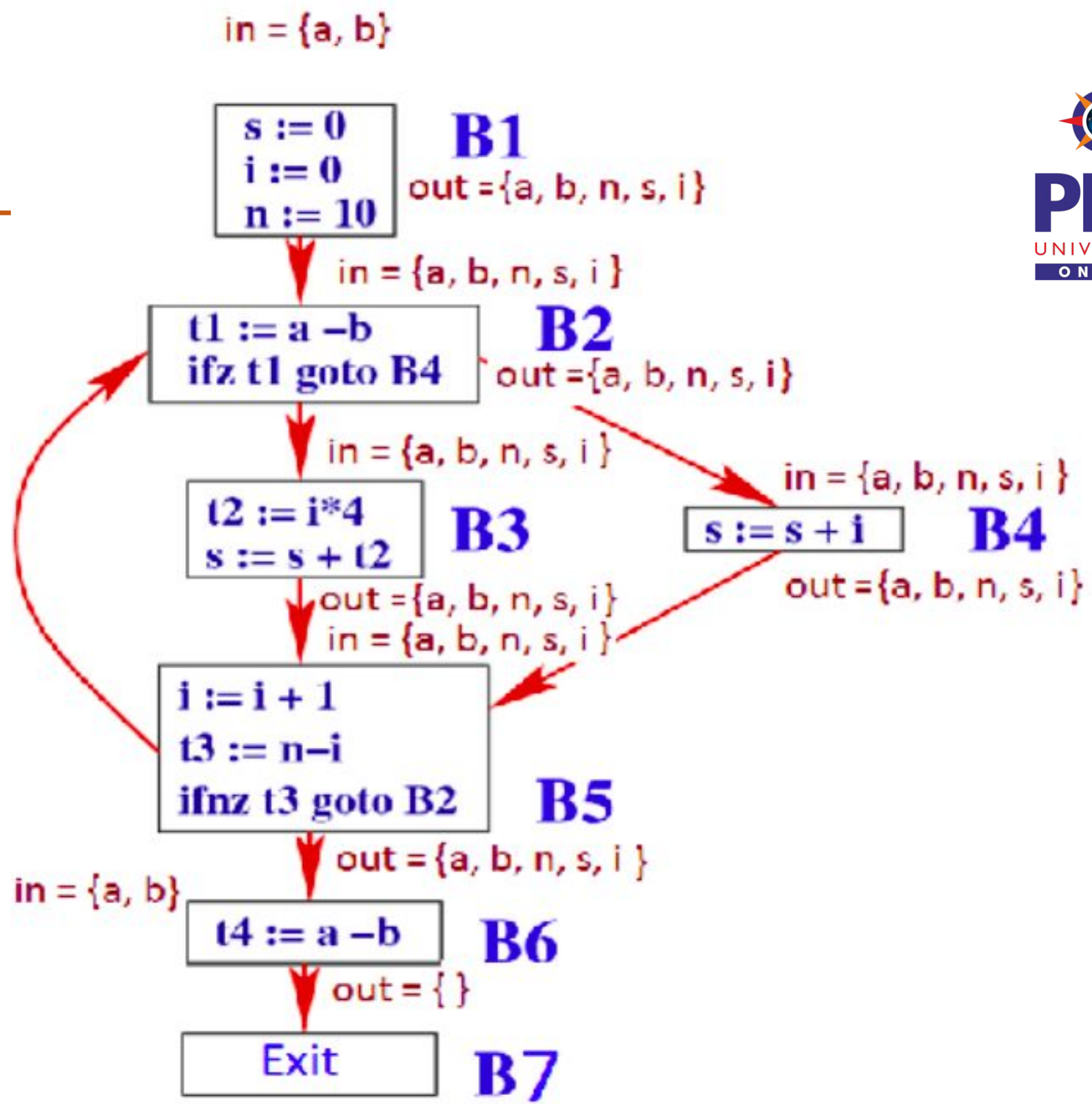
Compiler Design

Example 4 - Solution



Compiler Design

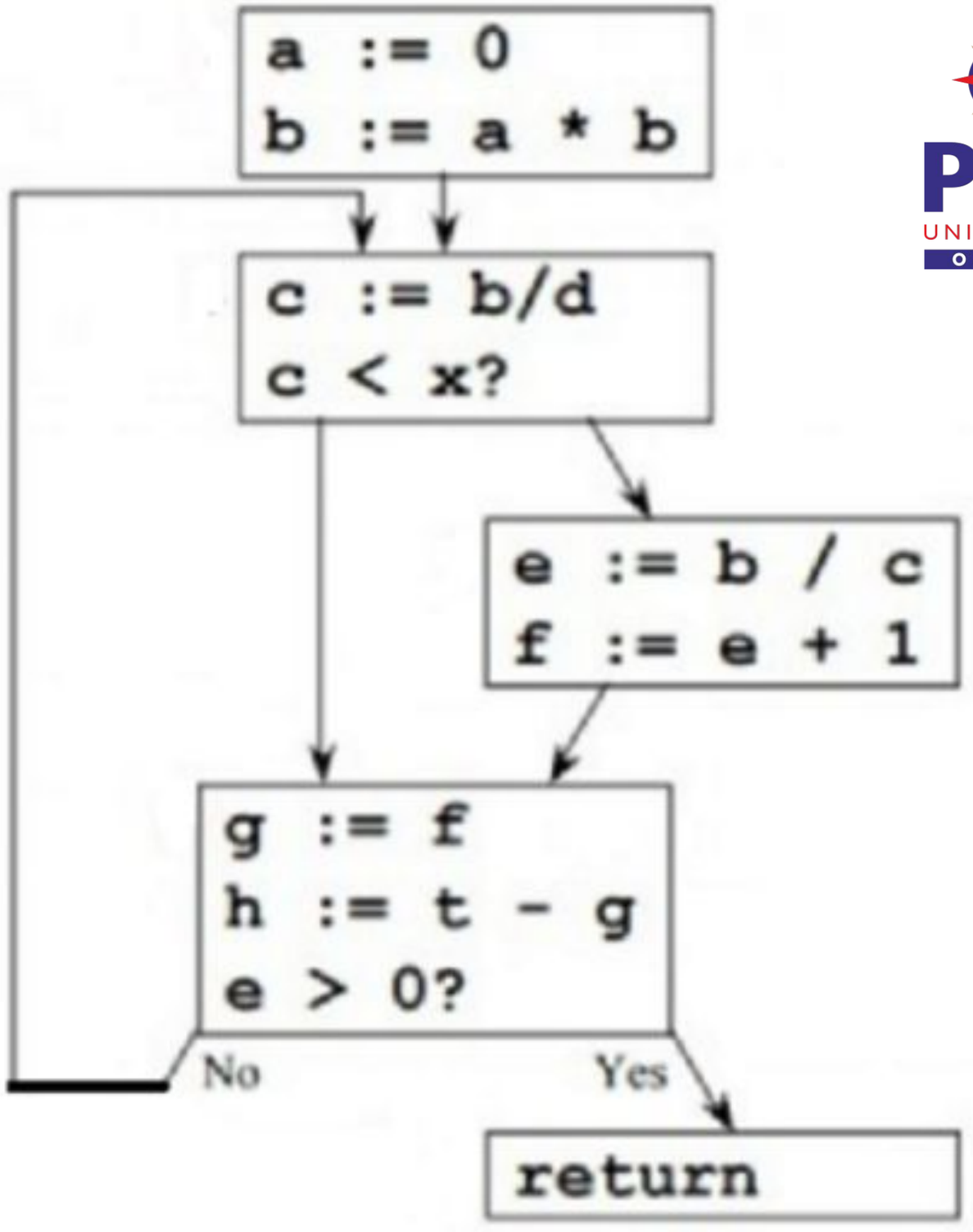
Example 4 - Solution



Compiler Design

Example 5

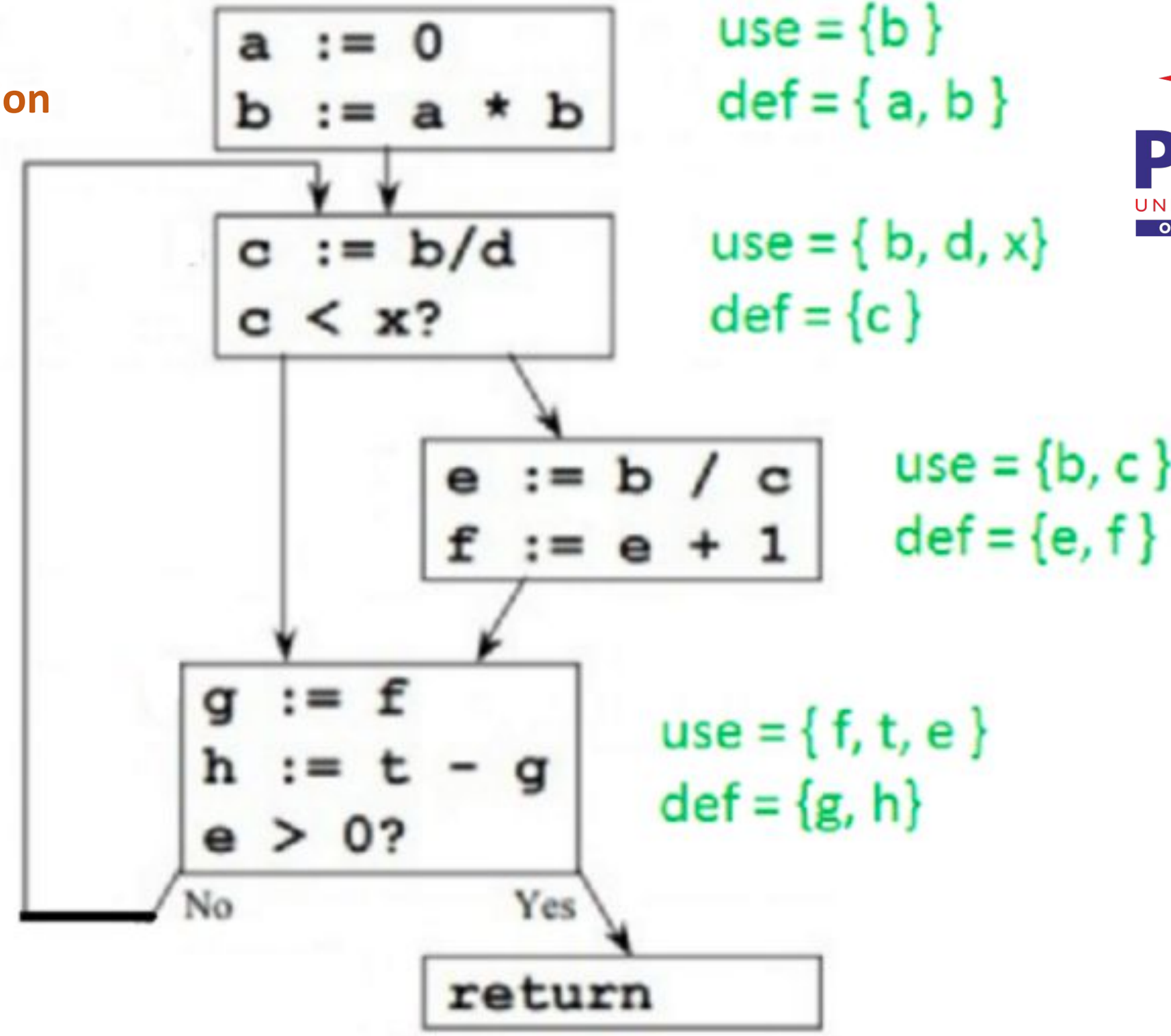
Compute Liveness Information for the following CFG -



Compiler Design

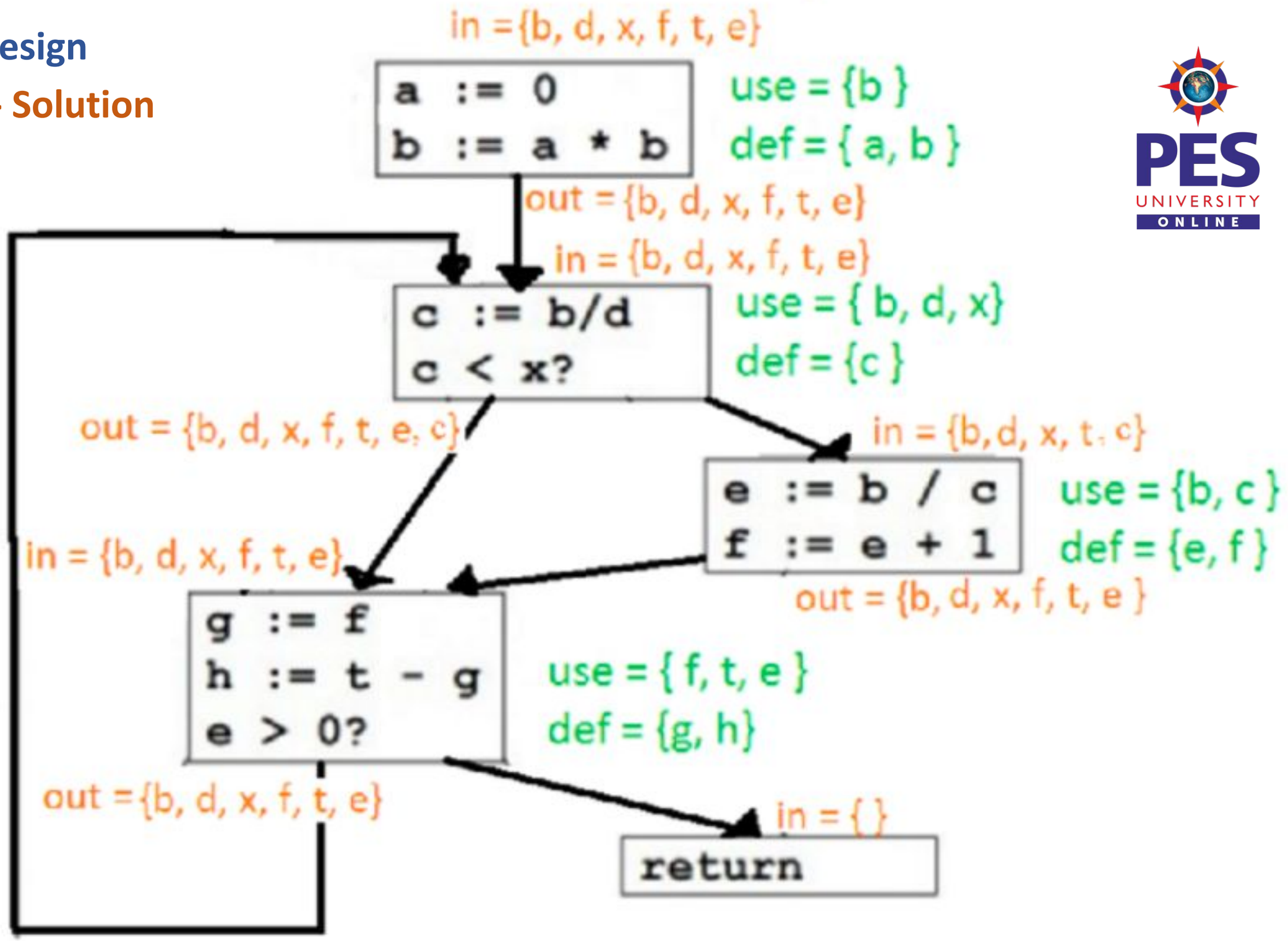
Example 2 - Solution

Step 2 - Compute use and def for each basic block.



Compiler Design

Example 2 - Solution



THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu

COMPILER DESIGN

Unit 5: Run-Time Environments

Prakash C O

Department of Computer Science and Engineering

COMPILER DESIGN

Unit 5: Run-Time Environments

Introduction

Prakash C O
Department of Computer Science and Engineering

Introduction

- A compiler must accurately implement the abstraction embodied in the source language definition. These abstractions typically include the concepts such as
  1. Names, Scopes, Bindings,
  2. Data types,
  3. Operators,
  4. Procedures, Parameters, and
  5. Flow-of-Control constructs.
- The compiler must cooperate with the Operating System and other System Softwares to support these abstractions on the target machine.
- To do so, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed.

Introduction

- A runtime environment is a set of data structures maintained at runtime to implement high-level structures such as
  - Functions,
  - Objects,
  - Exceptions,
  - Data types,
  - Flow-of-Control constructs, etc., .
- Run time environment - Everything you need to execute a program.

Introduction

- **Run-time environment deals with a variety of issues such as**

1. **the allocation of storage locations for the objects named in the source program,**
2. **the mechanisms used by the target program to access variables,**
3. **the linkages between procedures,**
4. **the mechanisms for passing parameters,** and
5. **the interfaces to the OS, i/o devices, and other programs.**

Introduction

- Many questions to answer regarding execution of functions:
  - What does the dynamic execution of functions look like?
  - Where is the executable code for functions located?
  - How are parameters passed in and out of functions?
  - Where are local variables stored?
  - …

COMPILER DESIGN

Unit 5: Run-Time Environments

Storage Organization

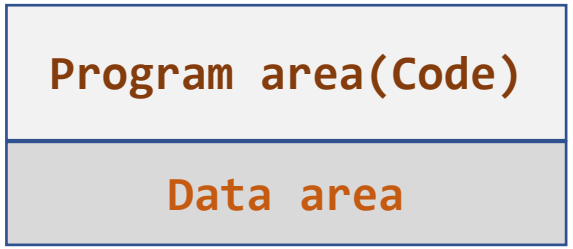
Prakash C O
Department of Computer Science and Engineering

Storage Organization

- ❑ The compiler deals with logical address space.
- ❑ OS maps the logical addresses to physical addresses

➢ The representation of an object program (Target Code) in the logical address space consists of

1. **Program area** and
2. **Data area.**



Note:

Logical Address Space is set of all logical addresses generated by CPU in reference to a program.

Physical Address (location in a memory unit) is set of all physical addresses mapped to the corresponding logical addresses.

Storage Organization

- **Runtime storage can be subdivided to hold** :
  1. **Target code -** it is static as its size can be determined at compile time
  2. **Static data objects**
  3. **Dynamic data objects – Heap Memory**
  4. **Automatic data objects – Stack memory**

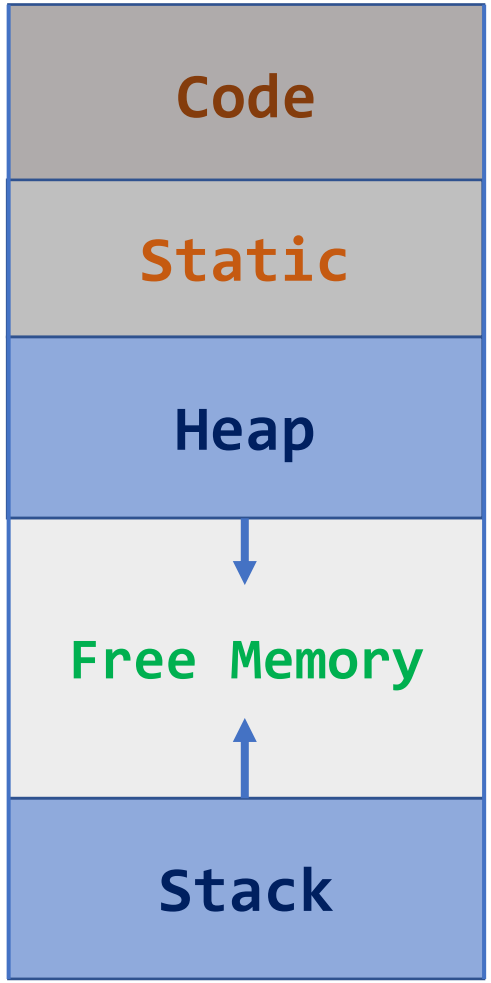


Figure 7.1: Subdivision of run-time memory into code and data areas

Storage Organization

1. Code (Target Machine Code):

- **The size of the generated target code is fixed at compile time,**

  So the compiler can place the executable target code in a statically determined area called *Code,* usually in the low end of memory.

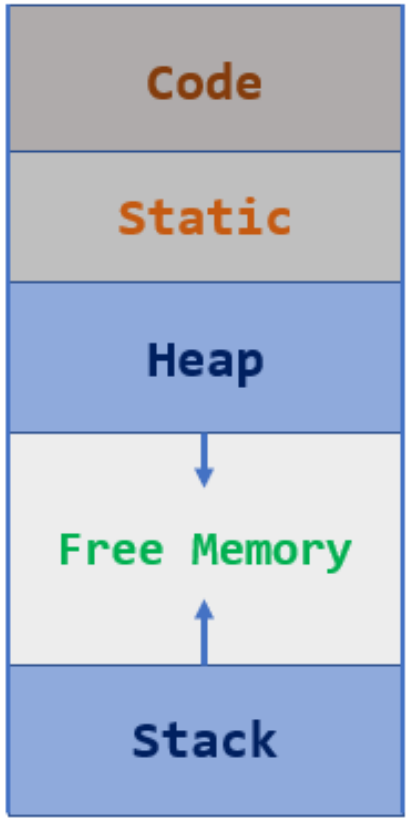


Figure 7.1: Subdivision of run-time memory into code and data areas

Storage Organization

2. Static data objects:

- The size of data objects, such as Global Constants are known at compile time, and can be placed in another statically determined area called *Static*.
- Static allocation is possible only when the compiler knows the size of data object at compile time.
- Reason for statically allocating as many data objects as possible is that *the addresses of these objects can be compiled into the target code*.

