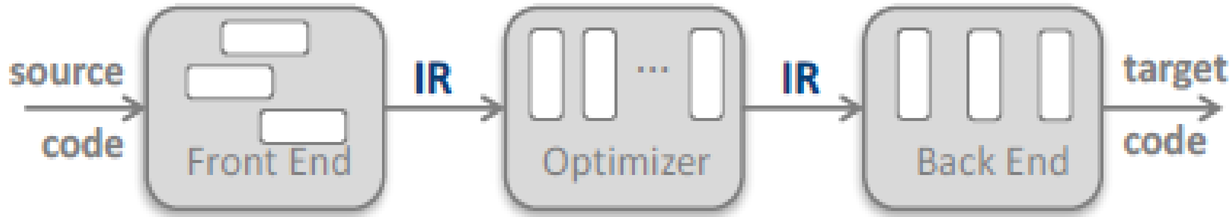
Intermediate Representations

Chapter-5 (Keith Cooper)

Amrita School of Engineering, Bengaluru
Amrita Vishwa Vidyapeetham

Intermediate Representations



IR is the vehicle that carries information between phases

- ***Front end:*** produces an **IR** version of the code
- ***Optimizer:*** transforms the **IR** into an equivalent **IR** that runs faster
  - Each "pass" reads and writes **IR**
- ***Back end:*** systematically transforms the **IR** into native code

IR determines both the compiler's ambition & its chances for success

- The compiler's knowledge of the code is encoded in the **IR**
- The compiler can only manipulate what is represented by the **IR**

Intermediate Representations

Some important IR properties

- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Procedure size
- Expressiveness
- Level of abstraction

The importance of different properties varies between compilers

⇒ Selecting an appropriate **IR** for a compiler is critical

Taxonomy of Intermediate Representations

Three major categories

- Structural **IR**s
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large

Examples:
Trees, **DAG**s

- Linear **IR**s
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange

Examples:
3 address code
Stack machine code

- Hybrid **IR**s
  - Combination of graphs and linear code
  - Example: control-flow graph

Examples:
Control-flow graph
SSA Form

Static Single Assignment

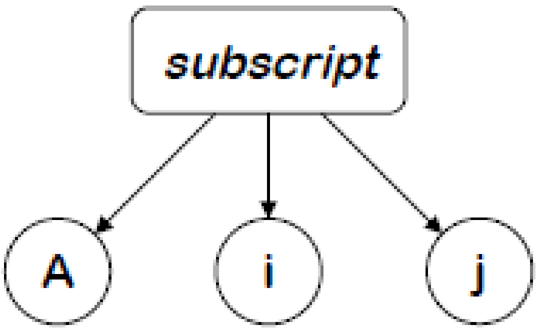
Level of Abstraction

The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

Here are two different representations of an array reference:

Assume an array A[1..10,1..10] of 4 byte elements stored in row-major order and consider how the compiler might represent array reference A[i, j] . Eqn is **A[i,j]= BA+ [(i-1)*N + (j-1)]*S**

A corresponding ILOC code can be written



subI r_i, **1 =>** r_1
multI r_1, **10 =>** r_2
subI r_j, **1 =>** r_3
add r_2, r_3 **=>** r_4
multI r_4, **4 =>** r_5
loadI @A => r_6
Add r_5, r_6 **=>** r_7
load r_7 **=>** r_{Aij}

High level AST

Good for memory disambiguation

Low level linear code

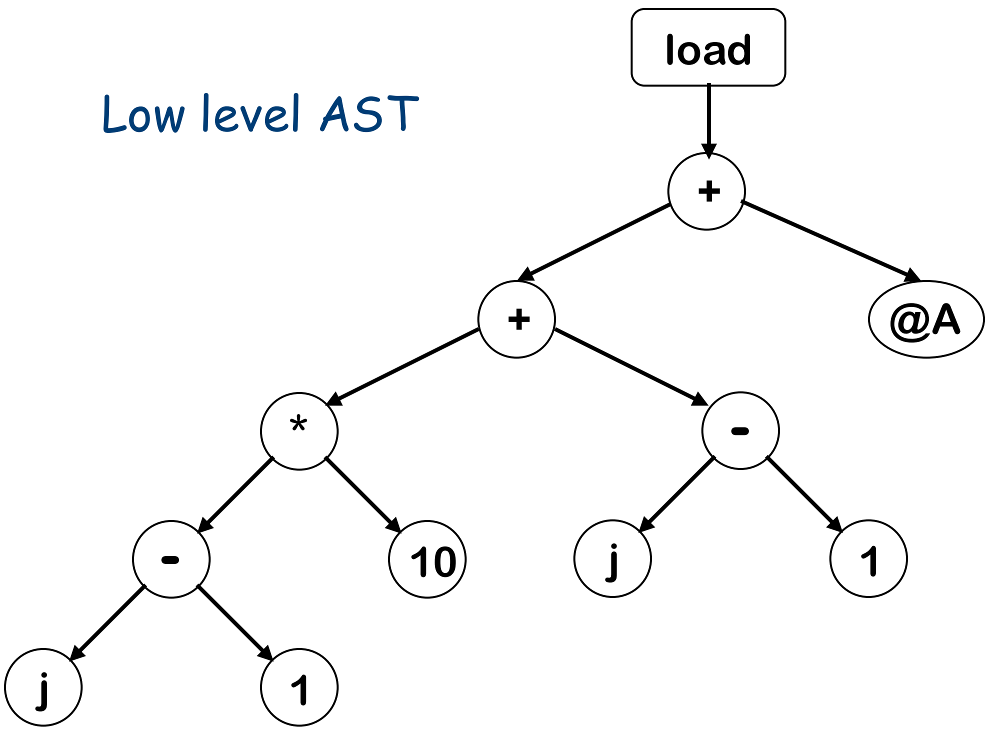
Good for optimizing the address calculation

A[i, j] is an array reference

Level of Abstraction

- Structural *IRs* are usually considered high-level
- Linear *IRs* are usually considered low-level
- Not necessarily true:

Low level AST



loadArray A,i,j

High level linear code

Graphical IRs

- Syntax Related Trees
  1. Parse Trees
  2. Abstract Syntax Trees
  3. Directed Acyclic Graphs

  Levels of Abstractions:
    - **Source Level AST**
    - **Low Level AST(AST for ILOC)**

- Graphs
  1. Control-Flow Graph
  2. Dependence Graph
  3. Call Graph

Graphical IRs

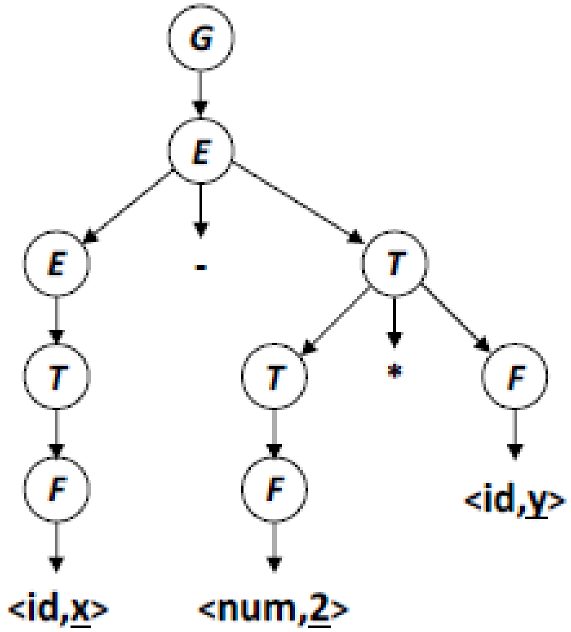
- Syntax Related Trees
  1. Parse Trees
  2. Abstract Syntax Trees
  3. Directed Acyclic Graphs

  Levels of Abstractions

- Graphs
  1. Control-Flow Graph
  2. Dependence Graph
  3. Call Graph

Parse Tree or Syntax Trees

A syntax tree represents the front ends' parse of the code, in detail



***Parse tree for* x - 2 * y**

Syntax trees are often used in source-to-source systems

- Captures the precise (syntactic) form of the input program
- Has all of the detail that you could need
  - Compiler generated all the detail it has from the parse

Syntax trees tend to be inefficient

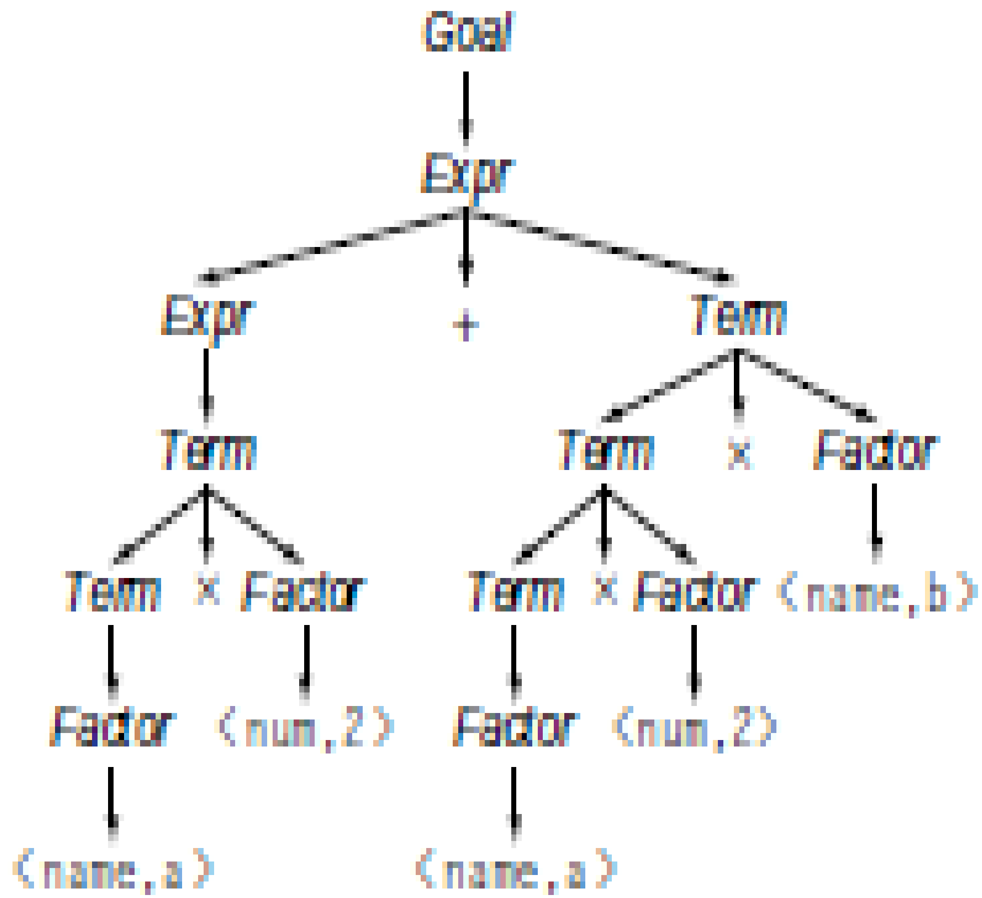
- Lots of unnecessary nodes and edges
- Lots of implicit detail that might be useful to represent explicitly