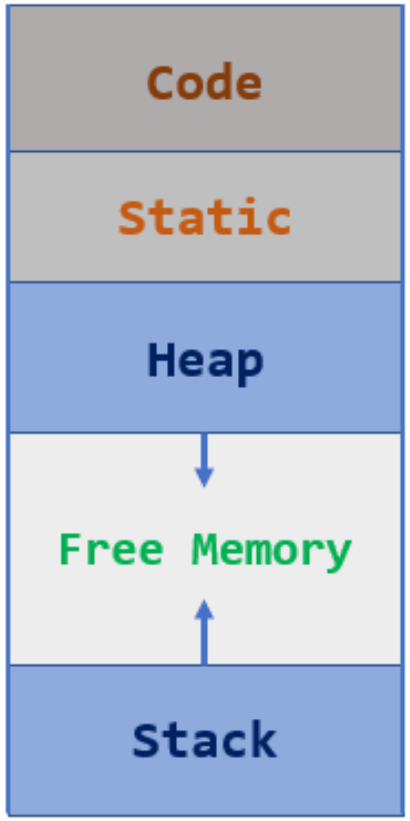


Figure 7.1: Subdivision of run-time memory into code and data areas

Storage Organization

3. Heap Memory:

- To maximize the utilization of space at run time, *Stack* and *Heap*, are at the opposite ends of the remainder of the address space.
- Stack and Heap areas are dynamic; their size can change as the program executes.
- Heap memory is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

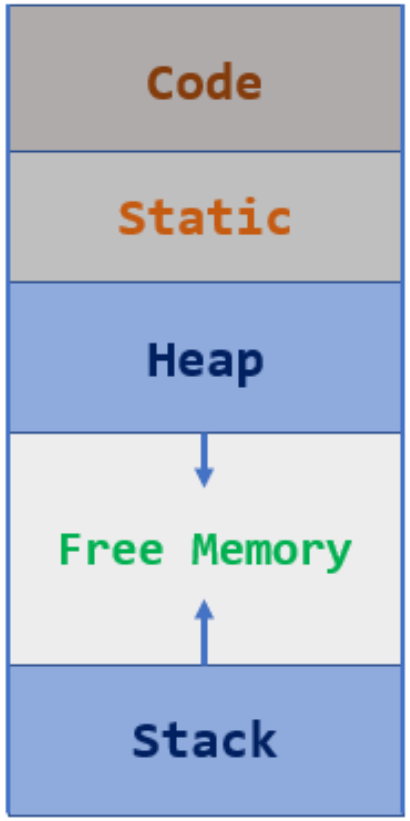


Figure 7.1: Subdivision of run-time memory into code and data areas

Storage Organization

4. Stack Memory:

- **The *Stack* is used to store data structures called activation records that get generated during procedure calls.**
- Stack grows with each call and shrinks with each procedure return/terminate.

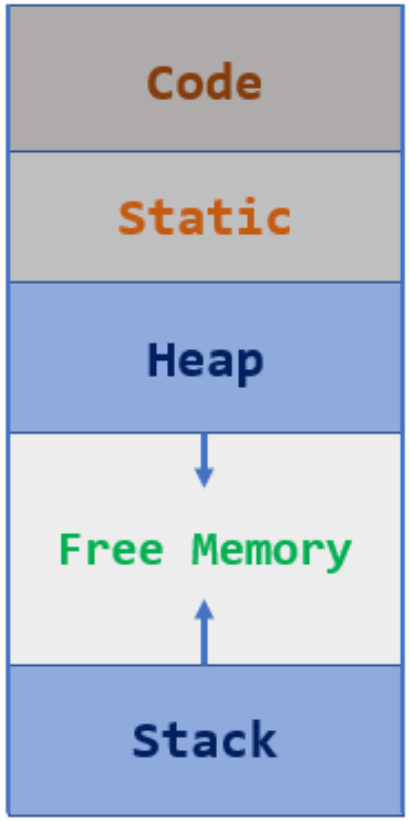


Figure 7.1: Subdivision of run-time memory into code and data areas

Static Versus Dynamic Storage Allocation

➢Static allocation:

A storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.

Static allocation

- Names are bound to storage locations at compilation time
  - Bindings do not change, so no run time support is required
  - Names are bound to the same location on every invocation
  - Values are retained across activations of a procedure
- Limitations
  - Size of all data objects must be known at compile time
  - Data structures cannot be created dynamically
  - Recursive procedures are not allowed

Static Versus Dynamic Storage Allocation

➢ **Dynamic allocation –Storage allocation decisions are made when the program is running.**

- **Stack storage allocation**
- **Heap storage allocation**

Storage Organization

Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. **Stack storage**.
   - **When a function is called the local variables are stored in a stack, and it is automatically destroyed once returned**.
   - A stack is used when a variable is not used outside that function.
   - Stack automatically cleans up the object.
   - Not easily corrupted
   - Variables cannot be resized.

Storage Organization

Many compilers use some combination of the following two strategies for dynamic storage allocation:

2. Heap storage.

- **The heap allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.**
- **To support heap management, "garbage collection" enables the run-time system to detect useless data elements and reuse their storage, even if the programmer does not return their space explicitly.**
- **Automatic garbage collection** is an essential feature of many modern languages

Storage Organization

Stack Vs Heap Memory

- Stack is a linear data structure whereas Heap is a hierarchical data structure.
- Stack memory will never become fragmented whereas Heap memory can become fragmented as blocks of memory are first allocated and then freed.
- Stack accesses local variables only while Heap allows you to access variables globally.
- Stack variables can't be resized whereas Heap variables can be resized.
- Stack memory is allocated in a contiguous block whereas Heap memory is allocated in any random order.
- Stack doesn't require to de-allocate variables whereas in Heap de-allocation is needed.
- Stack allocation and deallocation are done by compiler instructions whereas Heap allocation and deallocation is done by the programmer.

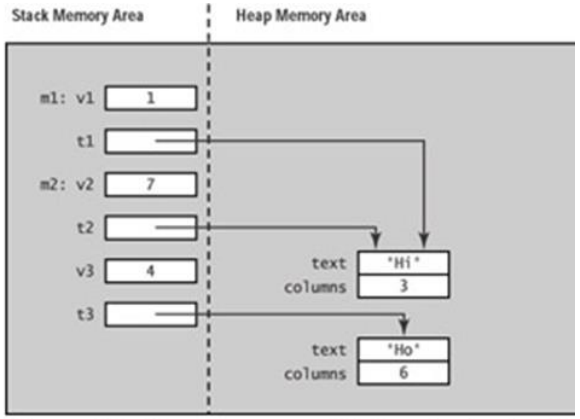


Figure 5.4: The memory model after checkpoint #4

Storage Organization

Stack and Heap Storage Allocation: Example

```
void m1() {
 int v1 = 1;
 TextField t1;          // checkpoint #1
 t1 = new TextField("Hi", 3); // checkpoint #2
 m2(v1, t1);
 v1 = 8;              // checkpoint #5
}
void m2(int v2, TextField t2) {
 int v3 = 4;
 TextField t3;          // checkpoint #3
 t3 = new TextField("Ho", 6);
 v2 = 7;              // checkpoint #4
}
```

Stack

Heap

Storage Organization

Stack and Heap Storage Allocation: Example

```
void m1() {
 int v1 = 1;
 TextField t1;          // checkpoint #1
 t1 = new TextField("Hi", 3); // checkpoint #2
 m2(v1, t1);
 v1 = 8;                // checkpoint #5
}
void m2(int v2, TextField t2) {
 int v3 = 4;
 TextField t3;          // checkpoint #3
 t3 = new TextField("Ho", 6);
 v2 = 7;                // checkpoint #4
}
```

Function m1 partial Activation Record.

Function m2 partial Activation Record.

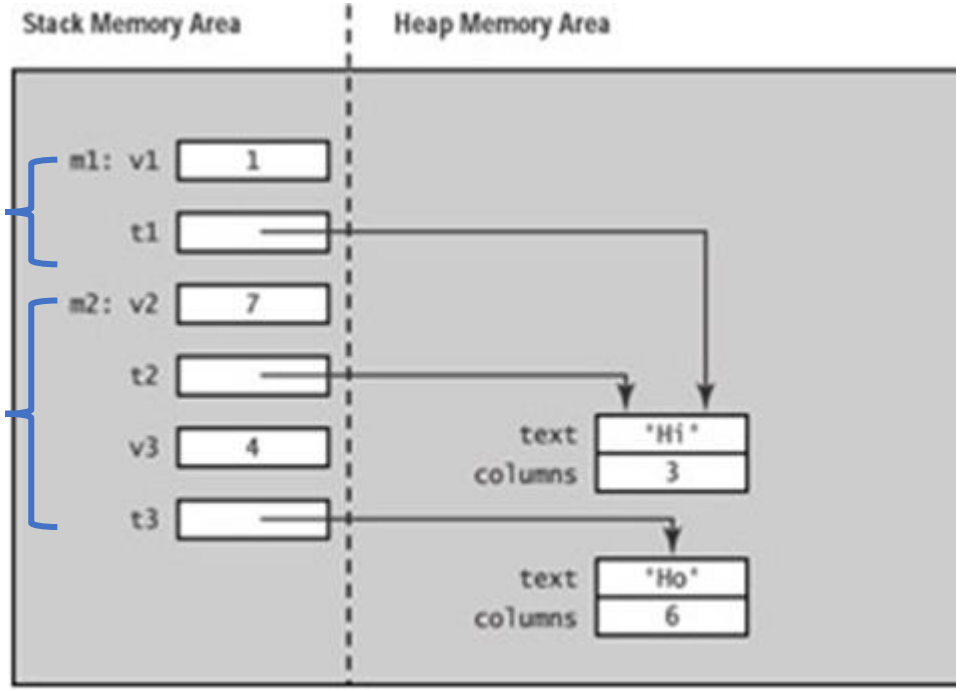


Figure 5.4: The memory model after checkpoint #4

COMPILER DESIGN

Unit 5: Run-Time Environments

Stack Allocation of Space:

Activation Tree & Activation Record

Prakash C O
Department of Computer Science and Engineering

Activation Trees

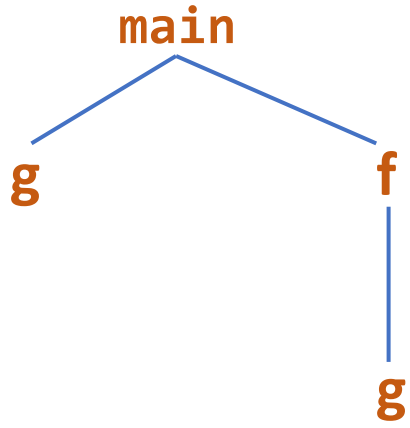
➢ **Activation:**

- Each execution of procedure is referred to as an *activation*.

➢ **Activation tree:**

- A tree structure representing all of the function calls made by a program on a particular execution.
- Activation tree shows the way control enters and leaves activations.
- Activation tree root represents the activation of main.

```
void g() { return 42; }
void f() { return g(); }

main() {
 g();
 f():
}
```



Activation Trees

➢ **Activation tree:**

- **A representation of the activations of procedures during the running of an entire program by a tree.**
- **Activation tree depends on the runtime behavior of a program; can't always be determined at compile-time.**
- The static equivalent is the call graph.

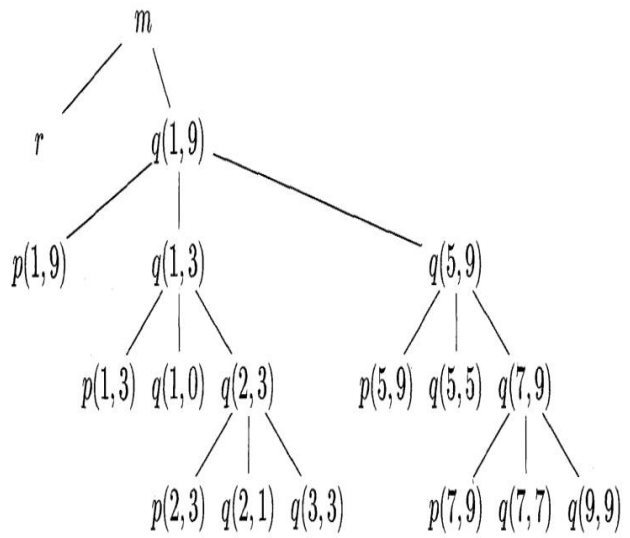


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

Activation Trees

Properties of activation trees are :-

1. Each node represents an activation of a procedure.
2. The root shows the activation of the main function.
3. Node a is the parent of b if control flows from a to b
4. Node a is to the left of b if lifetime of a occurs before b

Note:

- We show these activations in the order that they are called, from left to right.
- One child must finish before the activation to its right can begin.

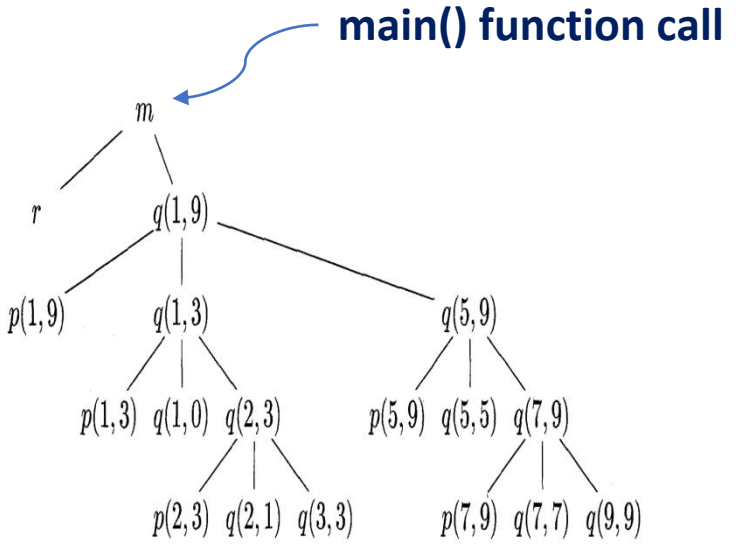


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Activation Trees

Example:

Figure 7.2 contains a quicksort program that reads 9-integers into an *array* a and sorts them using the recursive quicksort algorithm.

The main function has three tasks.

1. **Calls *readArray*,**
2. **Sets the sentinels**, and then
3. **Calls quicksort on the entire data array.**

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

Activation Trees

- Figure 7.3 suggests a sequence of calls that might result from an execution of the program.

  In this execution, the call to partition(1,9) returns 4, so a[1] through a[3] hold elements less than its chosen separator value v, while the larger elements are in a[5] through a[9].

```
enter main()
  enter readArray()
  leave readArray()
  enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
    ...
    leave quicksort(1,3)
    enter quicksort(5,9)
    ...
    leave quicksort(5,9)
  leave quicksort(1,9)
leave main()
```

Fig 7.3: Possible activations of Fig 7.2

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

Activation Trees

- ***Each execution of procedure*** **is referred to as an *activation* of the procedure.**
- ***Lifetime of an activation*** **is the sequence of steps present in the execution of the procedure.**
- **If 'a' and 'b' be two procedures, then their activations will be**
  - **non-overlapping (when one is called after other) or**
  - **nested (nested procedures).**
- **A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended.**

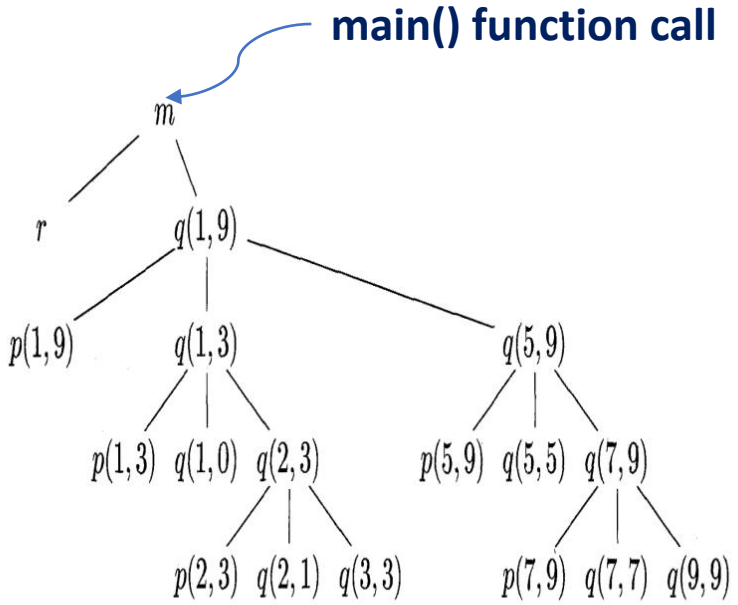


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Activation Trees

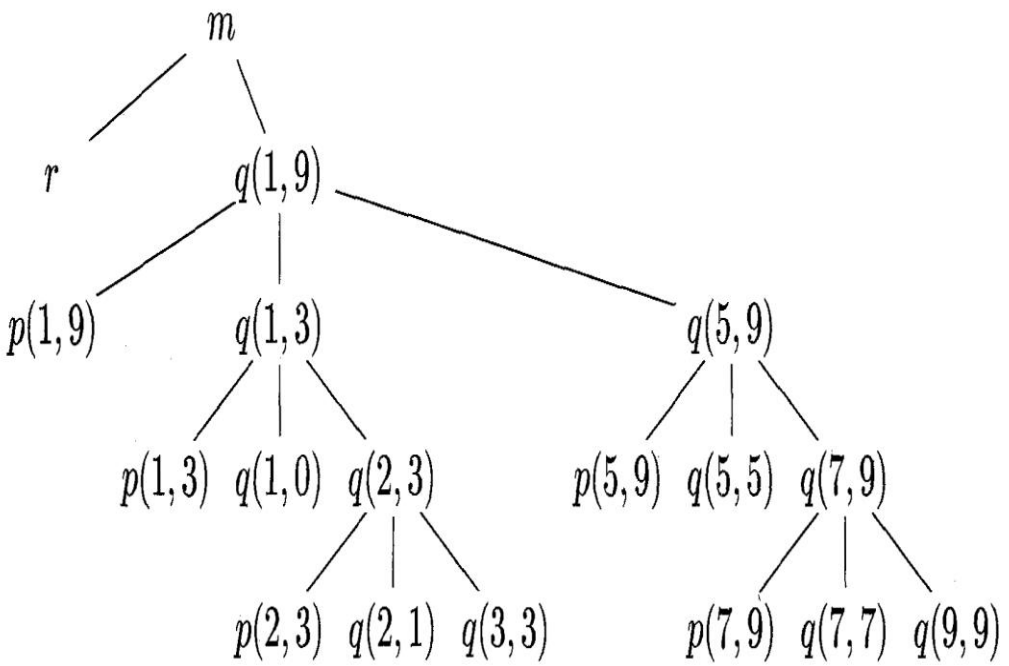


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

Activation Trees

- **Example:** One possible activation tree that completes the sequence of calls and returns suggested in Fig. 7.3 is shown in Fig. 7.4.

Functions are represented by the first letters of their names.

Remember that this tree is only one possibility, since the arguments of subsequent calls, and also the number of calls along any branch is influenced by the values returned by ***partition***.

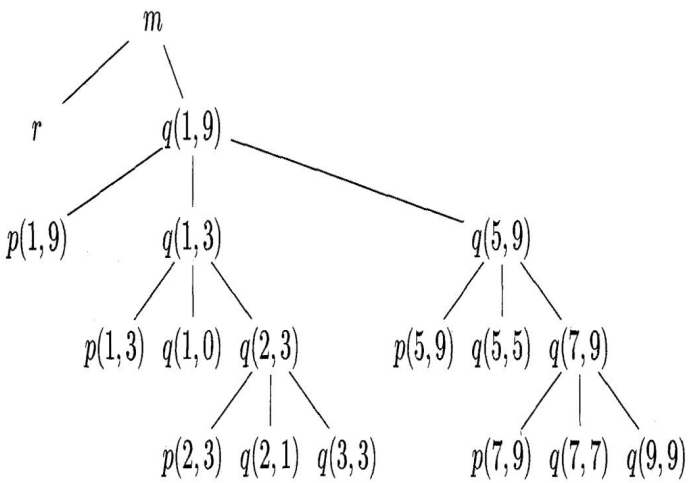


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

Activation Trees

- The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:
  1. The sequence of procedure calls corresponds to a _____ traversal of the activation tree.
  2. The sequence of returns corresponds to a _____ traversal of the activation tree.
  3. Flow of the control in a program corresponds to a _____ traversal of the activation tree.

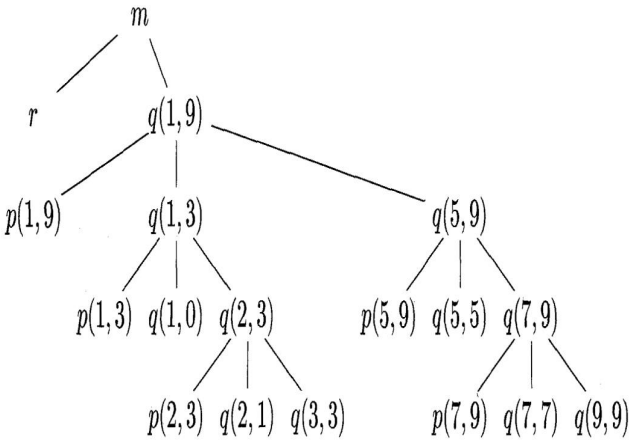


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Activation Trees

- **The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:**

  1. **The sequence of procedure calls corresponds to a preorder traversal of the activation tree.**
  2. **The sequence of returns corresponds to a postorder traversal of the activation tree.**
  3. **Flow of the control in a program corresponds to a depth-first traversal of the activation tree.**

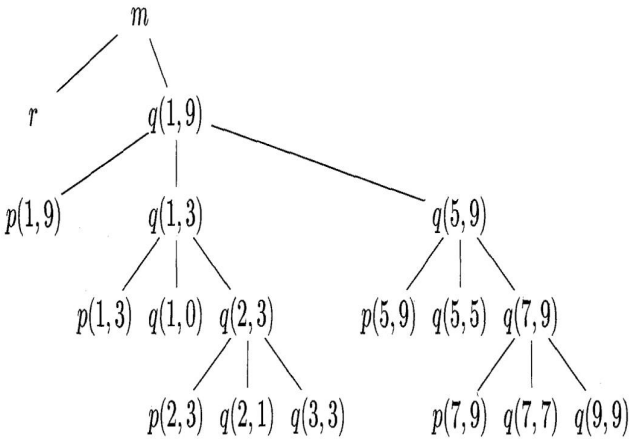


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Activation Trees

- **The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:** cont..

  4. If control lies within a particular activation of some procedure, corresponding to a node N of the activation tree.

  Then the *activations that are currently open(live)* are those that correspond to node N and its ancestors.

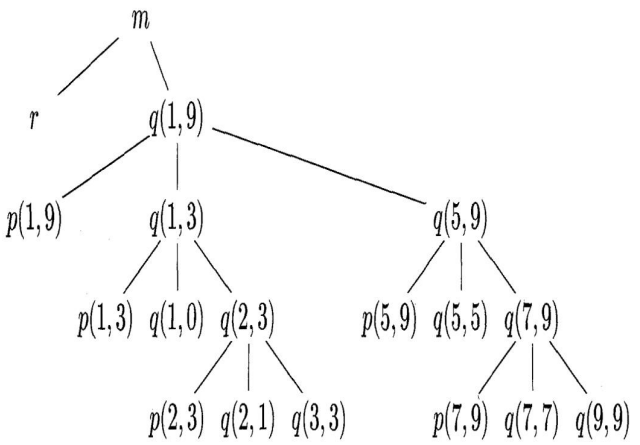


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Activation Trees

- **If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end.**

There are three common cases:

1. **The activation of q terminates normally**. Then in essentially any language, control resumes just after the point of p at which the call to q was made.
2. **The activation of q, or some procedure q called, either directly or indirectly, aborts**; i.e., it becomes impossible for execution to continue. In that case, p ends simultaneously with q.

Activation Trees

- **If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end.**

There are three common cases: cont…

3. **The activation of q terminates because of an exception that q cannot handle.**

   Procedure p may handle the exception, in which case the activation of q has terminated while the activation of p continues, although not necessarily from the point at which the call to q was made.

   If p cannot handle the exception, then this activation of p terminates at the same time as the activation of q, and presumably the exception will be handled by some other open activation of a procedure.

Activation Trees

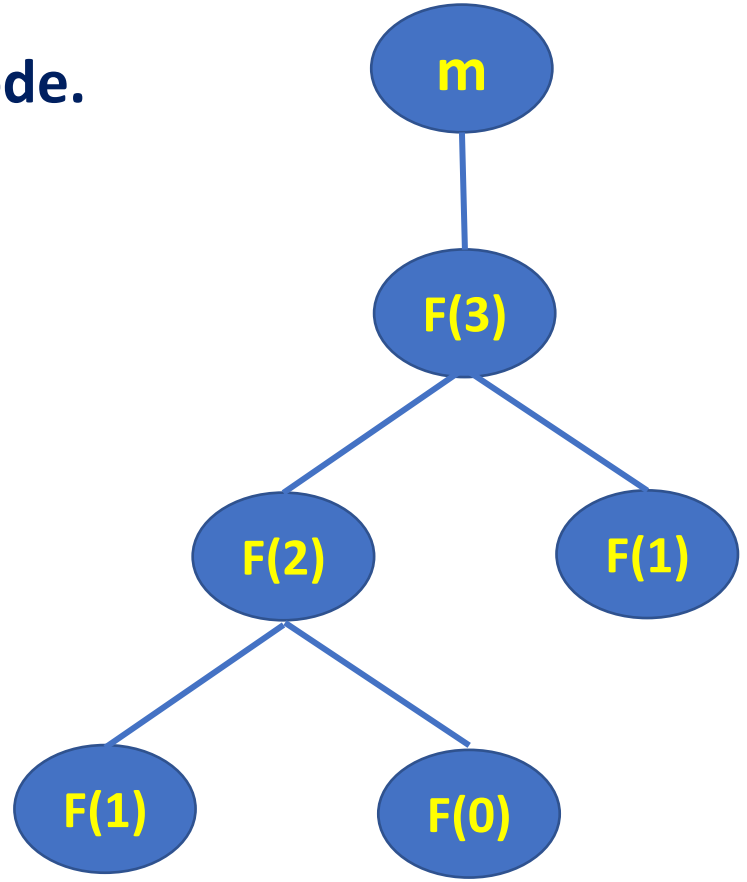
➢ **Exercise 1: Write Activation tree for the below code.**

```
int main() {
  Fib(3);
}

int Fib(int n) {
  if (n<=1) return n;
  return Fib(n-1) + Fib(n-2);
}
```

Activation Trees

➢ **Exercise 1: Write Activation tree for the below code.**

```
int main() {
  Fib(3);
}

int Fib(int n) {
  if (n<=1) return n;
  return Fib(n-1) + Fib(n-2);
}
```



Activation Trees

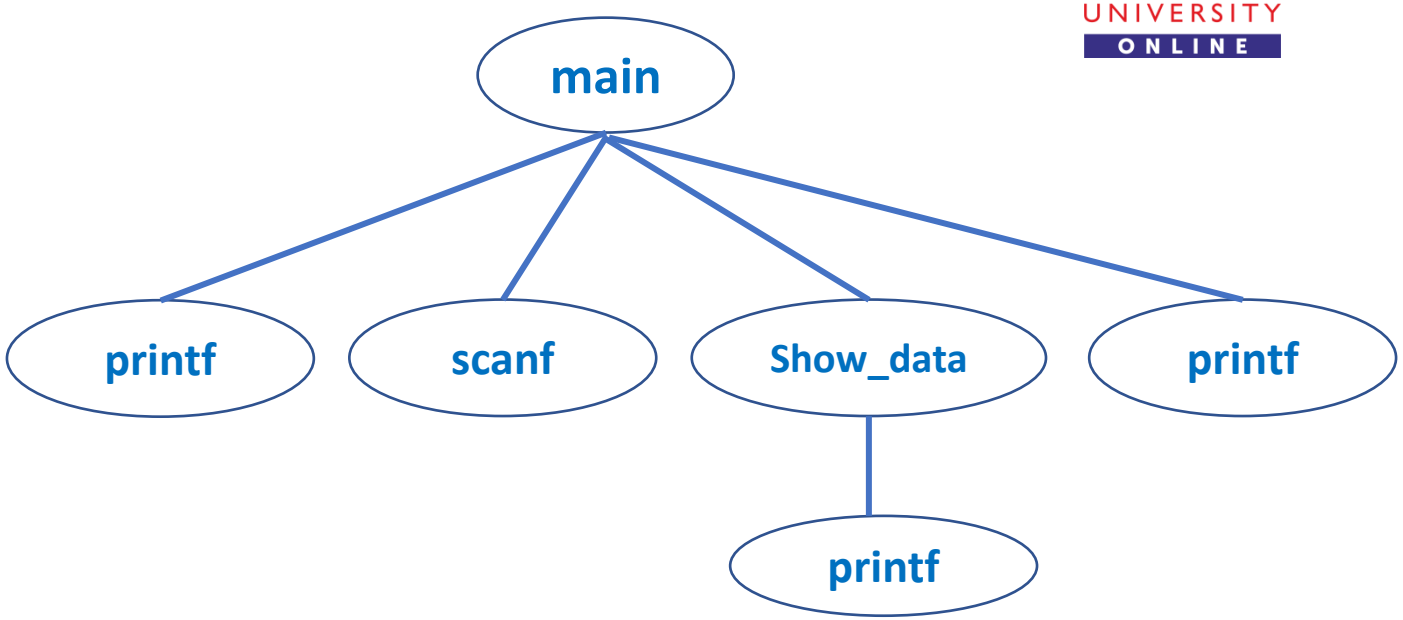
➢ **Exercise 2: Write Activation tree for the below code.**

```
main()
{
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue…");
. . .
}

int show_data(char *user)
{ printf("Your name is %s", user);
  return 0;
}
```

Activation Trees

➢ **Exercise 2: Write Activation tree for the below code.**

```
main()
{
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue…");
. . .
}

int show_data(char *user)
{ printf("Your name is %s", user);
  return 0;
}
```



When a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller.
This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.

Activation Records (or Stack frames)

- An activation record is a block of storage that contains all the necessary information required by a single execution of a procedure.
- An activation record is used to store Status of the machine (the value of PC and machine registers), Actual Parameters, Return values, Access and Control links, and Local variables, when a procedure call occurs.
- Function calls are implemented using a stack of activation records.
  - Calling a function pushes a new activation record onto the stack.
  - Returning from a function pops the current activation record from the stack.

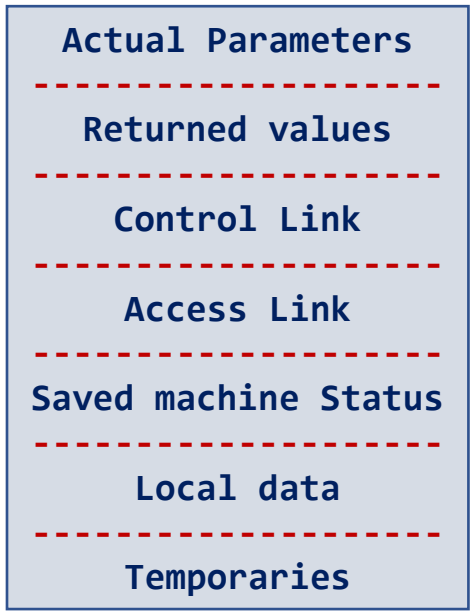


Figure 7.5: A general activation record

Activation Records (or Stack frames)

- When control returns from the *called procedure*, the activation of the *calling procedure* can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call.
- "Once a function returns, its activation record cannot be referenced again."

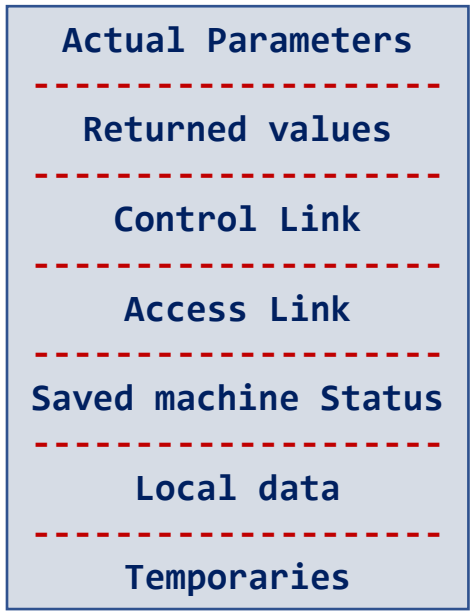


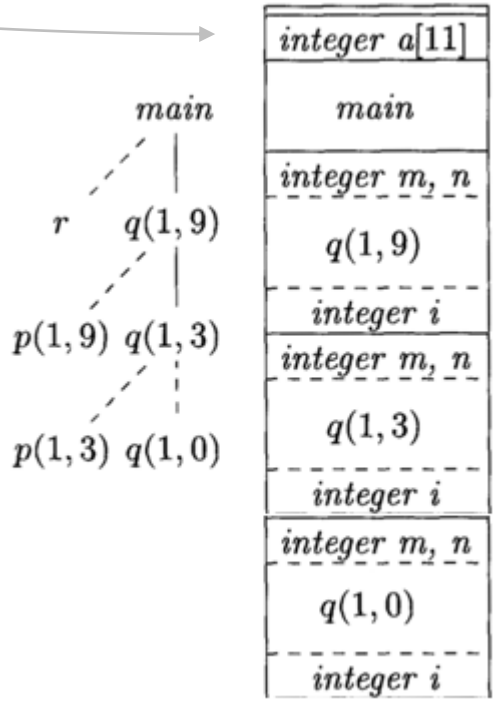
Figure 7.5: A general activation record

Activation Records

- **Procedure calls and returns are usually managed by a run-time stack called the control stack.**

- **Each live activation has an _____ on the control stack**, with the root of the activation tree at the bottom.

- **The entire sequence of activation records on the stack corresponds to the ____ in the activation tree to the activation where control currently resides**.
  The latter activation has its record at the top of the stack.




```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

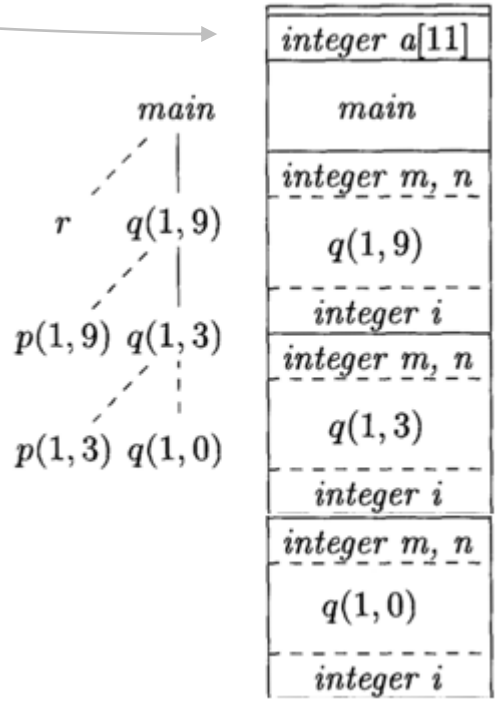
Figure 7.2: Sketch of a quicksort program

Activation Records

- **Procedure calls and returns are usually managed by a run-time stack called the control stack.**

- **Each live activation has an activation record on the control stack**, with the root of the activation tree at the bottom.

- **The entire sequence of activation records on the stack corresponds to the path in the activation tree to the activation where control currently resides.**
  The latter activation has its record at the top of the stack.




```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

Activation Records

- **Example:**

  If control is currently in the activation q(2,3) of the tree of Fig. 7.4, then the activation record for q(2,3) is at the top of the control stack.

  Just below is the activation record for q(1,3), the parent of q(2,3) in the tree.

  Below that is the activation record q(1,9), and at the bottom is the activation record for m, the main function and root of the activation tree.

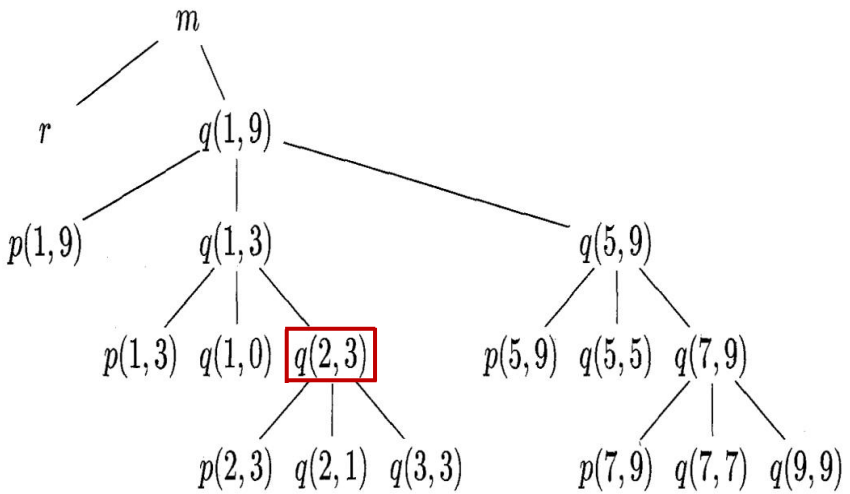
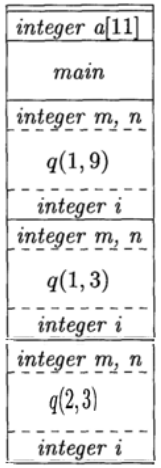


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Stack bottom



Stack top

Activation Records

- **The contents of activation records vary with the language being implemented.**
- **A list of the kinds of data that might appear in an activation record are:**
  1. **Temporary values**, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
  2. **Local data** belonging to the procedure whose activation record this is.

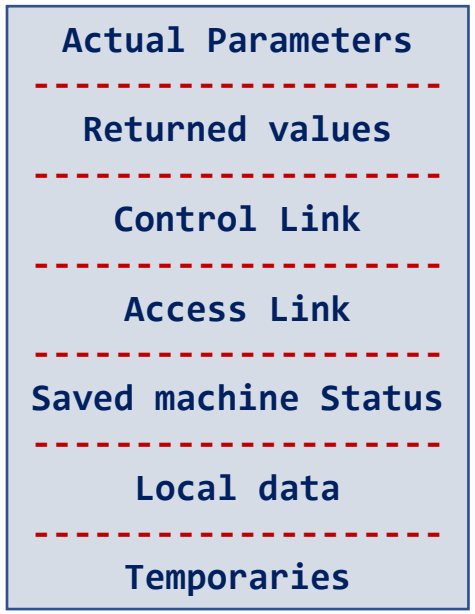


Figure 7.5: A general activation record

Activation Records

- **A list of the kinds of data that might appear in an activation record are:**

  3. **A *saved machine status*, with information about the state of the machine just before the call to the procedure.**

     This information typically includes

     - ***return address*** (value of the program counter) and
     - ***contents of registers*** that were used by the calling procedure and that must be restored when the return occurs.

  4. **An "*access link*" may be needed to locate data needed by the called procedure but found elsewhere**, e.g., in another activation record.

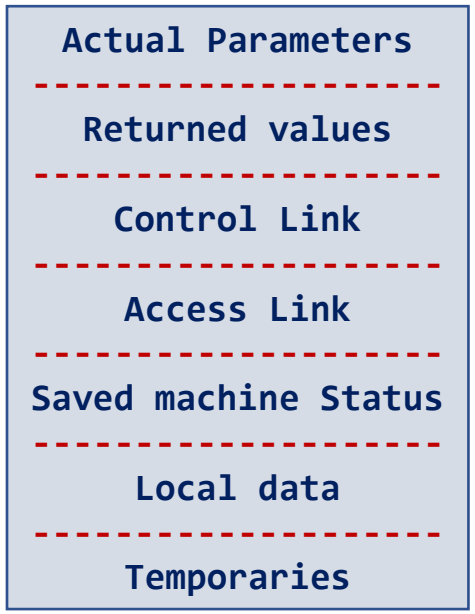


Figure 7.5: A general activation record

Activation Records

- **A list of the kinds of data that might appear in an activation record are:**

5. **A *control link*, pointing to the activation record of the caller.**
6. **Space for the *return value* of the called function, if any.**

   Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. **The *actual parameters* used by the calling procedure.**

   Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

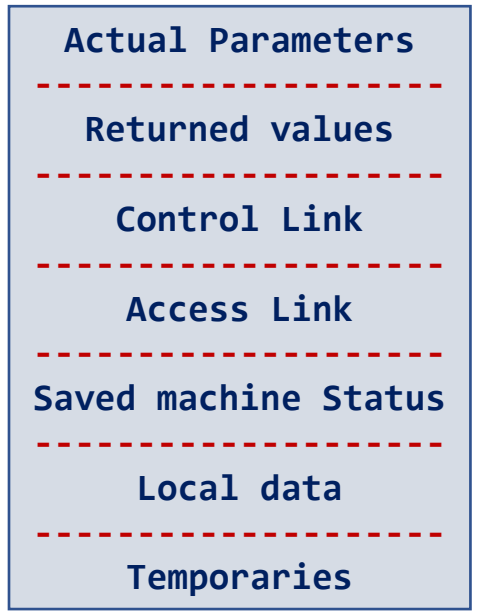
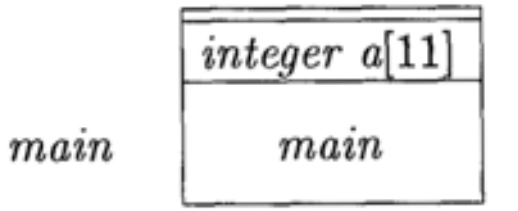


Figure 7.5: A general activation record

Activation Records

- **Example 7.4: Figure 7.6 shows snapshots of the run-time stack as control flows through the activation tree of Fig. 7.4.**

Since array *a* is global, space is allocated for it before execution begins with an activation of procedure main, as shown in Fig. 7.6(a).



(a) Frame for *main*

Figure 7.6: Downward-growing stack of activation records

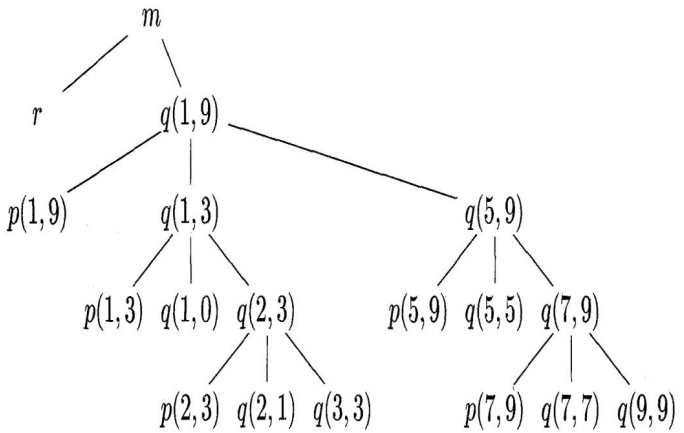


Figure 7.4: Activation tree representing calls during an execution of *quicksort*

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

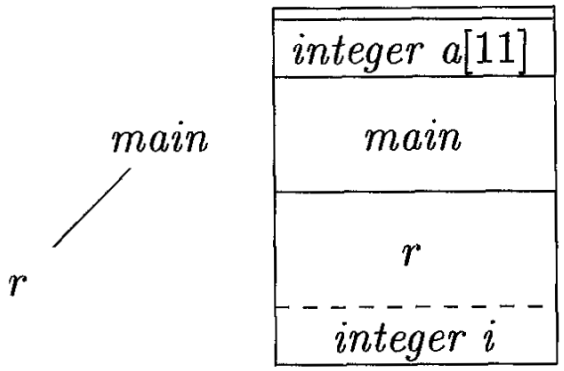
Figure 7.2: Sketch of a quicksort program

Activation Records

- **Example 7.4:** cont…

  When control reaches the first call in the body of main, procedure r is activated, and its activation record is pushed onto the stack (Fig. 7.6(b)).

  The activation record for r contains space for local variable i. When control returns from this activation, its record is popped, leaving just the record for main on the stack.



(b) r is activated

Figure 7.6: Downward-growing stack of activation records

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

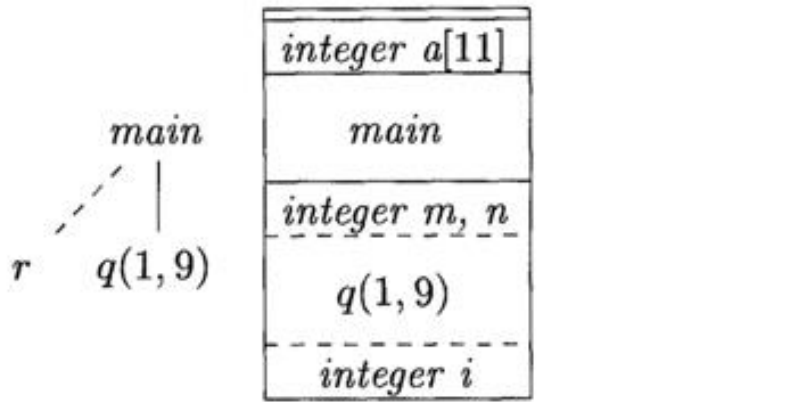
Figure 7.2: Sketch of a quicksort program

Activation Records

- **Example 7.4:** cont…

Control then reaches the call to q (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack, as in Fig. 7.6(c).

The activation record for q contains space for the parameters m and n and the local variable i, following the general layout in Fig. 7.5.



(c) r has been popped and $q(1,9)$ pushed

Figure 7.6: Downward-growing stack of activation records

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

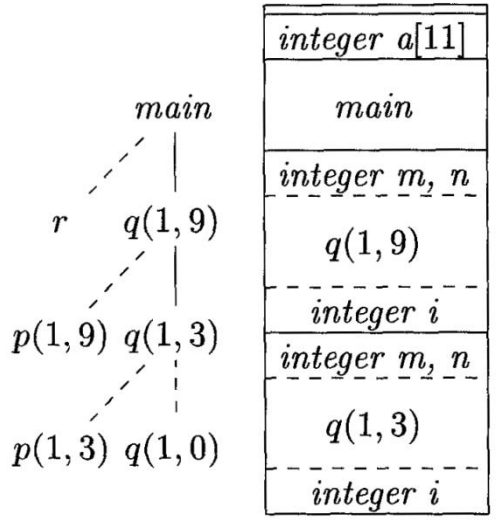
Figure 7.2: Sketch of a quicksort program

Activation Records

- **Example 7.4:** cont…

  Several activations occur between the last two snapshots in Fig. 7.6. A recursive call to q(1,3) was made. Activations p(1,3) and q(1,0) have begun and ended during the lifetime of q(1,3), leaving the activation record for q(1,3) on top (Fig. 7.6(d)).