Syntax trees can be represented with a linear notation (e.g., prefix or postifx)

Parse Tree or Syntax Trees

Goal	$\rightarrow$	*Expr*
Expr	$\rightarrow$	*Expr* + *Term*
	|	*Expr* - *Term*
	|	*Term*
Term	$\rightarrow$	*Term* × *Factor*
	|	*Term* ÷ *Factor*
	|	*Factor*
Factor	$\rightarrow$	(*Expr*)
	|	num
	|	name

(a) Classic Expression Grammar



(b) Parse Tree for a × 2 + a × 2 × b

Graphical IRs

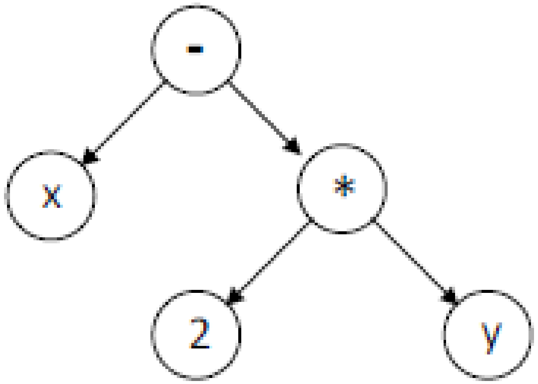
- Syntax Related Trees
  1. Parse Trees
  2. Abstract Syntax Trees
  3. Directed Acyclic Graphs

  Levels of Abstractions

- Graphs
  1. Control-Flow Graph
  2. Dependence Graph
  3. Call Graph

Abstract Syntax Tree (AST)

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed

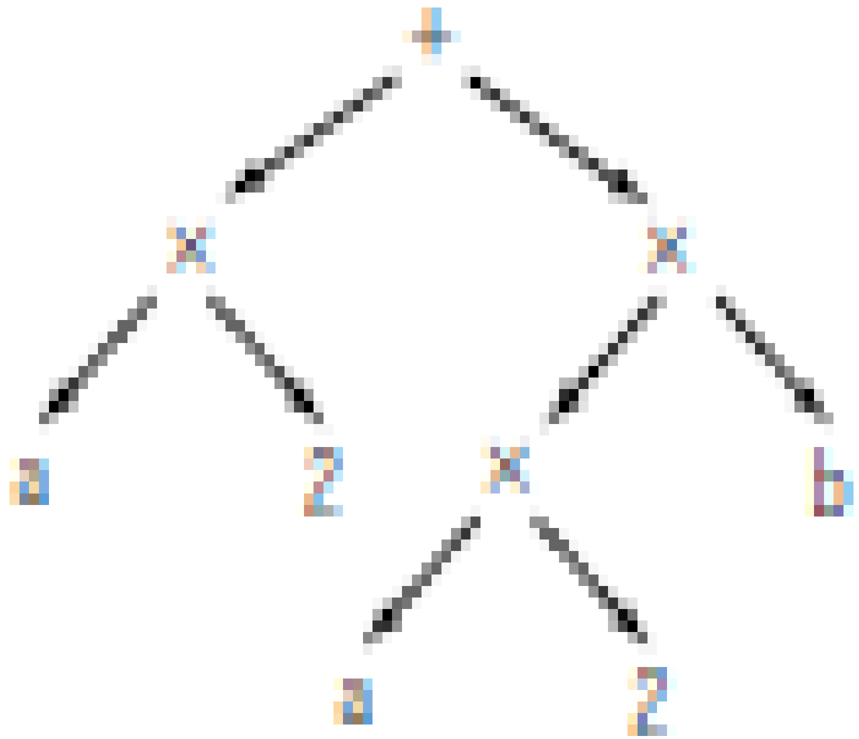


AST for x – 2 * y

ASTs are used as an initial **IR** in some well-known compilers & interpreters.

- **AST**s are space efficient trees that capture most of the interesting information found in a syntax tree
  - Can regenerate source code in a treewalk, with a little cleverness
  - Many fewer nodes and edges than in a syntax tree.
- S-expressions in Scheme or Lisp, are (essentially) **AST**s

Abstract Syntax Tree



AST for a X 2 + a X 2 X b

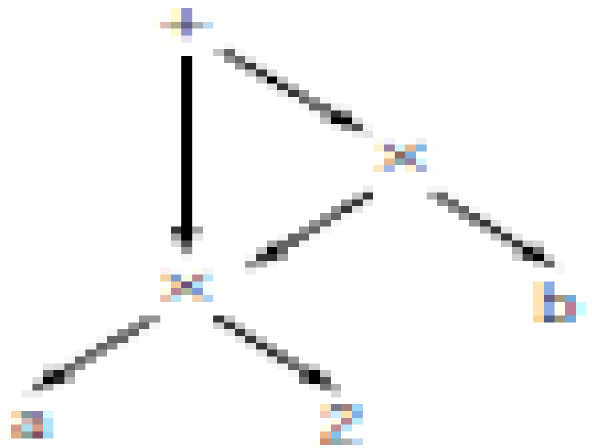
Graphical IRs

- Syntax Related Trees
  1. Parse Trees
  2. Abstract Syntax Trees
  3. Directed Acyclic Graphs