  Notice that when a procedure is recursive, it is normal to have several of its activation records on the stack at the same time.



(d) Control returns to $q(1,3)$

Figure 7.6: Downward-growing stack of activation records

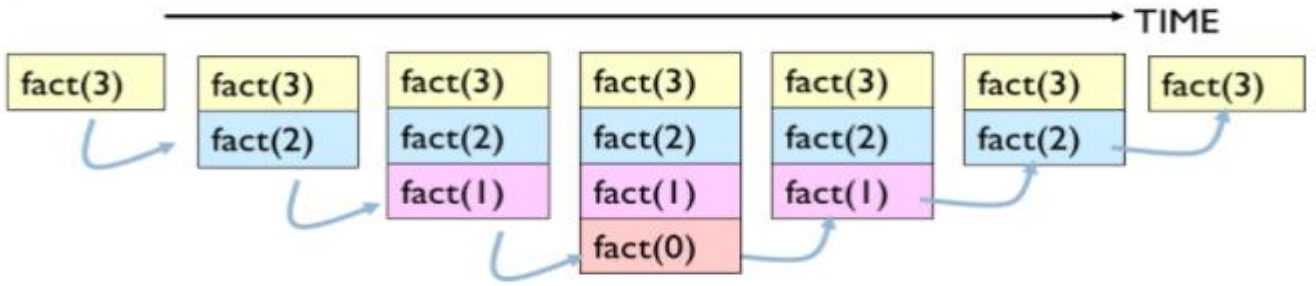
```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 7.2: Sketch of a quicksort program

Activation Records

- Notice that when a procedure is recursive, it is normal to have several of its activation records on the stack at the same time.
- Example:

```
int fact(int n) {
  if (n > 0) return n*fact(n - 1);
  else return 1;
}
```



A procedure call creates a new activation record. Caller's record is preserved because we'll need it when callee finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

COMPILER DESIGN

Unit 5: Run-Time Environments

Stack Allocation of Space: Calling Sequence and Return Sequence

Prakash C O
Department of Computer Science and Engineering

Calling Sequence and Return sequence

- Calling Sequence (is a code sequence)
  - Procedure calls are implemented by what are known as Calling Sequences.
  - Calling sequence is a code sequence that allocates an activation record on the stack and enters information into its fields.

- Return Sequence (is a code sequence)
  - A return sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.

Calling Sequences

- Calling sequences and the layout of activation records may differ greatly, even among implementations of the same language.
- **The code in a calling sequence is often divided between the calling procedure (the "caller") and the procedure it calls (the "callee").**
  - **Responsibility is shared between the caller and the callee**
- There is **no exact division of run-time tasks between caller and callee**; the source language, the target machine, and the operating system impose requirements that may favor one solution over another.

Calling Sequences

- **When designing calling sequences and the layout of activation records, the following principles are helpful:**

1. **Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.**
   - The motivation is that the caller can compute the values of the actual parameters of the call and place them on top of its own activation record, without having to create the entire activation record of the callee, or even to know the layout of that record.
   - **The callee knows where to place the return value, relative to its own activation record,** while however many arguments are present will appear sequentially below that place on the stack.

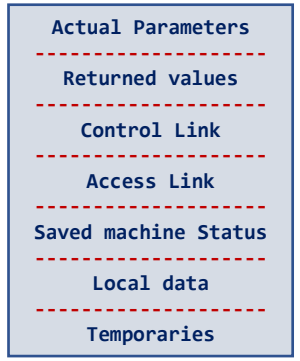


Figure 7.5: A general activation record

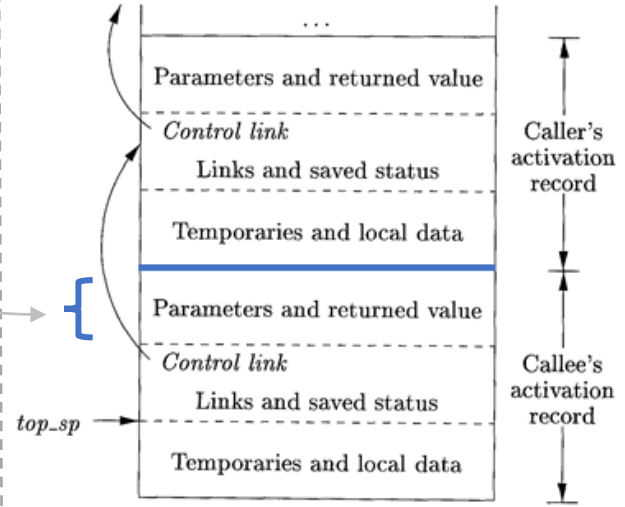


Figure 7.7: Division of tasks between caller and callee

Calling Sequences

2. **Fixed-length items are generally placed in the middle,** such items typically include the **control link**, the **access link**, and the **machine status fields**.

   If exactly the same components of the machine status are saved for each call, then the same code can do the saving and restoring for each.

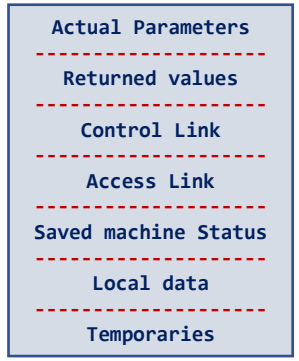


Figure 7.5: A general activation record

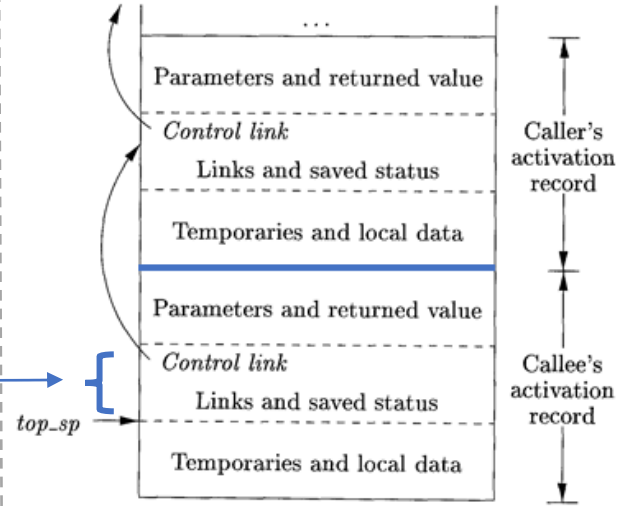


Figure 7.7: Division of tasks between caller and callee

Calling Sequences

3. **Items whose size may not be known early enough are placed at the end of the activation record**.

   - **Most local variables have a fixed length**, which can be determined by the compiler by examining the type of the variable.
   - **Some local variables have a size that cannot be determined until the program executes.**

     Example: Dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
   - **The amount of space needed for temporaries usually depends on how successful the code-generation phase is in keeping temporaries in registers.**

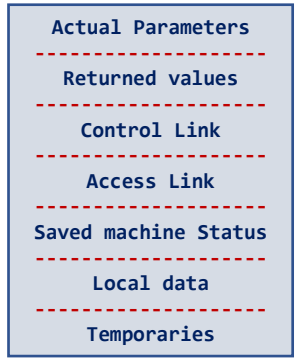


Figure 7.5: A general activation record

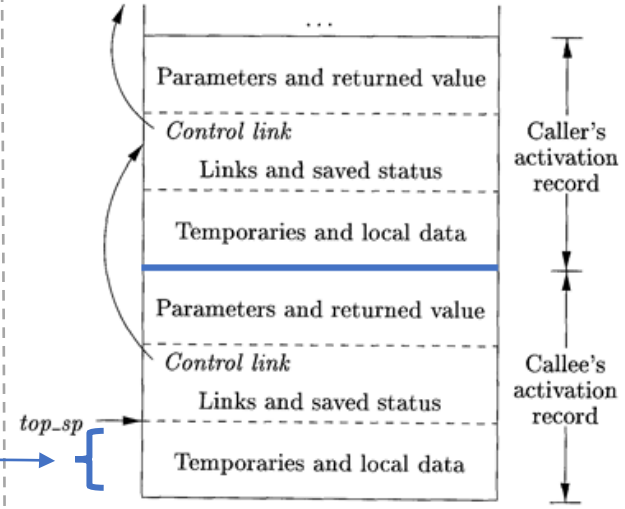
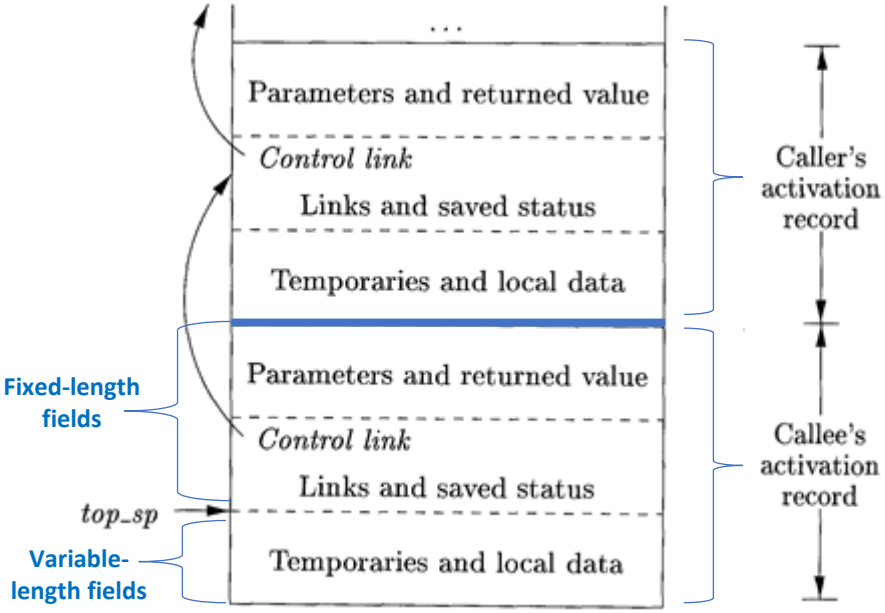


Figure 7.7: Division of tasks between caller and callee

Calling Sequences

4. **Top-of-stack pointer *(i.e., top_sp)* points to the end of the fixed-length fields in the activation record.**
   - **Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.**
   - **Variable-length fields in the activation records are actually "above" the top-of-stack.** Their offsets need to be calculated at run time, but they too can be accessed from the top-of stack pointer, by using a positive offset.



Calling Sequences

- An example of how caller and callee might cooperate in managing the stack is suggested by Fig. 7.7.
  - A register *top-sp* points to the end of the machine status field in the current top activation record.
  - The *top-sp* pointer position within the callee's activation record is known to the caller,
    so the caller can be made responsible for setting *top-sp*
    before control is passed to the callee.

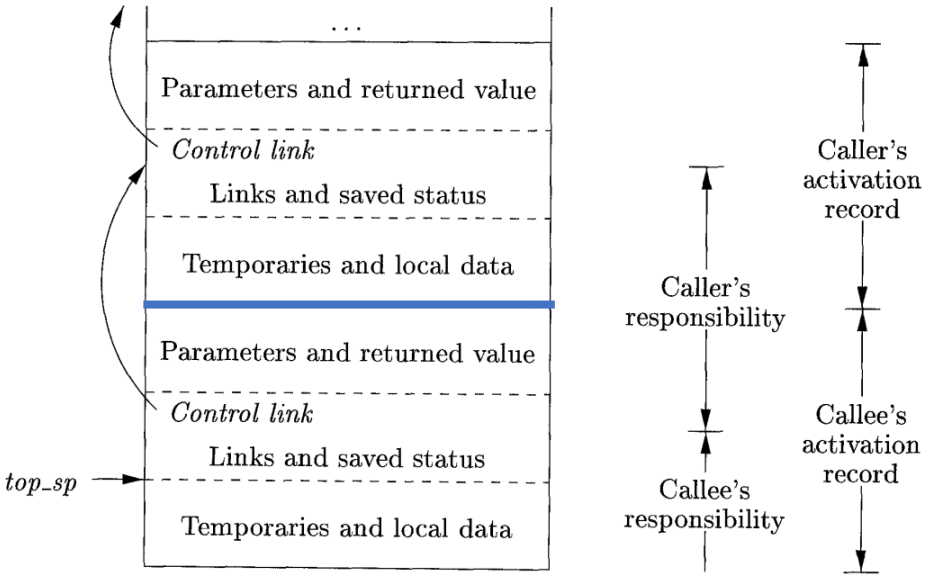


Figure 7.7: Division of tasks between caller and callee

- The *calling sequence* and its division between caller and callee is as follows:

1. The caller evaluates the actual parameters and stores them in the beginning of callee's activation record.
2. The caller stores a return address into the callee's activation record.
3. The caller stores the old value of top-sp into the callee's activation record (i.e., in control link field)

   The caller then increments top-sp to the position shown in Fig. 7.7.

   That is, top-sp is moved past the caller's local data and temporaries and the callee's parameters and status fields.
4. The callee saves the register values and other status information.
5. The callee initializes its local data and begins execution.

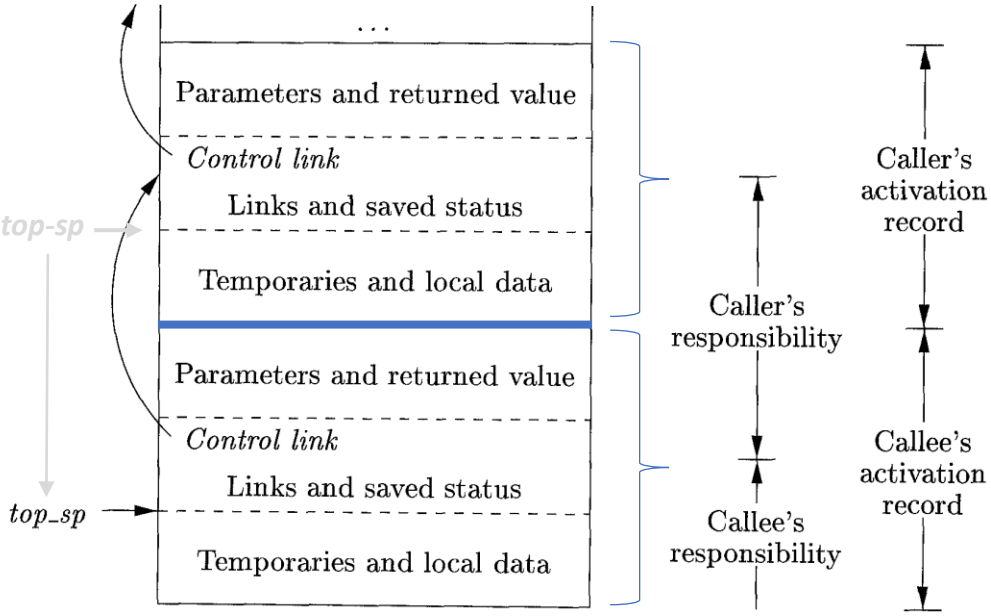


Figure 7.7: Division of tasks between caller and callee

➢ *A return sequence is:*

1. The callee places the return value next to the parameters, as in Fig. 7.5.
2. Using information in the machine-status field, the callee restores top-sp and other registers, and then branches to the return address that the caller placed in the status field.
3. Although top-sp has been decremented, the caller knows where the return value is, relative to the current value of top-sp; the caller therefore may use that value.

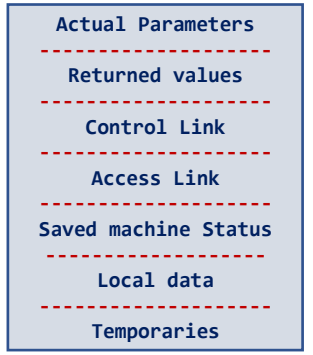


Figure 7.5: A general activation record

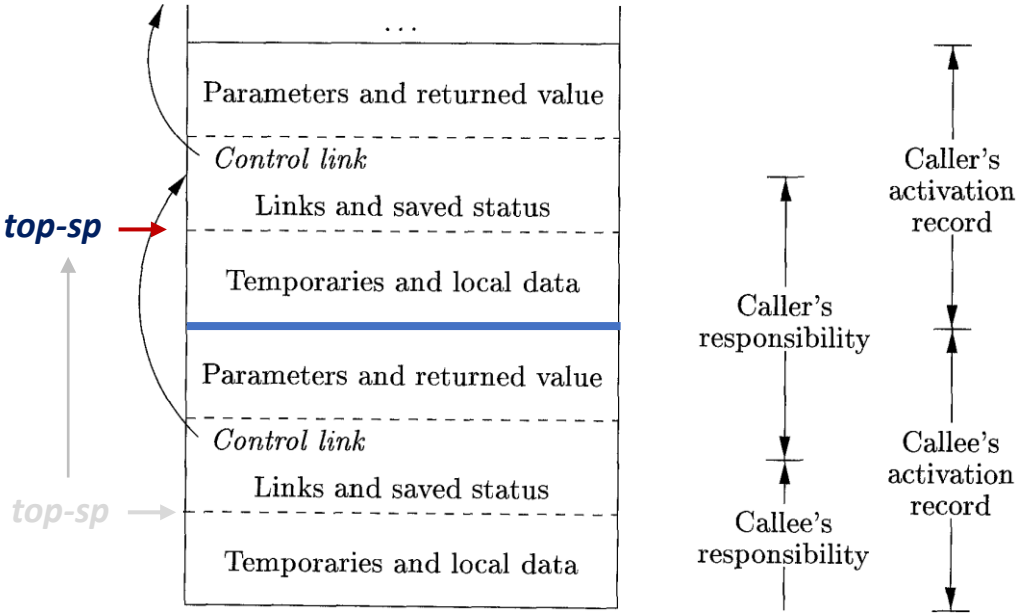


Figure 7.7: Division of tasks between caller and callee

COMPILER DESIGN

Unit 5: Run-Time Environments

Stack Allocation of Space: Variable-Length Data on the Stack

Prakash C O
Department of Computer Science and Engineering

Variable-Length Data on the Stack

- The run-time memory-management system must deal frequently with *the allocation of space for objects the sizes of which are not known at compile time, but which are local to a procedure and thus may be allocated on the stack.*
- For example, a local array whose size depends upon a parameter
- In modern languages, objects whose size cannot be determined at compile time are allocated space in the heap but would require garbage collection.