- Graphs
  1. Control-Flow Graph
  2. Dependence Graph
  3. Call Graph

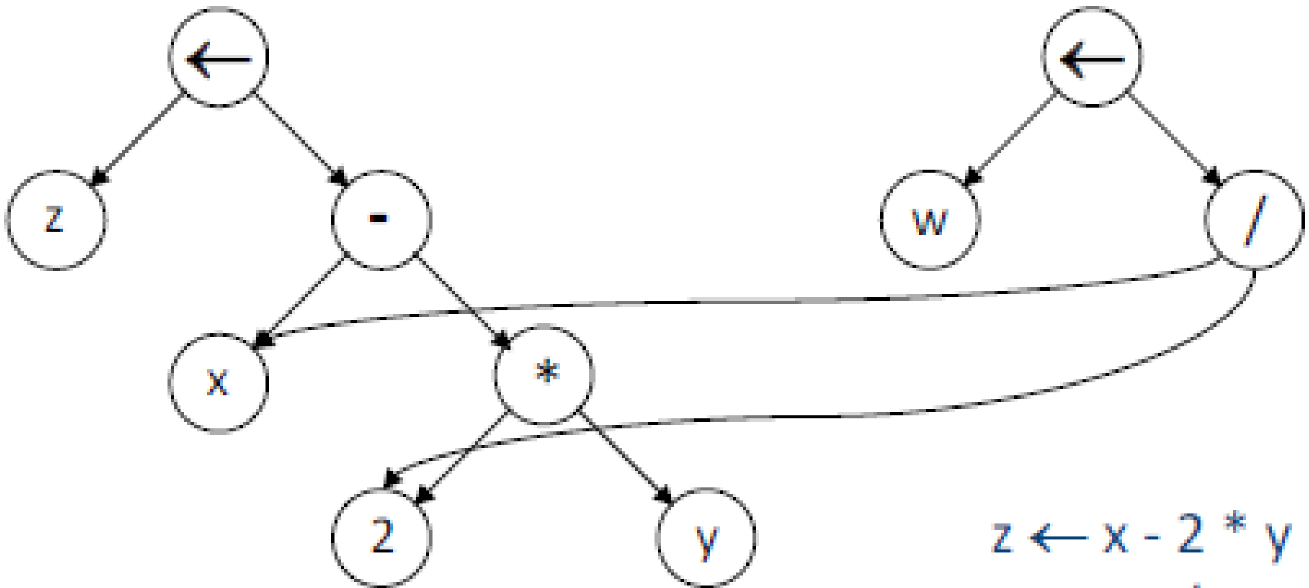
Directed Acyclic Graphs(DAG)

- **A directed acyclic graph (DAG) is an AST with a unique node for each value**
- **It is an AST with sharing. (less memory footprint)**



DAG for a X 2 + a X 2 X b

Directed Acyclic Graphs



$z \leftarrow x - 2 * y$
$w \leftarrow x \;/\; 2$

- Makes sharing explicit
- Encodes redundancy

If the compiler uses graphical **IR**s, a **DAG** is a natural way to represent redundancy.

With two copies of the same expression, the compiler may be able to arrange the code to evaluate it only once.

Graphs

- Control Flow Graph (CFG)
- Dependence Graph
- Call Graph

Control Flow Graphs (CFG)

- The simplest unit of control flow in a program is a *basic block*

- **Basic block:** a maximal-length sequence of branch-free code.
  - It begins with a labelled operation and ends with a branch, jump, or predicated operation.

Control Flow Graphs

- A basic block is a sequence of operations that always execute together, unless an operation raises an exception.

- Control always enters a basic block at its first operation and exits at its last operation.

Control Flow Graphs

- A *control-flow graph* (cfg) models the flow of control between the basic blocks in a program.

- A cfg is a directed graph, $G = (N, E)$,

  Each node $n \in N$ corresponds to a basic block.
- Each edge $e = (n_i , n_j) \in E$ corresponds to a possible transfer of control from block n_i to block n_j .

Control Flow Graphs

- We assume that each CFG has a unique entry node, n_0, and a unique exit node, n_f .

- In the cfg for a procedure, n_0 corresponds to the procedure's entry point.
  - If a procedure has multiple entries, the compiler can insert a unique n_0 and add edges from n_0 to each actual entry point

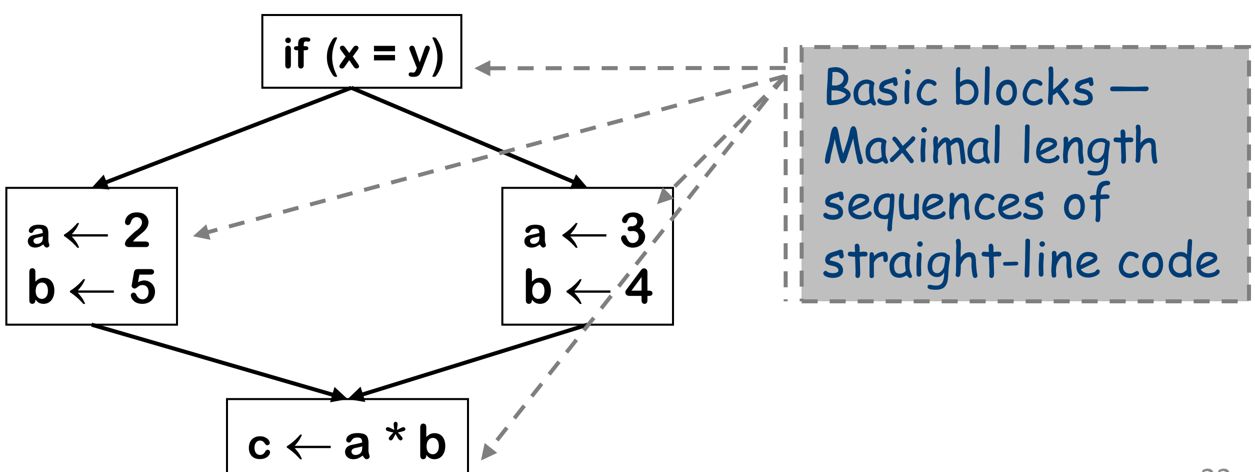
- Similarly, n_f corresponds to the procedure's exit.
  - Multiple exits are more common than multiple entries,
  - but the compiler can easily add a unique n_f and connect each of the actual exits to it.