```
int foo(int x, int y)
{
  int m, n, *p;
  p = (int *)malloc(sizeof(int)*x);
   …
  …
}
```

Variable-Length Data on the Stack

- **A common strategy for allocating variable-length arrays is shown in Fig.**

  (***variable-length arrays*** are arrays whose size depends on the value of one or more parameters of the called procedure)

- **A register *top-sp* points to the end of the machine status field in the current top activation record.**

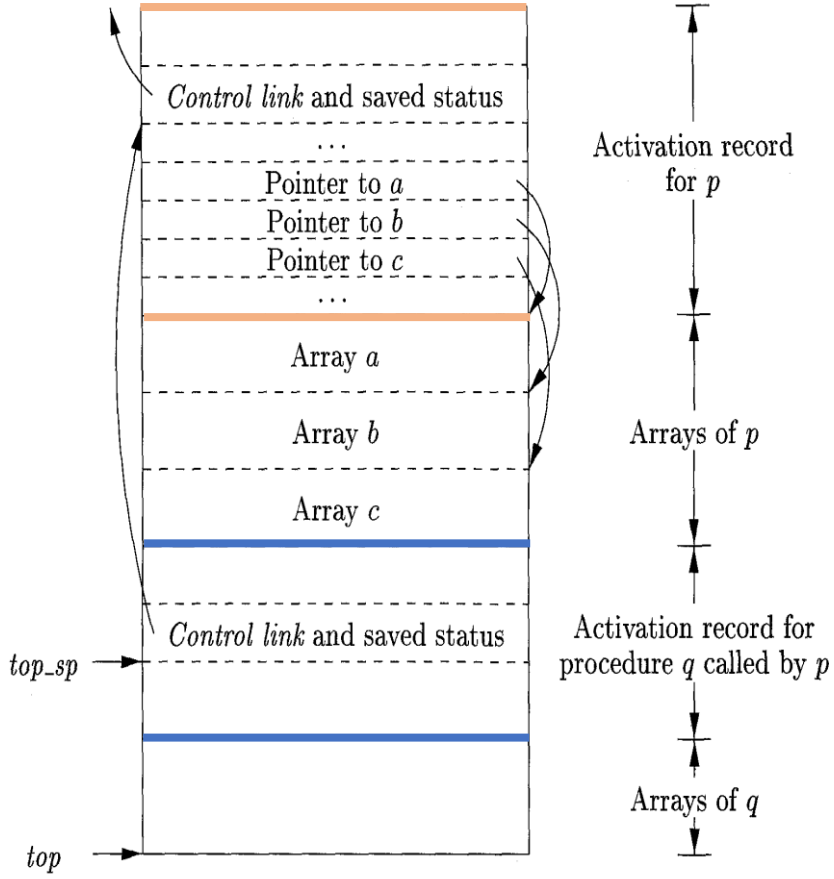


Figure 7.8: Access to dynamically allocated arrays

Variable-Length Data on the Stack

- **In Fig. 7.8, Procedure p has three local arrays, whose sizes we suppose cannot be determined at compile time.**
  - **The storage for these arrays is not part of the activation record for p**, although it does appear on the stack.
  - **Only a pointer to the beginning of each array appears in the activation record itself.**
  - When p is executing, these pointers are at known offsets from the top-of-stack pointer, so the target code can access array elements through these pointers.

Also shown in Fig. 7.8 is the activation record for a procedure q, called by p. The activation record for q begins after the arrays of p, and any variable-length arrays of q are located beyond that.

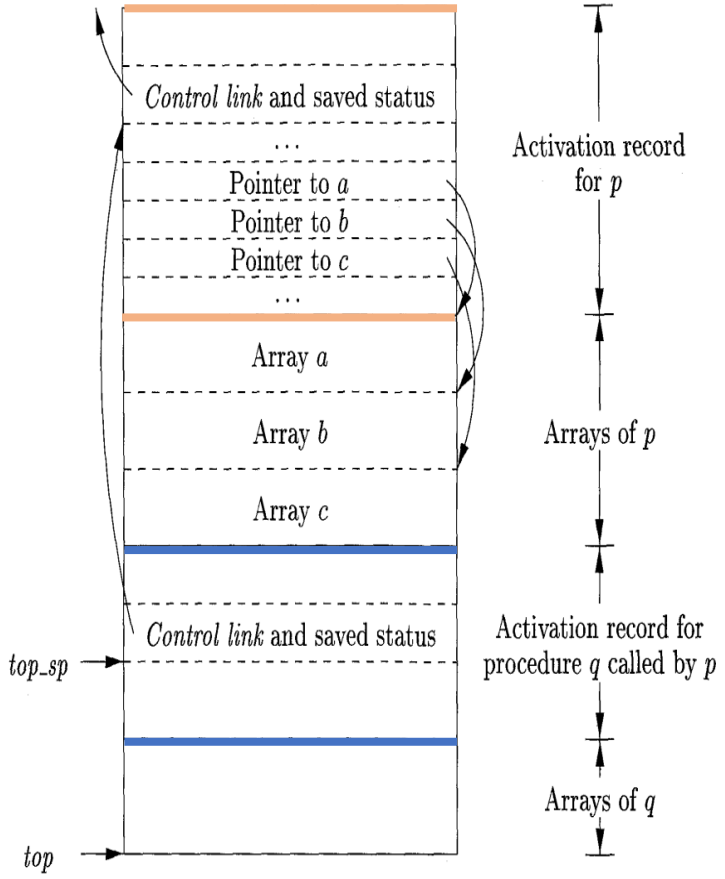


Figure 7.8: Access to dynamically allocated arrays

Variable-Length Data on the Stack

- **Access to the data on the stack is through two pointers, *top* and *top-sp*.**
  - ***top-sp* is used to find local, fixed-length fields of the top activation record.**
  - ***top* marks the actual top of stack; it points to the position at which the next activation record will begin.**
- In Fig. 7.8, **top-sp** points to the end of this field in the activation record for q. From there, we can find the control-link field for q, which leads us to the place in the activation record for p where top-sp pointed when p was on top.

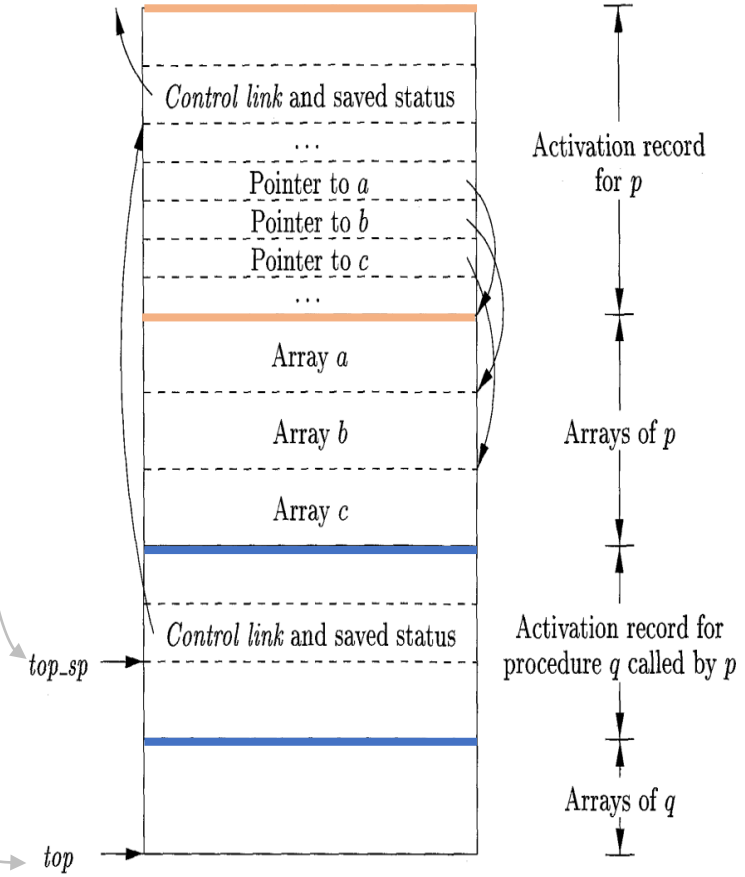


Figure 7.8: Access to dynamically allocated arrays

Variable-Length Data on the Stack

- The **code to reposition top and top-sp can be generated at compile time**, in terms of sizes that will become known at run time.
- **When q returns,**
  - The **top-sp** can be restored from the saved _____ in the activation record for q.
  - The **new value of top** is _____ **minus the length of the** _____ _____ **in q's activation record.** This length is known at compile time to the caller, although it may depend on the caller, if the number of parameters can vary across calls to q.

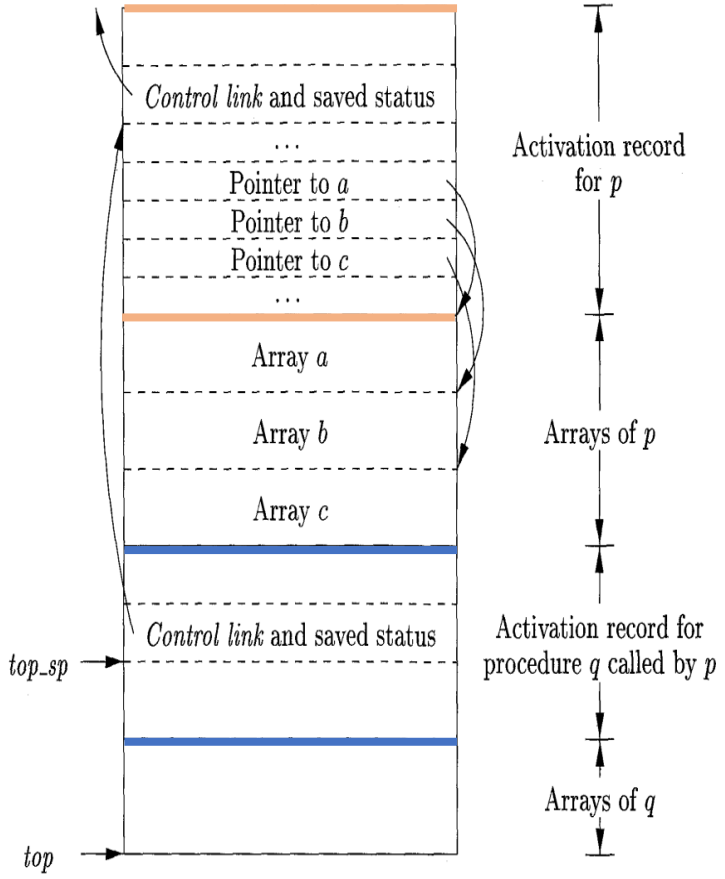


Figure 7.8: Access to dynamically allocated arrays

Variable-Length Data on the Stack

- The **code to reposition top and top-sp can be generated at compile time**, in terms of sizes that will become known at run time.
- **When q returns,**
  - The **top-sp** can be restored from the saved **control link** in the activation record for q.
  - The **new value of top** is **top-sp minus the length of the machine-status, control and access link, return-value, and parameter fields (as in Fig. 7.5) in q's activation record**. This length is known at compile time to the caller, although it may depend on the caller, if the number of parameters can vary across calls to q.

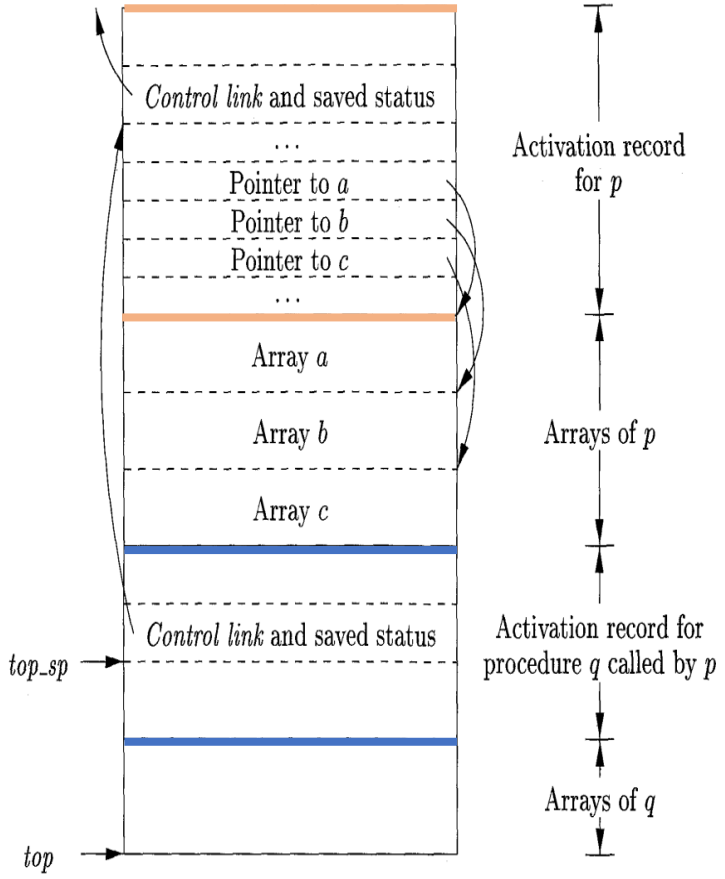


Figure 7.8: Access to dynamically allocated arrays

COMPILER DESIGN

Unit 5: Run-Time Environments

Access to Nonlocal Data on the Stack

Prakash C O
Department of Computer Science and Engineering

Access to Nonlocal Data on the Stack

Let's discuss how the procedures access their data.

- **Local data – For a procedure p, data that belongs to p.**
  - A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.
- **Nonlocal data – Data used within a procedure p, but doesnot belong to p.**
  - Example: A variable used within a procedure p, but defined in procedure q.
- **Access becomes even more difficult when procedures can be declared inside other procedures. Example: ML language – supports nested functions**

Data Access Without Nested Procedures

- **In C and it's family of languages, there is no problem of nested procedures. But the data can have nested scopes(blocks).**
- **Simple rules:**
  - **Global constants are allocated static storage.**

    Their locations(addresses) remain fixed and determined at compile time and hence to access these (not local to any procedure) **we can use statically determined address.**
  - **Data that is local to the procedure is at the activation record at the top of the stack, can be accessed using *top_sp* pointer of the stack.**

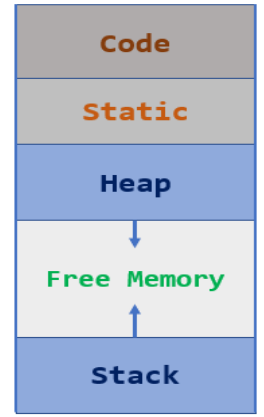


Figure 7.1: Subdivision of run-time memory into code and data areas

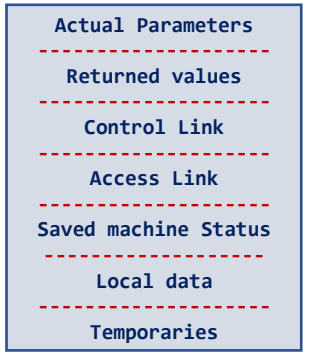
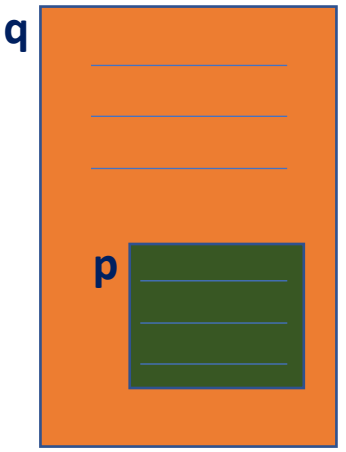


Figure 7.5: A general activation record

Access to Nonlocal data in Nested Procedures

- Access becomes difficult when a language allows nested procedures, i.e a procedure can access variables whose declarations surround it's own declarations.



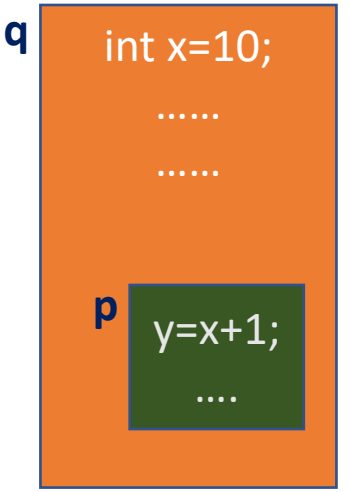
- Reason:
  - If p is within q, relative positions of their activation records at compile time cannot be determined(it's a dynamic decision).
  - p maybe recursive or q maybe recursive or both p and q maybe recursive. This can lead to several activation records of p and q on the stack.

Access to Nonlocal data in Nested Procedures

- **Suppose x is a nonlocal name used in nested procedure p, finding it's declaration is a static decision.**

  But if x is defined in q, finding relevant activation of q from an activation of p is a dynamic decision.

  It requires additional run-time information about activations.



- **Two common strategies: Access links and Displays**

ML : Language with Nested Procedure Declarations

- ML ("Meta Language") is a general-purpose functional programming language.
- ML is statically-scoped i.e., once the variables are declared and initialized it cannot change.
- Defining variables

```
val <name> = <expression>
```

A declaration introduced with keyword val, allows us to give a name of our choice to the expression.

A variable is introduced by binding it to a value as follows:

```
- val x = 4*5;
> val x = 20 : int
- val s = "Abc" ^ "def";
> val s = "Abcdef" : string
```

output

output

Note: - (hyhen) is ML prompt, Notice that the expression to be evaluated is terminated by a semicolon.

ML : Language with Nested Procedure Declarations

- **Declaring functions**
  - *fun* <name> (<arguments>) = < body >
  - ***Examples:*** *fun* inc x = x+1;

    fun double x = 2*x;

    fun adda s = s ^ “a”;

    To execute a function simply give the function name followed by the actual argument. For example:
    - - double 6; output: > 12 : int
    - - inc 100; output: > 101 : int
    - - adda “Indi"; output: > “India” : string

ML : Language with Nested Procedure Declarations

- **For function bodies it is of the form ;**

```
let <list of definitions> in <statements> end
```

The scope of definitions is upto the *in* and till the *end*.

ML : Language with Nested Procedure Declarations

- **Functions are not restricted to using parameters and local variables - they may freely refer to variables that are available when the function is defined.**

- **Consider the following definition:**

```
fun pairwith(x,l) =
    let
        fun p y = (x,y)
    in
        map p l
    end;
```

The local function p has a non-local reference to the identifier x.

```
- pairwith("Hi",l);          Note: l=[1,2,3]
> [("Hi",1),("Hi",2),("Hi",3)] : ( string * int ) list
```

The local function p has a non-local reference to the identifier x, the parameter of the function *pairwith*. The same rule applies here as with other non-local references: the nearest enclosing binding is used.

ML : Language with Nested Procedure Declarations

- ML supports higher-order functions
  - Takes functions as arguments
  - Can construct functions
  - Can return other functions
- ML has no iteration
  - Effect of iteration is achieved through recursion.

    for(i=0;i<10;i++)

    ML makes i a function argument and calls itself until i is reached.

ML : Language with Nested Procedure Declarations

- ML takes lists and labelled tree structures as primitive data types.
- ML doesnot require declaration of it's data types.
- It recognizes types at compile time.
- For example :
  - `var x = 1 is taken to be an integer.`
  - `var y = 2.3 * x is taken to be float.`

Nesting Depth

- Procedures not nested within other procedures have nesting depth 1
  - For example, all functions in C have depth 1
- If p is defined immediately within a procedure at depth i, then p is at depth i+1

Nesting Depth

Quicksort in ML using Nested Procedures

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

Procedure	Nesting depth
sort	
readarray	
excahnge	
quicksort	
partition	

- Procedures not nested within other procedures have nesting depth 1
  - For example, all functions in C have depth 1
- If p is defined immediately within a procedure at depth i, then p is at depth i+1

Nesting Depth

Quicksort in ML using Nested Procedures

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

Procedure	Nesting depth
sort	1
readarray	2
excahnge	2
quicksort	2
partition	3

- Procedures not nested within other procedures have nesting depth 1
  - For example, all functions in C have depth 1
- If p is defined immediately within a procedure at depth i, then p is at depth i+1

COMPILER DESIGN

Unit 5: Run-Time Environments

Access to Nonlocal Data on the Stack:

Access Links and Displays

Prakash C O
Department of Computer Science and Engineering

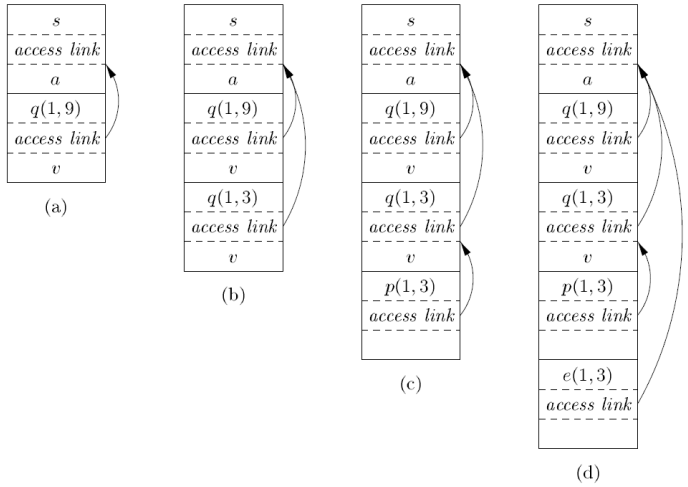
Access Links (Refers to non-local data in other activation record)

- The main purpose of the access link is to access the data which is not present in the local scope of the activation record. It is a static link.
- The chain of access links traces the static structure (or scopes) of the program.
- An access link is a pointer to each activation record that obtains a direct implementation of lexical scope for nested procedures.

Access Links

- *The normal static scope rule for nested functions* is obtained by adding a pointer called the *access link* to each activation record.
- *If procedure p is nested immediately within procedure q* in the source code, then *the access link in any activation record of p points to the most recent activation record of q.*

Note that the nesting depth of q must be exactly one less than the nesting depth of p.

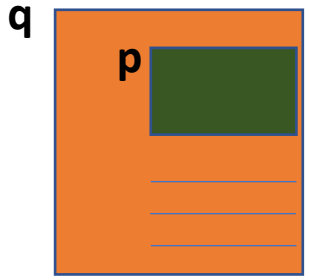



```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                     ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

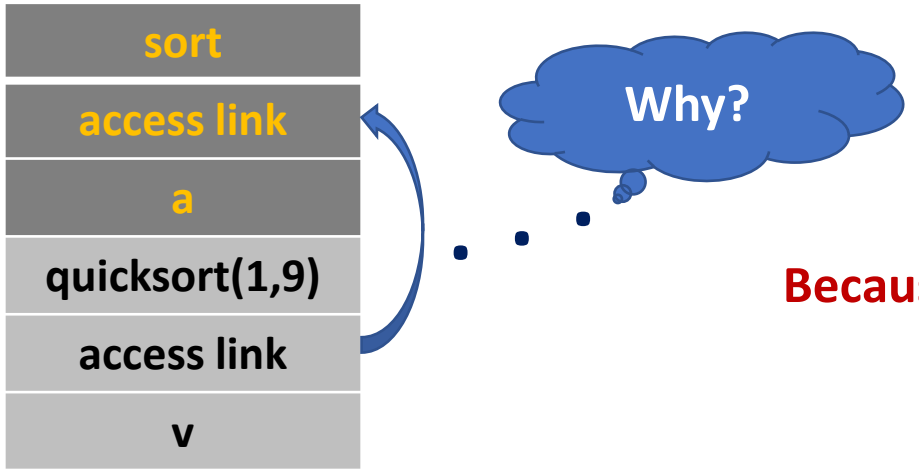
Actual Parameters
Returned values
Control Link
Access Link
Saved machine Status
Local data
Temporaries

Figure 7.5: A general activation record



Access Links

- Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths.
- Along this chain are all the activations whose data and procedures are accessible to the currently executing procedure.
- Example of Access Links



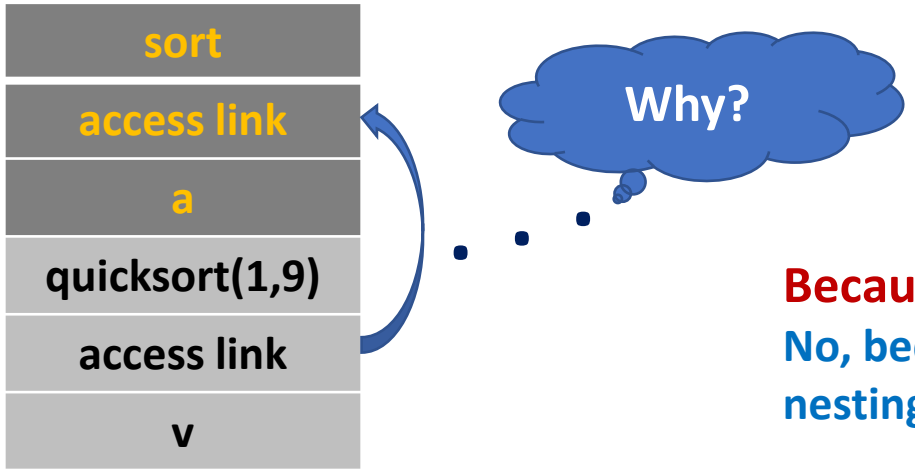
Because sort called quicksort?

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

Access Links

- Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths.
- Along this chain are all the activations whose data and procedures are accessible to the currently executing procedure.
- Example of Access Links



Because sort called quicksort?

No, because sort is the most closely nesting function surrounding quicksort

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

Access Links

➢ Example of Access Links

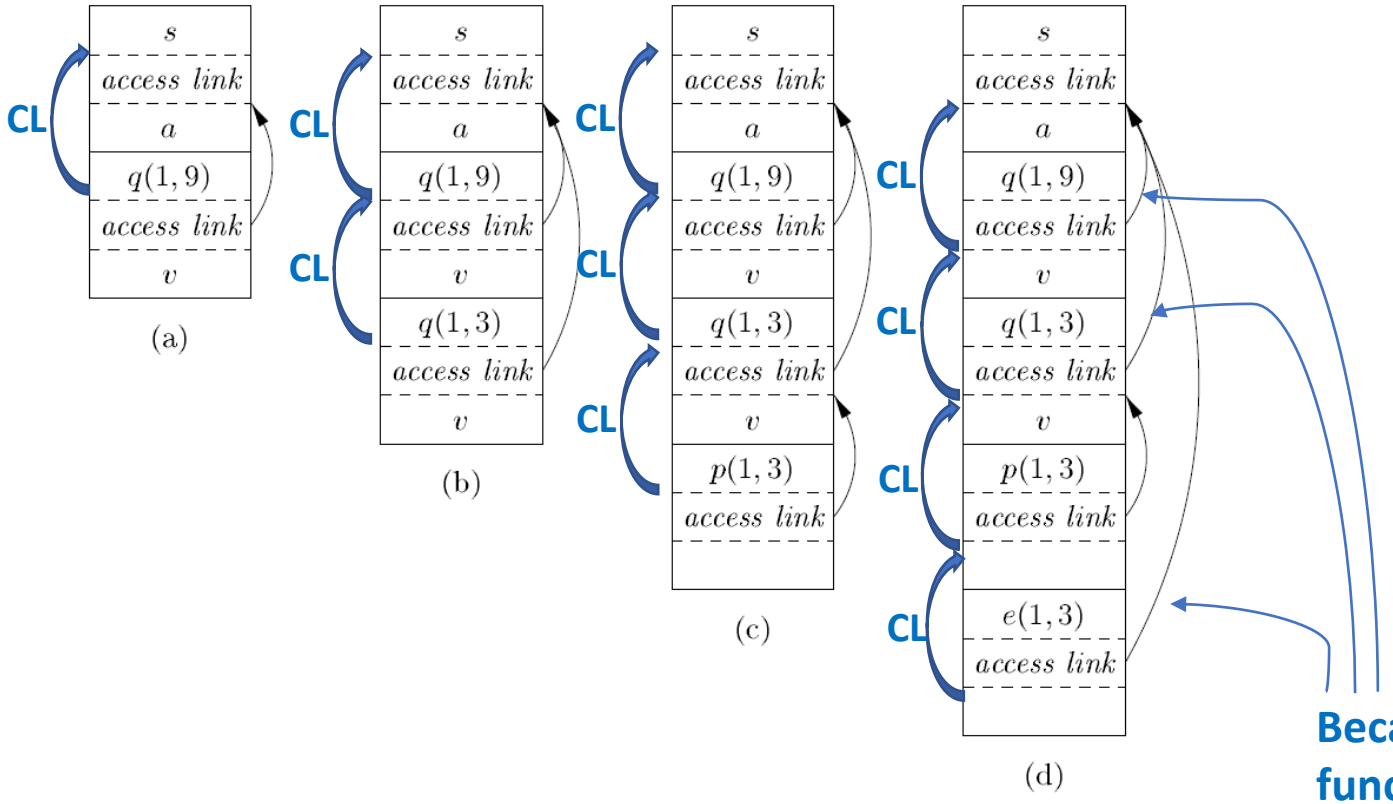


Figure 7.11: Access links for finding nonlocal data

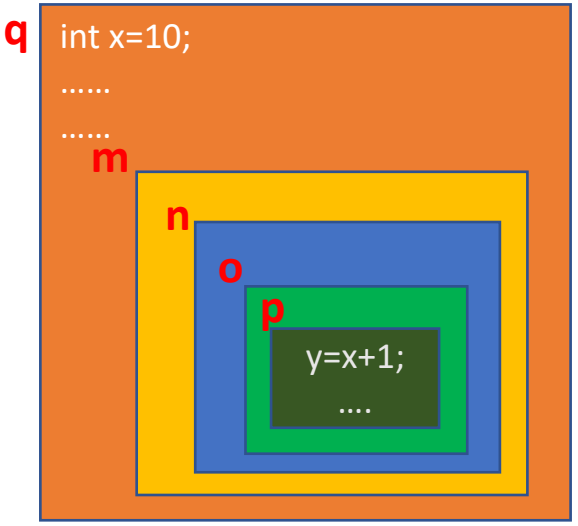
```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

Because sort is the most closely nesting function surrounding quicksort and exchange

Access Links

- Suppose procedure p activation record is at the top of the stack and has nesting depth n_p, and p needs to access x, which is an element defined within some procedure q that surrounds p and has nesting depth n_q.
  - Usually $n_q < n_p$
- To find non-local x, we start at the activation record for p at the top of the stack and follow the access link $n_p - n_q$ times, from activation record to activation record.
- Finally, we wind up at an activation record for q, this activation record contains the element x that we want.



Manipulating Access Links

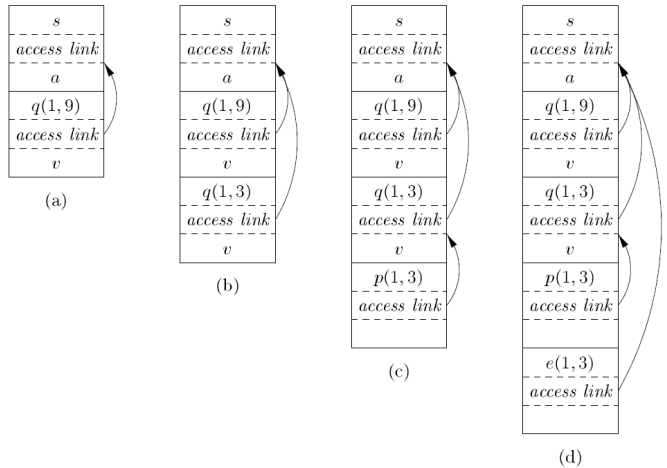
How are access links determined? what should happen when a procedure q calls procedure p, explicitly. There are two cases:

Case 1: $n_q < n_p$ (Procedure p is at a higher nesting depth than q)

- Called procedure p is nested within q
- Therefore, p must be declared in q, or the call by q will not be within the scope of p
- Access link in p should point to the access link of the activation record of the caller q

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

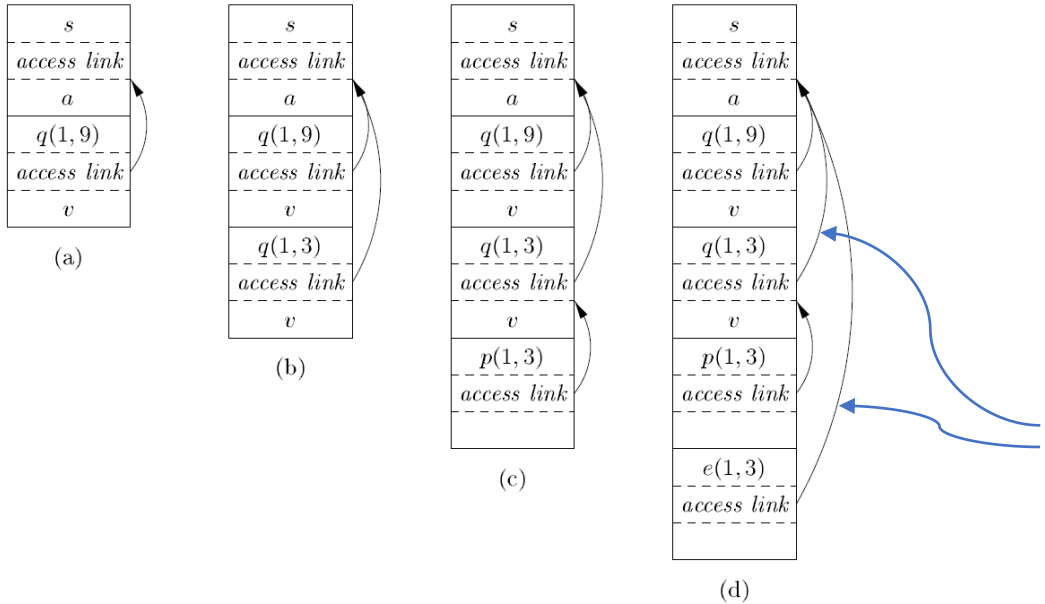


Manipulating Access Links

Case 2: $n_p == n_q$

- Procedures are at the same nesting level
- Access link of called procedure p is the same as q

Example:




```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

- Nesting depth of exchange is 2
- Nesting depth of quicksort is 2

Access link of called procedure *exchange* is the same as *quicksort*

Access links for Procedure Parameters

- **When a procedure p is passed to another procedure q as parameter, q may not know the context in which p appears and its impossible to set access link for p.**
  - **Solution: when procedures are used as parameters, the caller needs to pass proper access link as the parameter along with name of procedure parameter.**
  - Caller always knows the link. i.e, in the above case p must be a name accessible to q, therefore q determines access link for p exactly as if p being called by r directly

Access links for Procedure Parameters

```
fun a(x) =
    let
        fun b(f) =
            ··· f ··· ;
        fun c(y) =
            let
                fun d(z) = ···
            in
                ··· b(d) ···
            end
    in
        ··· c(1) ···
    end;
```

Figure 7.12: Sketch of ML program that uses function-parameters

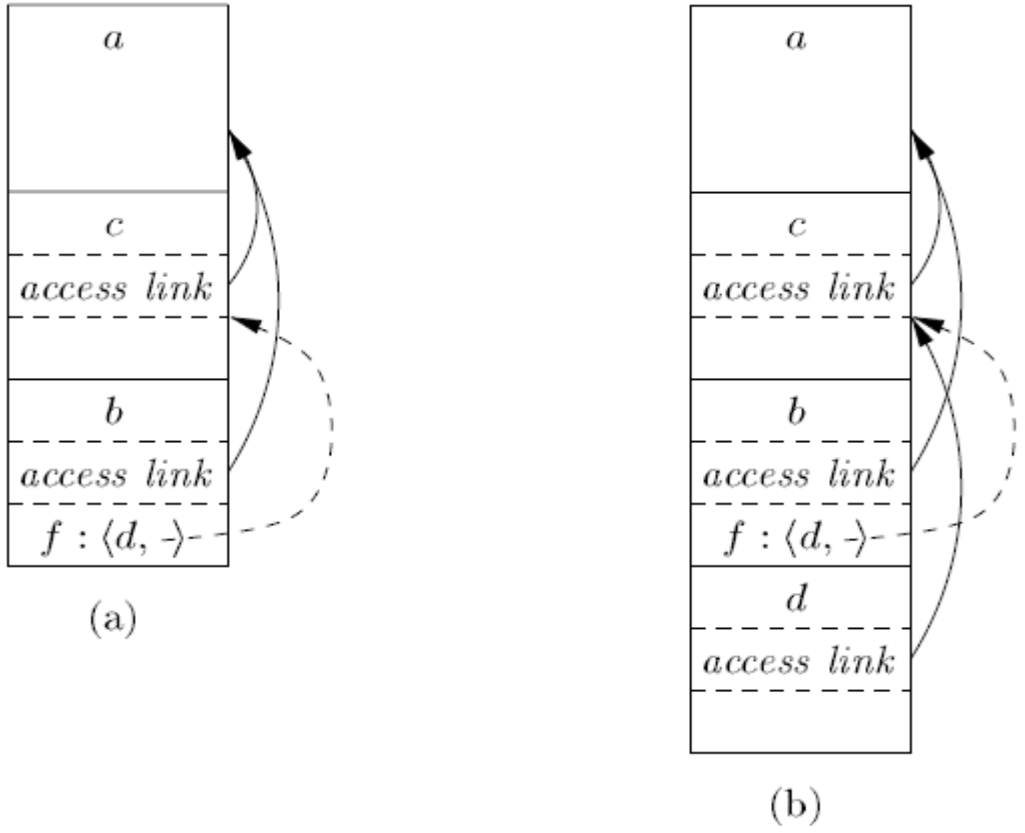


Figure 7.13: **Actual parameters carry their access link with them**

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ···
4)             ··· a ··· ;
5)         fun exchange(i,j) =
6)             ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                    ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Figure 7.10: A version of quicksort, in ML style, using nested functions

Displays

- **One problem with the access-link approach to nonlocal data is that if the nesting depth gets large, we may have to follow long chains of links to reach the data we need.**
- **A more efficient implementation uses an auxiliary array d, called the *display*, which consists of one pointer for each nesting depth.**
- **At all times, *d[i] is a pointer to the highest activation record on the stack for any procedure at nesting depth i*.**
- **Previous value of d[i] must be stored in the current A.R. with depth i.**
- Examples of a display are shown in Fig. 7.14.

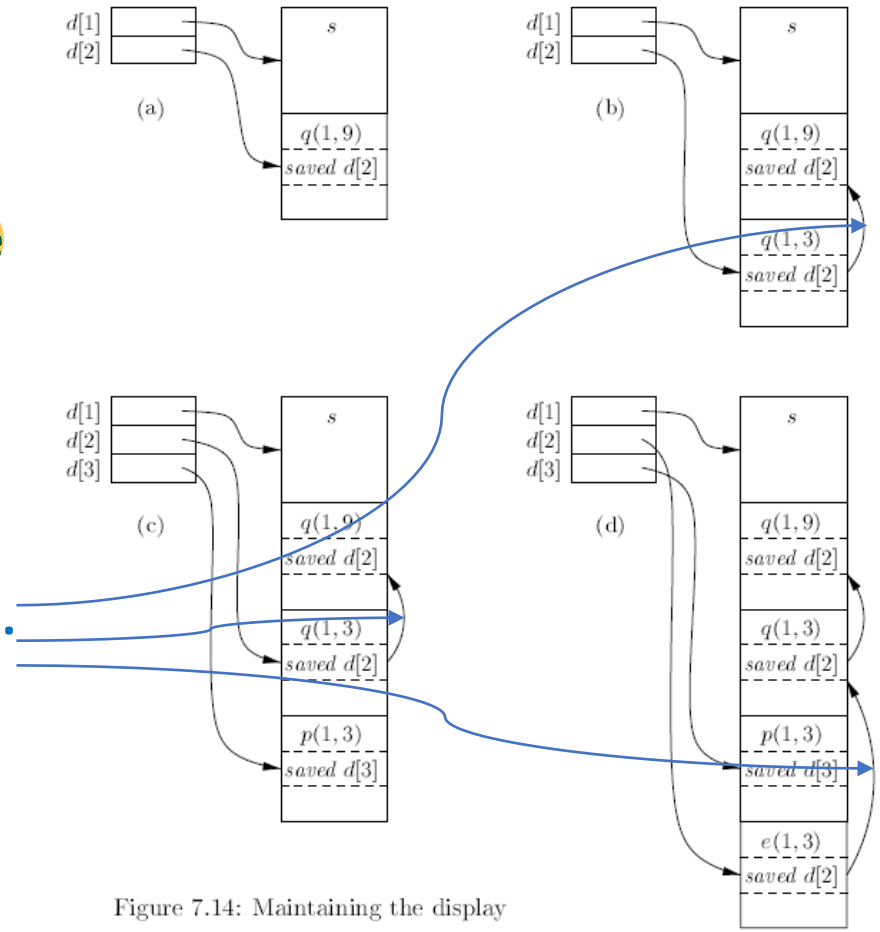


Figure 7.14: Maintaining the display

Maintaining the Display

d[i] is a pointer to the highest activation record on the stack for any procedure at nesting depth i.

Example:

- In Fig. 7.14(d), we see the display d, with
  - d[1] holding a pointer to the activation record for sort, the highest (and only) activation record for a function at nesting depth 1.
  - d[2] holding a pointer to the activation record for exchange, the highest record at depth 2, and
  - d[3] points to partition, the highest record at depth 3.

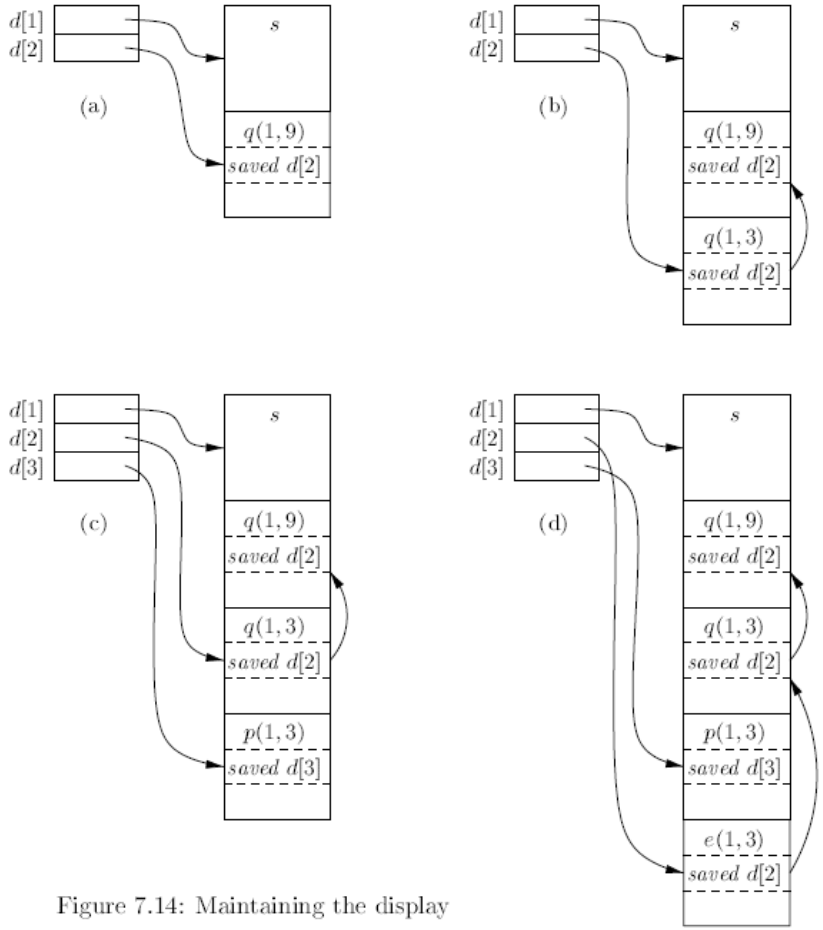


Figure 7.14: Maintaining the display

Maintaining the Display

The advantage of using a display is that

- **Suppose a procedure p is executing and needs to access element x belonging to procedure q.**
  - The runtime only needs to search in activations from $d[i]$, where i is the nesting depth of q.
  - Follow the pointer $d[i]$ to the activation record for q, wherein x should be defined at a known offset.

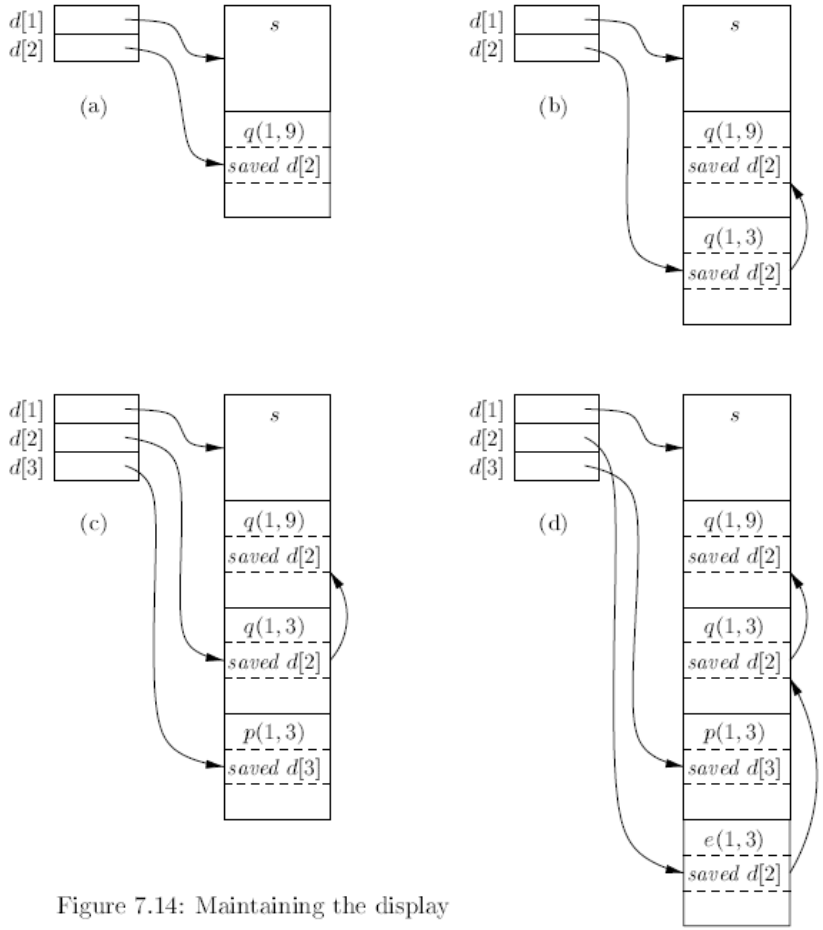


Figure 7.14: Maintaining the display

Access Links Vs Displays

Access Links	Displays
Cost of lookup varies Common case is cheap, but long chains can be costly	Cost of lookup is constant
Cost of maintenance also is variable	Cost of maintenance is constant

References

- **Compilers–Principles, Techniques and Tools,** Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman, 2nd Edition

THANK YOU

Prakash C O

Department of Computer Science and Engineering

coprakasha@pes.edu

+91 98 8059 1946

Manipulating Access Links

How are access links determined? what should happen when a procedure q calls procedure p, explicitly. There are two cases:

Case 1: $n_q < n_p$ (Procedure p is at a higher nesting depth than q)

- **Then p must be defined immediately within q, or**

  the call by q would not be at a position that is within the scope of the procedure name p. **Thus, the nesting depth of p is exactly one greater than that of q, and the access link from p must lead to q.**

- **It is a simple matter for the calling sequence to include a step that places in the access link for p a pointer to the activation record of q.**
- **Examples include the call of quicksort by sort to set up Fig. 7.11(a), and the call of partition by quicksort to create Fig. 7.11(c).**

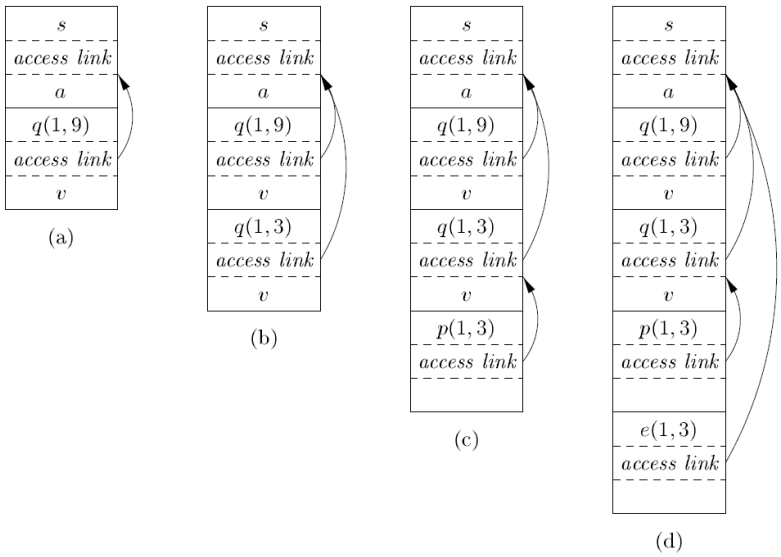


Figure 7.11: Access links for finding nonlocal data

Manipulating Access Links

Case 2: $n_q > n_p$ (The nesting depth n_p of p is less than the nesting depth n_q of q)

- **In order for the call within q to be in the scope of name p, procedure q must be nested within some procedure r, while p is a procedure defined immediately within r.**
- **The top activation record for r can therefore be found by following the chain of access links, starting in the activation record for q, for $n_q - n_p + 1$ hops. Then, the access link for p must go to this activation of r.**