Control-flow Graph

Models the transfer of control in the procedure

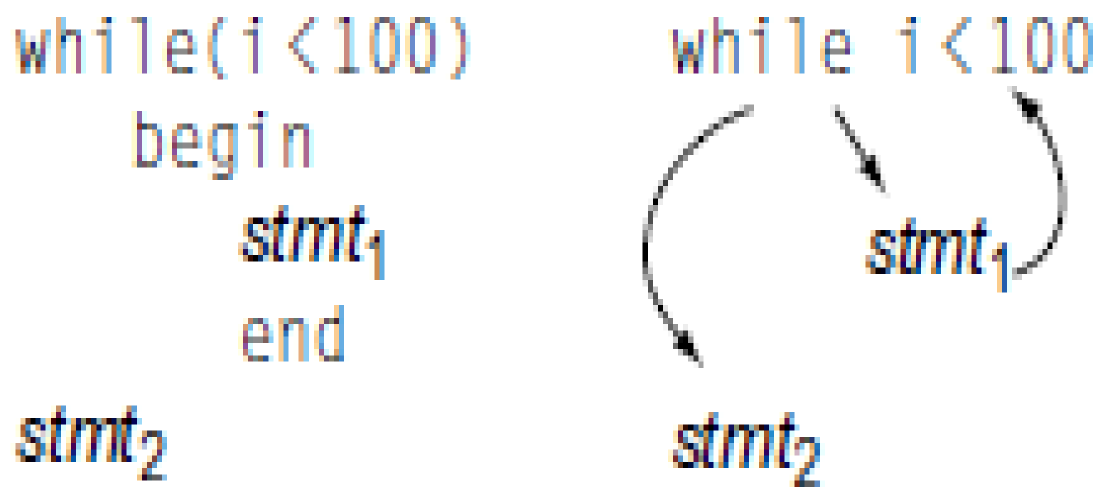
- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



Control Flow Graphs

- The cfg provides a graphical representation of the possible runtime control-flow paths.

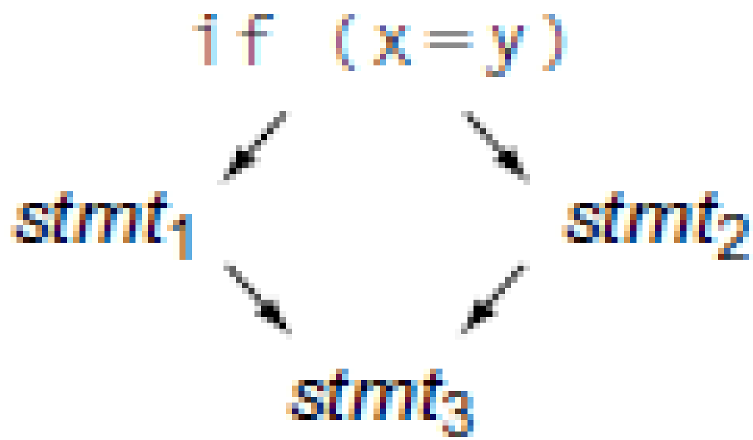


- The AST for this fragment would be acyclic.

Control Flow Graphs

- For an if-then-else construct, the cfg is acyclic:

```
if (x=y)
    then stmt1
    else stmt2
stmt3
```



- It shows that control always flows from *stmt*1 and *stmt*2 to *stmt*3.
- In an AST, that connection is implicit, rather than explicit.

CFG Example 1

```
X := 20; WHILE X < 10 DO
   X := X-1; A[X] := 10;
   IF X = 4 THEN X := X - 2; ENDIF;
ENDDO; Y := X + 5;
```

Write the sequence of statements corresponding to the code and convert to control flow graph.

CFG Example 1

```
X := 20; WHILE X < 10 DO
   X := X-1; A[X] := 10;
   IF X = 4 THEN X := X - 2; ENDIF;
ENDDO; Y := X + 5;
```