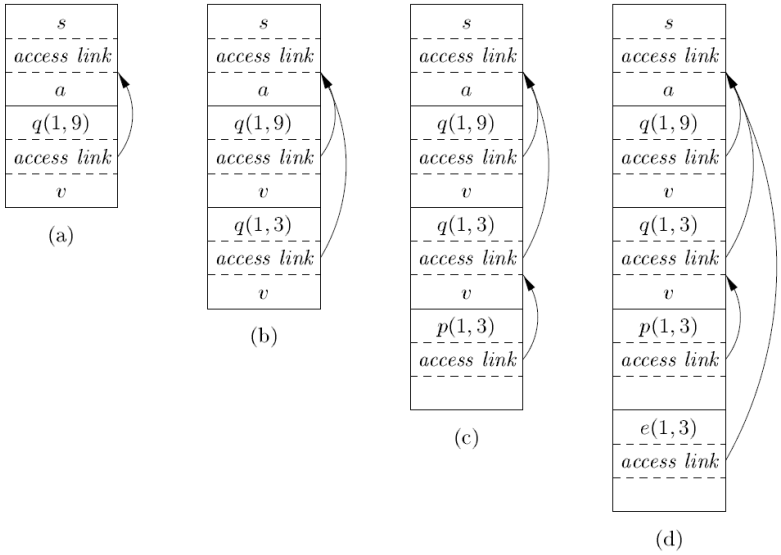


Figure 7.11: Access links for finding nonlocal data

- Nesting depth of exchange is 2
- Nesting depth of partition is 3

COMPILER DESIGN

Unit 5: Code Generation

Prakash C O

Department of Computer Science and Engineering

COMPILER DESIGN

Unit 5: Code Generation

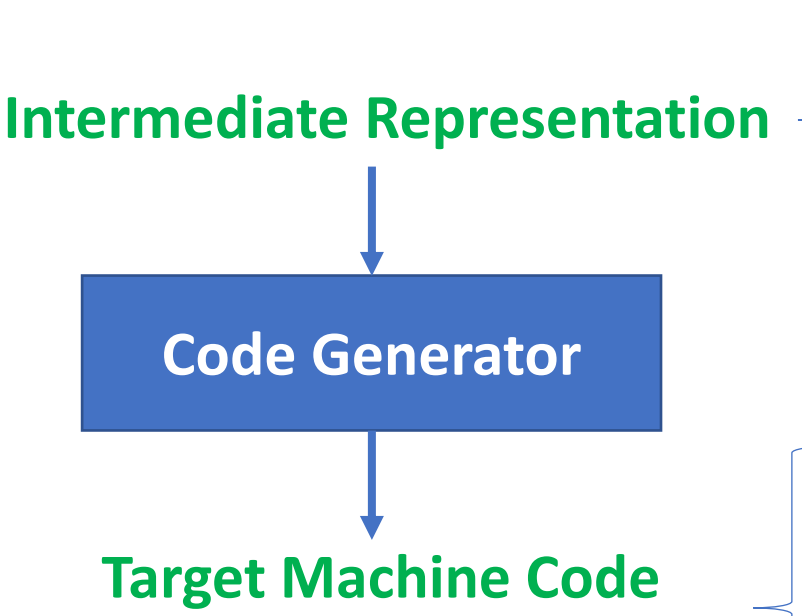
Introduction

Prakash C O
Department of Computer Science and Engineering

Introduction

- **The final phase in our compiler model is the code generator.**
- **Code generator takes as input** the **intermediate representation(IR)** produced by the front end of the compiler, **along with relevant symbol table information**, and **produces as output a semantically equivalent target program**, as shown in Figure.

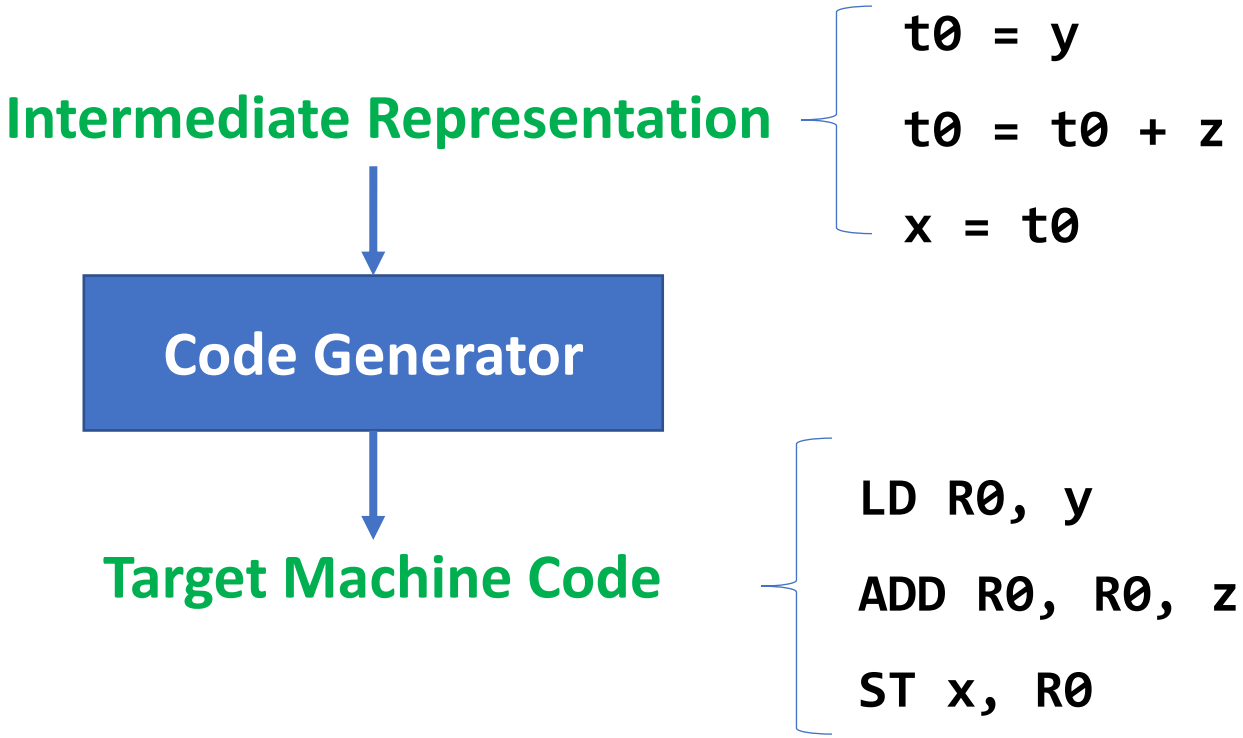
Introduction



- **Linear representations - 3AC/SSA** (Quadruples / Triples / Indirect triples)
- **VM instructions** (Bytecodes / Stack machine codes)
- **Graphical representations** (Syntax trees / DAGs)

- **RISC** (many registers, 3AC, simple addressing modes, simple ISA)
- **CISC** (few registers, 2AC, variety of addressing modes, variable length instructions, Instruction may take more than a single clock cycle to get executed)
- **Stack machine** (push/pop, stack top uses registers, used in JVM, JIT compilation)

Introduction



Introduction

- **The requirements imposed on a code generator are severe.**

1. **The target program must preserve the semantic meaning of the source program.**
   - Meaning intended by the programmer in the original source program should carry forward in each compilation stage until code-generation.
2. **The target program must be of high quality.**
   - Execution time or space or energy or …
3. **The code generator itself must run efficiently.**
   - Instruction Selection, Register Allocation and Instruction ordering.

Introduction

➢**The challenges in code generation are**,

1. Mathematically, the problem of generating an optimal target program for a given source program is *undecidable*;
2. Many of the subproblems encountered in code generation such as register allocation are *Computationally intractable(NP-Hard)*.

Introduction

- **A code generator has three primary tasks:**

1. **Instruction Selection**
   - It involves choosing appropriate target-machine instructions to implement the IR statements.
2. **Register Allocation & Assignment**
   - It involves deciding what values to keep in which registers.
3. **Instruction ordering**
   - It involves deciding in what order to schedule the execution of instructions

IR Code:

```
t0 = y
t0 = t0 + z
x = t0
```

Target Code:

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

COMPILER DESIGN

Unit 5: Code Generation

Issues in the Design of a Code Generator

Prakash C O
Department of Computer Science and Engineering

Issues in the Design of a Code Generator

- The code generator design issues details are dependent on
  1. the specifics of Intermediate Representation,
  2. the Target Language, and the Run-time system,
  3. tasks such as Instruction Selection, Register Allocation and Assignment, and Instruction Ordering.
- The most important criterion for a code generator is that it produce correct target code.

Issues in the Design of a Code Generator

Issues in the Design of a Code Generator are:

1. Input to the Code Generator
2. The Target Program
3. Instruction Selection
4. Register Allocation
5. Evaluation Order

1. Input to the Code Generator

- **The input to the code generator is**
  1. **The intermediate representation(IR) of the source program produced by the front end**, and
  2. **Information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.**

1. Input to the Code Generator

- The many choices for the intermediate representation(IR) include
  1. **Three-address representations** such as
     - **Quadruples, Triples,** and **Indirect triples**
  2. **Virtual machine representations** such as
     - **Bytecodes** and **stack-machine code**
  3. **Linear representations** such as
     - **Postfix notation**
  4. **Graphical representations** such as
     - **Syntax trees** and **DAG's**.

1. Input to the Code Generator

➢ **Assumptions:**

- **The front end has scanned, parsed, and translated the source program into a relatively low-level Intermediate Representation.**
- **All syntactic** and **static semantic errors have been detected properly.**
  - The necessary type checking has taken place, and that type conversion operators have been inserted wherever necessary.
  - The code generator can therefore proceed on the assumption that **its input is free of these kinds of errors.**

2. The Target Program

- ***The instruction-set architecture of the target machine*** **has a significant impact on the difficulty of constructing a good code generator** that produces high-quality machine code.
- **The most common target-machine architectures are**
  a) **RISC** (Reduced Instruction Set Computer),
  b) **CISC** (Complex Instruction Set Computer), and
  c) **Stack based Architecture**.

2. The Target Program

a) A RISC machine typically has

- many registers,
- three-address instructions,
- simple addressing modes, and
- a relatively simple instruction-set architecture.

Examples of RISC microprocessors are Alpha, ARC, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture, and SPARC

b) A CISC machine typically has

- few registers,
- two-address instructions,
- a variety of addressing modes,
- variable-length instructions.

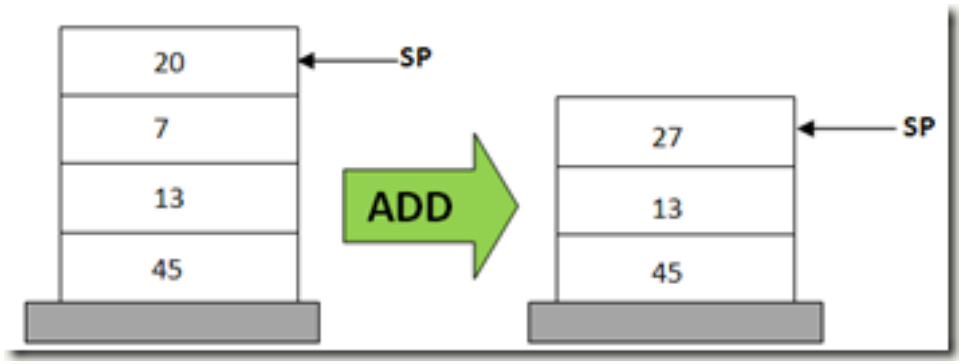
Examples of CISC processors are the System/360, VAX, PDP-11, Motorola 68000 family, AMD and Intel x86 CPUs.

2. The Target Program

c) Stack based Architecture

- Stack-based architectures were revived with the introduction of the JVM.
- **Example:** In a stack based virtual machine, the operation of adding two numbers would usually be carried out in the following manner (where 20, 7, and 'result' are the operands):

```
1.POP 20
2.POP 7
3.ADD 20, 7, result
4.PUSH result
```



2. The Target Program

The target program is the output of the code generator.

The output may be absolute machine language code, relocatable machine language code or assembly language code

- **Producing an *absolute machine-language program* as output has the advantage** that it **can be placed in a fixed location in memory and immediately executed**.
- **Producing a *relocatable machine-language program* (often called an object module) as output allows subprograms to be compiled separately.**

  A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

2. The Target Program

The target program is the output of the code generator.

The output may be absolute machine language code, relocatable machine language code or assembly language code

- **Producing an assembly-language program as output makes the process of code generation somewhat easier.**

  We can generate symbolic instructions and use the macro facilities of the assembler to help generate code.

  The price paid is the assembly step after code generation.

2. The Target Program

- **For readability, we use assembly code as the target language**.

  As long as addresses can be calculated from offsets and other information stored in the symbol table, the code generator can produce relocatable or absolute addresses for names just as easily as symbolic addresses.

3. Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine.

- The complexity of Instruction Selection depends upon
  a) Level of the IR
  b) Nature of the Instruction-Set Architecture(ISA)
  c) Desired quality of the generated code.

IR Code:

```
t0 = y
t0 = t0 + z
x = t0
```

Target Code:

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

3. Instruction Selection

The complexity of Instruction Selection depends upon

a) Level of the IR

- **If the IR is high level**, the code generator may **translate each IR statement into a sequence of machine instructions using code templates**.

  Such **statement-by-statement code generation**, however, often produces poor code that needs further optimization.

- **If the IR reflects some of the low-level details of the underlying machine**, then the code generator can use this information to generate more efficient code sequences.

  - e.g., *intsize* versus *4*.

3. Instruction Selection

The complexity of Instruction Selection depends upon

b) Nature of the Instruction-Set Architecture(ISA)

- ***The nature of the instruction set of the target machine*** **has a strong effect on the difficulty of instruction selection.**

  For example, the uniformity and **completeness of the instruction set are important factors**.

  - If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. **On some machines, for example, floating-point operations are done using separate registers.**
  - The set of instructions are said to be complete if the target machine includes a sufficient number of instructions in each of the category (Arithmetic, logical, shift, move, conditional-and-unconditional-jumps and i/o instructions)

3. Instruction Selection

The complexity of Instruction Selection depends upon

c) Desired quality of the generated code.

- **If we do not care about the efficiency of the target program, instruction selection is straightforward.**
- For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct.
- For example, **every three-address statement of the form x = y + z**, where x, y, and z are statically allocated, **can be translated into the code sequence**

```
LD   R0, y       // R0 = y        (load y into register R0)
ADD  R0, R0, z   // R0 = R0 + z   (add z to R0)
ST   x, R0       // x = R0        (store R0 into x)
```

3. Instruction Selection

- **If we do not care about the efficiency of the target program, instruction selection is straightforward.**

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

a = b + c →

```
LD  R0, b       // R0 = b
ADD R0, R0, c   // R0 = R0 + c
ST  a, R0       // a = R0
```

d = a + e →

```
LD  R0, a       // R0 = a
ADD R0, R0, e   // R0 = R0 + e
ST  d, R0       // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if a is not subsequently used.

3. Instruction Selection

- **The quality of the generated target code is usually determined by its speed and size.**
  - **A given IR program can be implemented by many different target code sequences, with significant cost differences between the different implementations.**
  - **A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.**

    For example, if the target machine has an INC instruction, then **a = a + 1** may be implemented more efficiently by the single instruction **INC a**, rather than

```
LD   R0, a        // R0 = a
ADD R0, R0, #1    // R0 = R0 + 1
ST   a, R0        // a = R0
```

4. Register Allocation

- Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.
- Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.
- A key problem in code generation is deciding what values to hold in what registers.
- Keep values in registers as long as possible to minimize the number of load / store statements executed.
- Values not held in registers need to reside in memory.

4. Register Allocation

- **The use of registers is often subdivided into two subproblems:**
  - **Register allocation**, during which we **select the set of variables that will reside in registers** at each point in the program.
  - **Register assignment**, during which we **pick the specific register that a variable will reside in.**

Register allocation - deciding which values to keep in registers. **Register assignment -** choosing specific registers for values.

- Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Mathematically, the problem is NP-complete.

4. Register Allocation

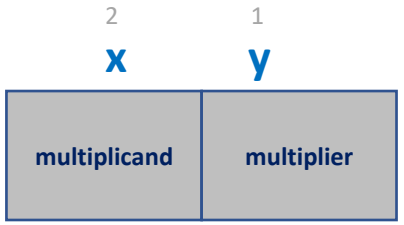
- **The register allocation problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.**
- **Example 8.1 : Certain machines require register-pairs (an even and next odd numbered register) for some operands and results.**

  For example, on some machines, integer multiplication and integer division involve register pairs.

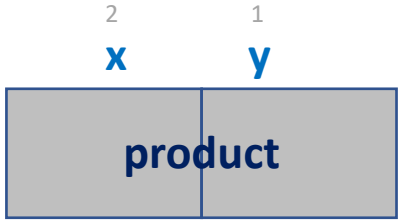
  - The multiplication instruction is of the form `M x, y`

    where x, the multiplicand, is the even register of an even/odd register pair and y, the multiplier, is the odd register. The product occupies the entire even/odd register pair.

Before operation:



After operation:



4. Register Allocation

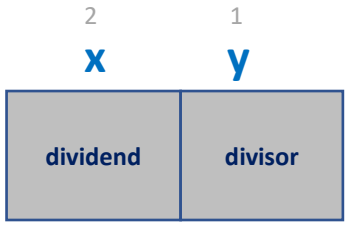
➢ **Example 8.1 :** cont…

- The division instruction is of the form

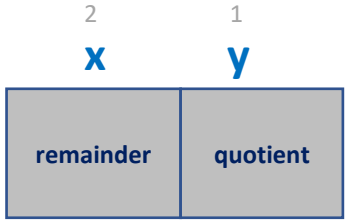
```
D x, y
```

where the dividend occupies an even/odd register pair whose even register is x; the divisor is y. After division, the even register holds the remainder and the odd register the quotient.

Before operation:



After operation:



5. Evaluation Order

- **The order in which computations are performed can affect the efficiency of the target code.**
  - Some computation orders require fewer registers to hold intermediate results than others.
  - Picking a best computation order in the general case is a difficult NP-complete problem.
- Initially, we shall avoid the problem by generating code for the TAC in the order in which they have been produced by the intermediate code generator.

COMPILER DESIGN

Unit 5: Code Generation

Target Machine Model (Hypothetical Model)

Prakash C O
Department of Computer Science and Engineering

Target Machine Model

- **Our target computer models a three-address machine** with
  1. Load and Store operations,
  2. Computation operations,
  3. Jump operations, and
  4. Conditional jumps.
- **The underlying computer is a byte-addressable machine with n general-purpose registers, R0, R1, . . . , R_{n-1}.**
- To avoid hiding the concepts in a myriad of details, **we shall use a very limited set of instructions and assume that all operands are integers.**
- **Most instructions consists of an operator, followed by a target, followed by a list of source operands. Ex:** `op dest, src1, src2`

  `op dest, src`

Target Machine Model

➢ **We assume the following kinds of instructions are available:**

1. **Load (from memory)** (LD reg, (memloc)) — dest, src
2. **Store (to memory)** (ST (memloc), reg) — dest, src
3. **Move (b/w registers)** (MOV reg1, reg2) — dest, src
4. **Computations** (op, dest, src1, src2) — reg

   a) ADD
   b) SUB
   c) MUL
   d) DIV

may be register, memory location
Or immediate constant.

Target Machine Model

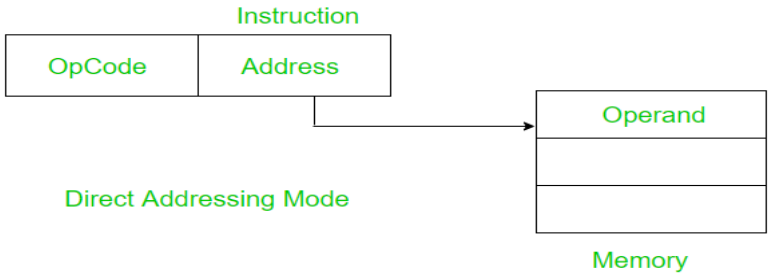
5. **Unconditional jumps** (`BR L`)

6. **Conditional jumps** (`Bcond R, L`)

cond : `LTZ, GTZ, EZ, LTEZ, GTEZ`

For example, **`BLTZ R, L`** causes a jump to label L if the value in register R is less than zero, and allows control to pass to the next machine instruction if not.

Target Machine Model



Addressing Modes

1. Direct Addressing mode:

- In direct addressing mode, the address field contains the address of the operand.
- **Example:** `Add the content of R1 and the content of 1001(memory address) and store the result back to R1`

  `ADD R1, (1001)` Here 1001 is the address where operand is stored.

Example (for hypothetical model): `ADD R1, a`

Note: In our hypothetical model, for simplicity, instead of memory address of a variable, we are using variable name in the instruction itself.

Target Machine Model

Addressing Modes

2. Index addressing mode: Index addressing mode is used to access an array whose elements are in successive memory locations.

Example: `x = a[i]`

```
t1 = 4 * i
t2 = a[t1]
x = t2
```

op	dest	src1	src2
LD	R1	i	
MUL	R1	R1	#4
MOV	R2	R1(a)	
ST	x	R2	

r-value

In the above MOV instruction

R2 ← contents (contents of R1 + a)

R2 ← contents (offset + base address)

Target Machine Model

Addressing Modes

3. Indirect addressing mode:

Here two references are required. First reference to get effective address.

Second reference to access the data.

Example : **x = *p**

```
t1 = *p
x = t1
```

op	dest	src1	src2
LD	R1	p	
MOV	R2	0(R1)	
ST	x	R2	

r-value

In the above MOV instruction: **R2 ← contents(0 + contents of R1),**
that is, loading into R2 the value in the memory location obtained by adding 0 to the contents of register R1.

Target Machine Model

Addressing Modes

4. Immediate addressing mode: **In this mode data is present in address field of instruction.**

Example: `a = 100`

```
a = 100
```

op	dest	src1	src2
LD	R1	#100	
ST	a	R1	

Note: Limitation in the immediate mode is that the range of constants are restricted by size of address field.

Target Machine Model

Note:

1. **Byte-addressable machine.**
2. **N general purpose registers are available:**

   R0, R1, R2…….. Rn-1
3. **Assume all operands are integers.**
4. **Comments are preceded by //**

Generate Three address code and Target code for

1) a[i] = c

```
t1 = 4 * i
a[t1] = c
```

op	dest	src1	src2
LD	R1	i	
MUL	R1	R1	#4
LD	R2	c	
ST	R1(a)	R2	

l-value

Contents of (R1 + a) ← R2

Generate Target code for

2) if x < y goto L

op	dest	src1	src2
LD	R1	x	
LD	R2	y	
SUB	R1	R1	R2
BLTZ	R1	M	

R1 has x

R2 has y

R1 has x-y

M is the equivalent machine instruction generated for label L

Generate Target code for

```
3)      i = 0
        s = 0
 L1:    if i >= n goto L2
        s = s + i
        i = i + 1
        goto L1
 L2:
```

	op	dest	src1	src2
	LD	R1	#0	
	MOV	R2	R1	
	LD	R3	n	
L1:	SUB	R4	R3	R1
	BEZ	R4	L2	
	ADD	R2	R2	R1
	ADD	R1	R1	#1
	BR	L1		
L2:	ST	i	R1	
	ST	s	R2	

R1 represents i, and its initial value is 0

R2 represents s, and its initial value is 0

R3 has value n.

R4 has value n-i

R2 has value s, i.e., s=s+i

R1 has value i, i.e., i=i+1

Generate Target code for

```
4)   F = 1;
     while(N > 0)
     {
       F = F * N;
       N = N – 1;
     }
```

TAC:

```
        F=1
L1 :    if(N <= 0) goto L2
        F = F * N
        N = N - 1
        goto L1
L2 :
```

	op	dest	src1	src2
	LD	R1	#1	
	LD	R2	N	
L1:	BLTEZ	R2	L2	
	MUL	R1	R1	R2
	SUB	R2	R2	#1
	BR	L1		
L2:	ST	F	R1	
	ST	N	R2	

R1 represents F, and its initial value is 1

R2 has value N

R2 has value N, i.e., N=N-1

Exercise 8.2.1: Generate code for the following three-address statements assuming all variables are stored in memory locations.

a) `x = 1`

b) `x = a`

c) `x = a + 1`

d) `x = a + b`

e) The two statements

```
x = b * c
y = a + x
```

Exercise 8.2.2: Generate code for the following three-address statements assuming a and b are arrays whose elements are 4-byte values.

a) The four-statement sequence

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

b) The three-statement sequence

```
x = a[i]
y = b[i]
z = x * y
```

c) The three-statement sequence

```
x = a[i]
y = b[x]
a[i] = y
```

How do we generate Target code when procedures are involved?