Sequence of statements corresponding to the above code

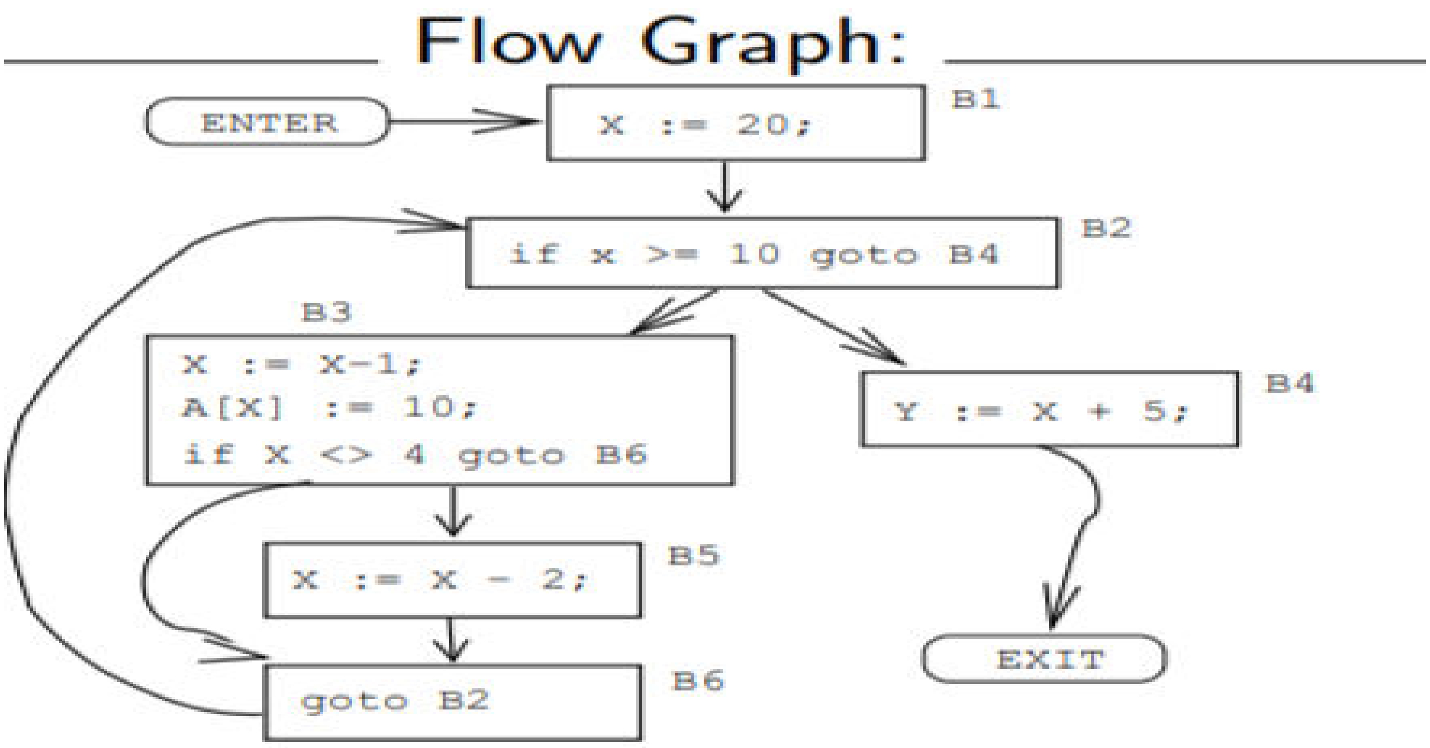
```
(1) X := 20               (5) if X<>4 goto (7)
(2) if X>=10 goto (8)     (6) X := X-2
(3) X := X-1              (7) goto (2)
(4) A[X] := 10            (8) Y := X+5
```

Note: Uses Branch on false condition in step (2), step (5) with respect to the given code

CFG Example 1

```
X := 20; WHILE X < 10 DO
   X := X-1; A[X] := 10;
   IF X = 4 THEN X := X - 2; ENDIF;
ENDDO; Y := X + 5;
```

```
(1) X := 20              (5) if X<>4 goto (7)
(2) if X>=10 goto (8)    (6) X := X-2
(3) X := X-1             (7) goto (2)
(4) A[X] := 10           (8) Y := X+5
```



CFG Example 2

```
P := 0; I := 1;
REPEAT
    P := P + I;
    IF P > 60 THEN
        P := 0;
        I := 5
    ENDIF;
    I := I * 2 + 1;
UNTIL I > 20;
K := P * 3
```

Control Flow Graphs in Hybrid IR

- Compilers typically use a CFG in conjunction with another IR.

- The cfg represents the relationships among blocks, while the operations inside a block are represented with another IR, such as an expression-level AST, a DAG, or one of the linear IRs.

- The resulting combination is a hybrid IR.

Single-statement blocks

- a block of code that corresponds to a single source-level statement

- The tradeoff between a cfg built with single-statement blocks and one built with basic blocks revolves around time and space.

Single-statement blocks

- cfg built on single statement blocks has more nodes and edges than a cfg built with basic blocks.

- The single-statement version uses more memory and takes longer to traverse than the basic-block version of a cfg.

- More important, as the compiler annotates the nodes and edges in the cfg, the single-statement cfg has many more sets than the basic-block cfg.

Importance of CFG

- Many parts of the compiler rely on a cfg, either explicitly or implicitly.
  - Analysis to support optimization generally begins with control-flow analysis and cfg construction (Chapter 9).
  - Instruction scheduling needs a cfg to understand how the scheduled code for individual blocks flows together (Chapter 12).
  - Global register allocation relies on a cfg to understand how often each operation might execute and where to insert loads and stores for spilled values (Chapter 13).

Dependence Graph

- **Data-dependence graph**
  - a graph that models the flow of values from definitions to uses in a code fragment.
  - Nodes in a data-dependence graph represent operations.
  - Most operations contain both definitions and uses.
  - An edge in a data-dependence graph connects two nodes, one that defines a value and another that uses it.
  - We draw dependence graphs with edges that run from definition to use.

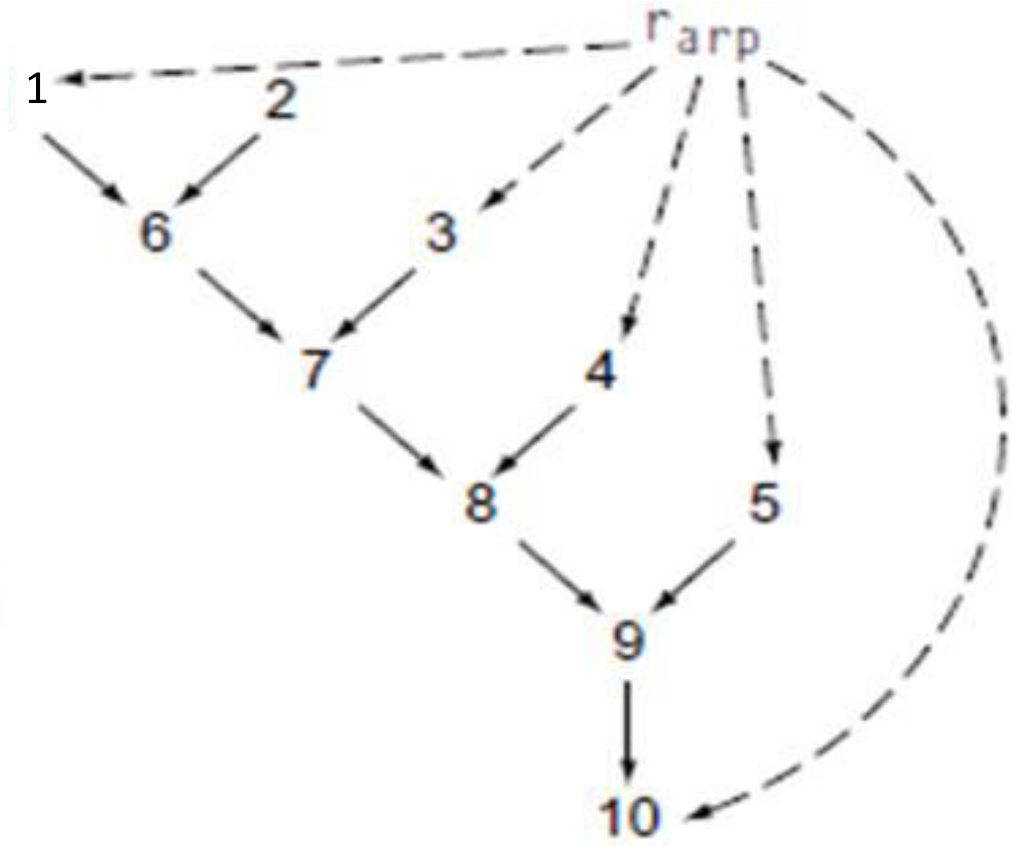
Dependence Graph

Expression
a = a * 2 * b * c * d

```
loadAI   r_arp, @a  ⇒ r_a
loadI    2          ⇒ r_2
loadAI   r_arp, @b  ⇒ r_b
loadAI   r_arp, @c  ⇒ r_c
loadAI   r_arp, @d  ⇒ r_d
mult     r_a, r_2   ⇒ r_a
mult     r_a, r_b   ⇒ r_a
mult     r_a, r_c   ⇒ r_a
mult     r_a, r_d   ⇒ r_a
storeAI  r_a        ⇒ r_arp, @a
```

Dependence Graph

1	loadAI	r_{arp}, @a	$\Rightarrow$	r_a
2	loadI	2	$\Rightarrow$	r_2
3	loadAI	r_{arp}, @b	$\Rightarrow$	r_b
4	loadAI	r_{arp}, @c	$\Rightarrow$	r_c
5	loadAI	r_{arp}, @d	$\Rightarrow$	r_d
6	mult	r_a, r_2	$\Rightarrow$	r_a
7	mult	r_a, r_b	$\Rightarrow$	r_a
8	mult	r_a, r_c	$\Rightarrow$	r_a
9	mult	r_a, r_d	$\Rightarrow$	r_a
10	storeAI	r_a	$\Rightarrow$	r_{arp}, @a



Expression

a = a * 2 * b * c * d

- Uses of r_{arp} refer to its implicit definition at the start of the procedure; they are shown with dashed lines.

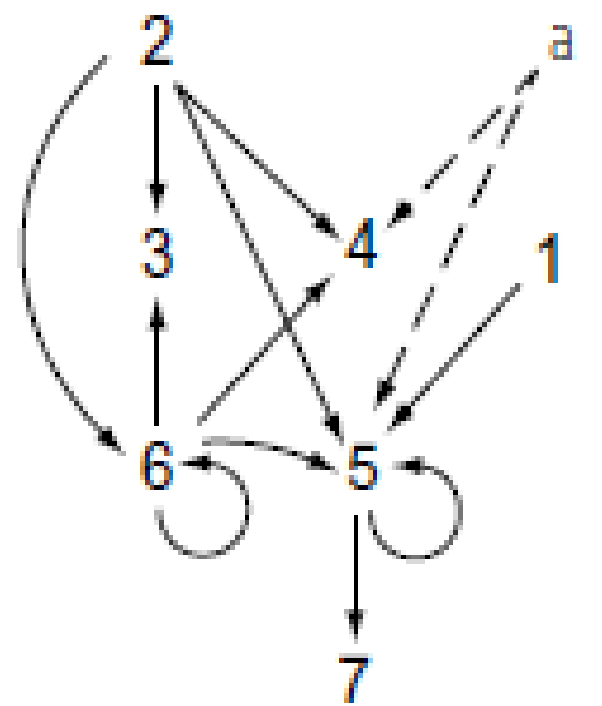
Dependence Graph

```
x ← 0
i ← 1
while (i < 100)
   if (a[i] > 0)
      then x ← x + a[i]
   i ← i + 1
print x
```

Draw the dependence graph for the above code.

Dependence Graph

```
x ← 0
i ← 1
while (i < 100)
   if (a[i] > 0)
      then x ← x + a[i]
   i ← i + 1
print x
```



Dependence Graph

- Data-dependence graphs are often used as a derivative IR—constructed from the definitive IR for a specific task, used, and then discarded.