```
int findFact(int n){
    int i, fact=1;
    for(i=1;i<=n;i++)
      fact=fact*i;
    return fact;
}
```

Need mechanisms for:

- Passing arguments
- Local Storage
- Returning results
- Linking control

How do we generate Target code when procedures are involved?

Many questions to answer:

1. **What does the dynamic execution of functions look like?**
2. **Where is the executable code for functions located?**
3. **How are parameters passed in and out of functions?**
4. **Where are local variables stored?**

Memory Layout of an executable program

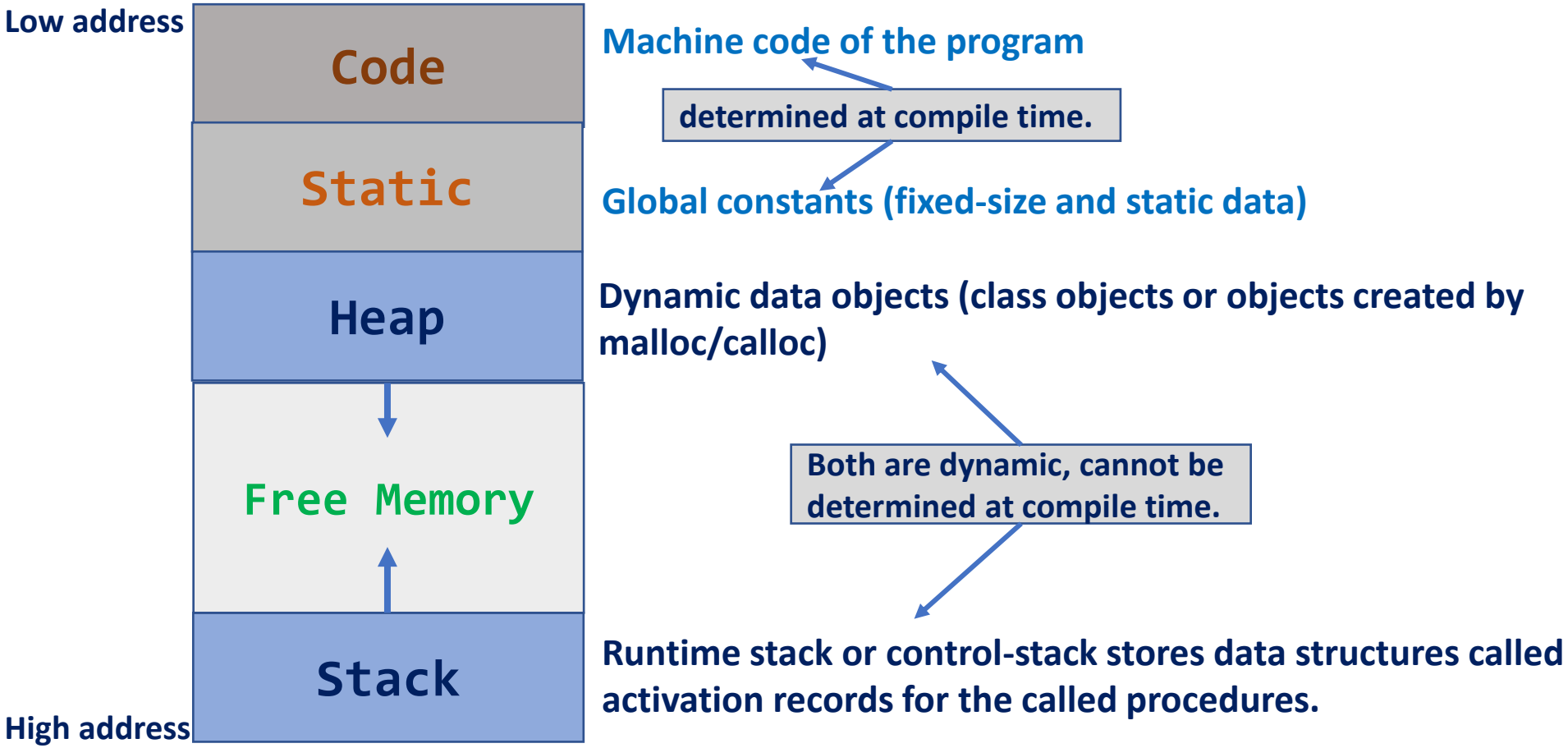


Figure: Subdivision of run-time memory into code and data areas

Code generation for procedures (Static allocation)

Example 1:

IC for Procedure c()

```
action
action
call p
action
halt
```

IC for Procedure p()

```
action
return
```

Assuming static allocation for procedures.

The code for procedures c() and p() are kept at the memory locations 100 and 200 respectively.

Also, the activation records for procedures c() and p() are kept at the memory locations 300 and 364 respectively.

Note: CPU Registers occupies zero bytes in target code instructions. Opcodes, immediate constants, memory addresses and memory operands(variables) needs 4-bytes in target code instructions.

Note: ***action*** **denotes set of three-address statements, target code of action (i.e., ACTION) takes 20 bytes memory.**

Code generation for procedures (Static allocation)

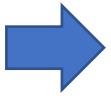
Example 1:

IC for Procedure c()

```
action
action
call p
action
halt
```

IC for Procedure p()

```
action
return
```



Target code for c()

```
100: ACTION
120: ACTION
140: ST 364, 160
152: BR 200
160: ACTION
180: HALT

Target code for p()
200: ACTION
220: BR *364
```

Activation record of c()

```
300: …
```

Activation record of p()

```
364: 160
```

Note: *action* denotes set of three-address statements, target code of action (i.e., ACTION) takes 20 bytes memory.

Code generation for procedures (Static allocation)

Exercise 1: Generate Target code for the following TAC, assuming static allocation for procedures.

The code for procedures p() and q() are kept at the memory locations 100 and 300 respectively.

Also, the activation records for procedures p() and q() are kept at the memory locations 400 and 600 respectively. Assume the names m, n and x represent addresses.

Note: CPU Registers occupies zero bytes in target code instructions. Opcodes, immediate constants, memory addresses and memory operands(variables) needs 4-bytes in target code instructions.

IC for Procedure p()

```
m=5
n=m*2
call q
halt
```

IC for Procedure q()

```
x=2*x
Return
```

Code generation for procedures (Static allocation)

Exercise 1: Solution

IC for Procedure p()

```
m=5
n=m*2
call q
halt
```

IC for Procedure q()

```
x=2*x
return
```

Target code for p()

```
100: LD   R1, #5
108: ST   m, R1
116: MUL  R1, R1, #2
124: ST   n, R1
132: ST   600, 152
144: BR   300
152: HALT
```

Target code for q()

```
300: LD R1, x
308: MUL R1,R1,#2
316: BR *600
```

Activation record of p()

```
400: …
```

Activation record of q()

```
600: 152
```

References

- **Compilers–Principles, Techniques and Tools,** Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman, 2nd Edition
- https://markfaction.wordpress.com/2012/07/15/stack-based-vs-register-based-virtual-machine-architecture-and-the-dalvik-vm/

THANK YOU

Prakash C O

Department of Computer Science and Engineering

coprakasha@pes.edu

+91 98 8059 1946

4. Register Allocation

- Consider the two three-address code sequences in Fig. 8.2 in which the only difference in (a) and (b) is the operator in the second statement. The shortest assembly-code sequences for (a) and (b) are given in Fig. 8.3.

```
t = a + b
t = t * c
t = t / d
```

(a)

```
t = a + b
t = t + c
t = t / d
```

(b)

Figure 8.2: Two three-address code sequences

```
L   R1,a
A   R1,b
M   R0,c
D   R0,d
ST  R1,t
```

(a)

```
L    R0, a
A    R0, b
A    R0, c
SRDA R0, 32
D    R0, d
ST   R1, t
```

(b)

Figure 8.3: Optimal machine-code sequences

Ri stands for register i. SRDA stands for Shift-Right-Double-Arithmetic and **SRDA R0,32** shifts the dividend into R1 and clears R0 so all bits equal its sign bit.

COMPILER DESIGN

Unit 5: Code Generator Algorithm

Prakash C O

Department of Computer Science and Engineering

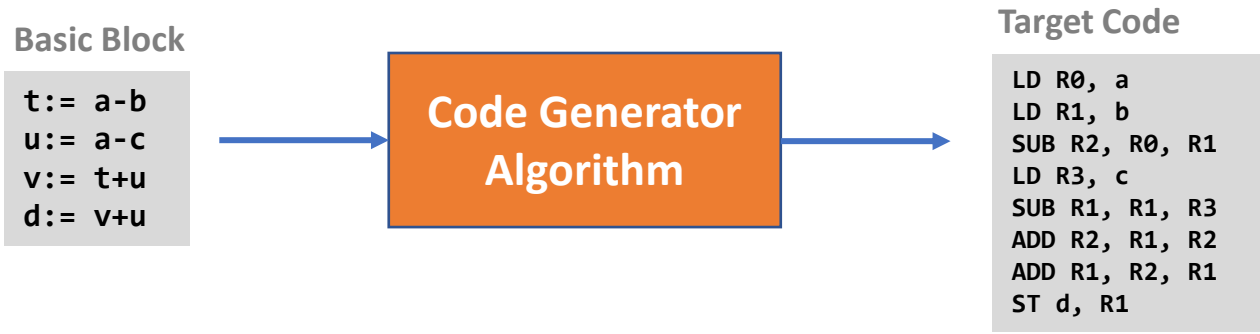
COMPILER DESIGN

Unit 5: Code Generator Algorithm

Prakash C O
Department of Computer Science and Engineering

Unit 5: Code Generator Algorithm

- Consider an algorithm that generates target code for a single basic block.



- Code generator algorithm considers each three-address instruction and keeps track of what values are in what registers so it can avoid generating unnecessary loads and stores.

- One of the primary issues during code generation is deciding how to use registers to best advantage.

➢ **There are four principal uses of registers:**

1. **In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.**

2. **Registers make good temporaries** - places to hold the result of a subexpression while a larger expression is being evaluated, or more generally, **a place to hold a variable that is used only within a single basic block.**

```
a = a + b * c
```

```
t0 =  b * c
t1 =  a + t0
a = t1
```

```
LD  R0, b
LD  R1, c
MUL R0, R0, R1
LD  R1, a
ADD R0, R0, R1
ST  a, R0
```

Here registers R0 and R1 are used to hold operands a, b and c

Here register R0 represents temporary t_0 and hold value of subexpression b*c

Here register R0 represents temporary t_1 and hold value of subexpression a+t_0

➢ **There are four principal uses of registers:** Cont…

3. **Registers are used to hold (global) values that are computed in one basic block and used in other blocks.** For example, a loop index that is incremented going around the loop and is used several times within the loop.

4. **Registers are often used to help with run-time storage management, for example, to manage the run-time stack, including the maintenance of stack pointers and possibly the top elements of the stack itself.**

   These are competing needs, since the number of registers available is limited.

➢The Target machine instructions are of the form

Opcode	Target	Source1	Source2	Description
LD	reg	mem		Move the contents of memory location(i.e., from variable) to register
ST	mem	reg		Move the contents of register to memory location(i.e., to variable)
MOV	reg1	reg2		Move the contents of register2 to register1
OP	reg3	reg1	reg2	Perform operation OP on the contents of register1 and register2, and the result will be stored in register3

Register Descriptor and Address Descriptor

- **The code generator must track both the *registers* (for availability) and *addresses* (location of values) while generating the target code.**

For both, the following descriptors are used:

1. **Register Descriptor**
2. **Address Descriptor**

Register Descriptor and Address Descriptor

➢ **The desired data structure has the following descriptors:**

1. Register Descriptor

- **Register descriptor is used to inform the code generator about the availability of registers.**
- **For each available register, a register descriptor keeps track of the variable names whose current value is in that register**.
- **Assume initially, all register descriptors are empty**. As the code generation progresses, each register will hold the value of zero or more names.

Note: A register descriptor keeps track of the availability of registers and what is currently in each register.

Register Descriptor and Address Descriptor

➢ **The desired data structure has the following descriptors:**

2. Address Descriptor

- **For each program variable, an *address descriptor* keeps track of the location or locations of an variable where the current value of that variable can be found.**
- **The location might be a register, a memory address, a stack location**, or **some set of more than one of these**. The information can be stored in the symbol-table entry for that variable name.

getReg(I) function:

- getReg(I)
  - has access to the *register* and *address descriptors* for all the variables of the basic block and
  - may also have access to certain useful data-flow information such as the variables that are live on exit from the block.
- Function getReg(I) selects registers for each operand(memory location) associated with the three-address instruction I.

```
x = x + y          R1 <- x, R2 <- y and result in R1
```

Algorithm getReg():

Input: Request for a register

Output: A register or the memory location

1. **Return a new empty register if available.**
2. **Else If value of y is stored in a register R and R only holds the value y, and y has no next use, then return R; Update address descriptor: value y no longer in R.**
3. **Else if there are no free registers and find an occupied register R having value y; Store contents by generating *ST M,R* for every M in address descriptor of y; Return register R.**
4. **Else If no such free register could be identified, then the instruction operates on memory location.**

Principle of Register Allocation :

- **If a variable needs to be allocated to a register, the system checks for any free register available, if it finds one, it allocates.**
- **If there is no free register, then it checks for a register that contains a dead variable (a variable whose value is not going to be used in future), and if it finds one then it allocates.**
- **Otherwise it goes for Spilling** (it checks for a register whose value is needed after the longest time, saves its value into the memory, and then use that register for current allocation, later when the old value of the register is needed, the system gets it from the memory where it was saved and allocate it in any register which is available).

Example 1: Generate target code sequence for the following basic block three address statements with register and address descriptors.

Assuming only three registers(R1, R2 and R3) are available and only variables a and d are live on exit from the basic block. Variables t, u and v are temporaries.

```
1.  t:= a-b
2.  u:= a-c
3.  v:= t+u
4.  a:= d
5.  d:= v+u
```

Note: Assuming that all arithmetic operations take their operands from registers.

For each available register, a register descriptor keeps track of the variable names whose current value is in that register.

For each program variable, an *address descriptor* keeps track of the location or locations where the current value of that variable can be found.

Example 1:

TAC	Target Code	Register Descriptor			Address Descriptor						
		R1	R2	R3	a	b	c	d	t	u	v
t = a - b		-	-	-	a	b	c	d			
	LD R1, a	a	-	-	a,R1	b	c	d			
	LD R2, b	a	b	-	a,R1	b,R2	c	d			
	SUB R2, R1, R2	a	t	-	a,R1	b	c	d	R2		
u = a - c	LD R3, c	a	t	c	a,R1	b	c,R3	d	R2		
	SUB R1, R1, R3	u	t	c	a	b	c,R3	d	R2	R1	
v = t + u	ADD R3, R2, R1	u	t	v	a	b	c	d	R2	R1	R3
a = d	LD R2, d	u	d, a	v	R2	b	c	d,R2		R1	R3
d = v + u	ADD R1, R3, R1	d	a	v	R2	b	c	R1			R3
exit	ST a, R2	d	a	v	a,R2	b	c	R1			R3
	ST d, R1	d	a	v	a,R2	b	c	d,R1			R3

Example 2 : Generate target code sequence for the following basic block three address statements with register and address descriptors.

Assuming only two registers (R1 and R2) are available and all the variables are live on exit from the basic block. No temporary variables are used.

```
1.  x:= y+z
2.  z:= x*x
3.  y:= z
4.  x:= y+z
```

Note: Assuming that all arithmetic operations take their operands from registers.

PES UNIVERSITY ONLINE

Example 2 :

For each available register, a register descriptor keeps track of the variable names whose current value is in that register.

For each program variable, an *address descriptor* keeps track of the location or locations where the current value of that variable can be found.

TAC	Target Code	Register Descriptor		Address Descriptor		
		R1	R2	x	y	z
x = y + z		-	-	x	y	z
	LD R1, y	y	-	x	y, R1	z
	LD R2, z	y	z	x	y, R1	z, R2
	ADD R1, R1, R2	x	z	R1	y	z, R2
z = x * x	MUL R2, R1, R1	x	z	R1	y	R2
y = z	ST y, R2	x	z, y	R1	y, R2	R2
x = y + z	ADD R1, R2, R2	x	z, y	R1	y, R2	R2
exit	ST x, R1	x	z, y	x, R1	y, R2	R2
	ST z, R2	x	z, y	x, R1	y, R2	z, R2

Exercise 1: Generate target code sequence for the following basic block three address statements with register and address descriptors.

Assuming only two registers are available, and y, z and d are live on exit from the basic block. No temporary variables are used.

```
1.   x := a+b
2.   y := c-d
3.   z := x+y
4.   d := y*z
```

Note: Assuming that all arithmetic operations take their operands from registers.

```
1.  x := a+b
2.  y := c-d
3.  z := x+y
4.  d := y*z
```

Exercise 1: Solution

For each available register, a register descriptor keeps track of the variable names whose current value is in that register.

For each program variable, an *address descriptor* keeps track of the location or locations where the current value of that variable can be found.

TAC	Target Code	Register Descriptor			Address Descriptor						
		R1	R2		a	b	c	d	x	y	z
x = a + b		-	-		a	b	c	d	x	y	z
	LD R1, a	a	-		a,R1	b	c	d	x	y	z
	LD R2, b	a	b		a,R1	b,R2	c	d	x	y	z
	ADD R2, R1, R2	a	x		a,R1	b	c	d	R2	y	z
	ST x,R2	a	x		a,R1	b	c	d	x,R2	y	z
y = c - d	LD R1, c	c	x		a	b	c,R1	d	x,R2	y	z
	LD R2, d	c	d		a	b	c,R1	d,R2	x	y	z
	SUB R2, R1, R2	c	y		a	b	c,R1	d	x	R2	z
z = x + y	LD R1, x	x	y		a	b	c	d	x,R1	R2	z
	ADD R1, R1, R2	z	y		a	b	c	d	x	R2	R1
	ST z,R1	z	y		a	b	c	d	x	R2	z,R1
d = y * z	MUL R1, R2, R1	d	y		a	b	c	R1	x	R2	z
exit	ST d,R1	d	y		a	b	c	d,R1	x	R2	z
	ST y,R2	d	y		a	b	c	d,R1	x	y,R2	z

What is Spilling?

Formally speaking, spilling is a technique in which, a variable is moved out from a register space to the main memory(the RAM) to make space for other variables, which are to be used in the program currently under execution.

Exercise 2: The program below uses six temporary variables a, b, c, d, e, f.

```
a=15
b=25
c=35
d=a+b
e=c+d
f=c+e
b=c+e
e=b+f
d=15+e
return d+f
```

Assuming that all operations take their operands from registers, what is the minimum number of registers needed to execute this program *without spilling* and *without code optimization*?

Exercise 2: Solution

```
a=15
b=25
c=35
d=a+b
e=c+d
f=c+e
b=c+e
e=b+f
d=15+e
return d+f
```

The Target code (i.e., in assembly code) using the load/store architecture can be written as follows:

```
LD R1, a
LD R2, b
ADD R1, R1, R2        R1 has d, i.e., a+b
LD R2, c              R2 has c

ADD R1, R2, R1        R1 has e, i.e., c+d
ADD R1, R2, R1        R1 has f, i.e., c+e

ADD R3, R2, R1        R3 has b, i.e., c+e
ADD R2, R3, R1        R2 has e, i.e., b+f
ADD R2, R2, #15       R2 has d, i.e., e+15
ADD R1, R2, R1        R1 has d+f

Hence minimum 3 registers required.
```

Exercise 3: Consider the expression (a-10) * (((b * c) / 6)) + d)) and its corresponding TAC

```
t1=b*c
t2=t1/6
t3=t2+d
t4=a-10
t5=t4*t3
```

- Let X be the minimum number of registers required by an optimal target code generation (without any register spill) algorithm for a load/store architecture, in which
  (i) only load and store instructions can have memory operands and
  (ii) arithmetic instructions can have only register or immediate operands

- The value of X is _____.

Exercise 3: Solution

```
t1=b*c
t2=t1/6
t3=t2+d
t4=a-10
t5=t4*t3
```

The Target code (i.e., in assembly code) using the load/store architecture can be written as follows:

```
LD R1, b
LD R2, c
MUL R1, R1, R2
DIV R1, R1, #6
LD R2, d
ADD R1, R1, R2
LD R2, a
SUB R2, R2, #10
MUL R2, R2, R1

Hence minimum 2 registers required.
```

References

- **Compilers–Principles, Techniques and Tools,** Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman, 2nd Edition
- **https://www.javatpoint.com/code-generation**

THANK YOU

Prakash C O

Department of Computer Science and Engineering

coprakasha@pes.edu

+91 98 8059 1946