- They play a central role in instruction scheduling (Chapter 12).

- They find application in a variety of optimizations, particularly transformations that reorder loops to expose parallelism and to improve memory behavior; these typically require sophisticated analysis of array subscripts to determine more precisely the patterns of access to arrays.

- In more sophisticated applications of the data dependence graph, the compiler may perform extensive analysis of array subscript values to determine when references to the same array can overlap.

Call Graph

- **Call graph**
  - a graph that represents the calling relationships among the procedures in a program
  - The call graph has a node for each procedure and an edge for each call site.
  - Thus, the code calls *q* from three textually distinct sites in *p*; the call graph has three edges (*p*, *q*), one for each call site.

Interprocedural: Any technique that examines interactions across multiple procedures.

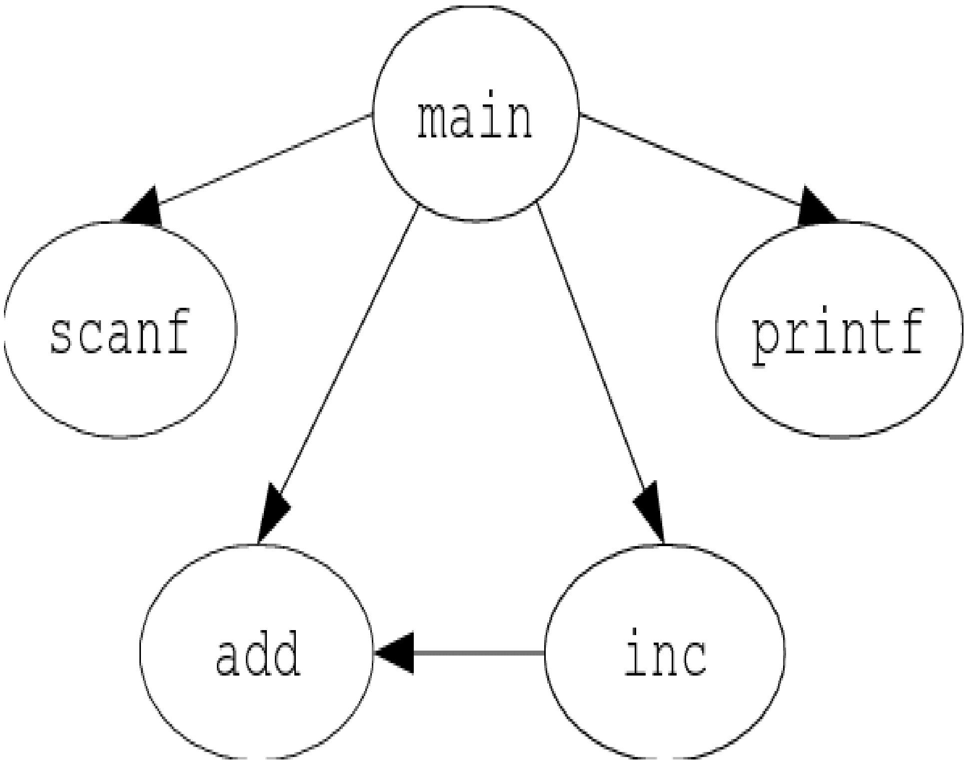
Intraprocedural: Any technique that limits its attention to a single procedure.

Call Graph example

```
main(){
…
printf(…);
scanf(…);
…
add(..);
…
inc(..);
…
}
```

```
add(…){
…}
```

```
inc(…){
…
add(…);
…}
```



Call Graph Challenges

Both software-engineering practice and language features complicate the construction of a call graph.

- **Separate compilation**:
  - the practice of compiling small subsets of a program independently, limits the compiler's ability to build a call graph and to perform inter-procedural analysis and optimization.
  - Some compilers build partial call graphs for all of the procedures in a compilation unit and perform analysis and optimization across that set.
  - To analyze and optimize the whole program in such a system, the programmer must present it all to the compiler at once.

Call Graph Challenges

- **Procedure-valued parameters**:
  - both as input parameters and as return values, complicate call-graph construction by introducing ambiguous call sites.
  - If **fee** takes a procedure-valued argument and invokes it, that site has the potential to call a different procedure on each invocation of **fee**.
  - The compiler must perform an inter-procedural analysis to limit the set of edges that such a call induces in the call graph.

Call Graph Challenges

- **Object-oriented programs:**
  - with inheritance routinely create ambiguous procedure calls that can only be resolved with additional type information.
  - In some languages, inter-procedural analysis of the class hierarchy can provide the information needed to disambiguate these calls.
  - In other languages, that information cannot be known until runtime.
  - Runtime resolution of ambiguous calls poses a serious problem for call graph construction;
  - it also creates significant runtime overheads on the execution of the ambiguous calls.

Linear IRS

- An assembly language program is a form of linear code.
- It consists of sequence of instructions that execute in their order of appearance.
- Control flow in a linear IR usually transfer control to the target machine – usually include conditional branches and jumps.

Linear IRS

- **Taken Branch –** control flows either to the label.
- **Fall-through branch (not-taken) –** to the operation that follows the label.
- The basic blocks of CFG in a linear IR – blocks ***end at branches***, *at* ***jumps*** or ***just before labeled operation***.

Types of Linear IRS

- One-address codes:
  - Models the behavior of accumulator machines and stack machines.
  - Code is compact.
- Two-address codes: (less important)
  - A machine that has destructive operation
  - Disuse of memory
- Three-address codes:
  - Models a machine where most operations take two operands and produce a result.
  - Simple RISC architecture

Stack Machine Code

Originally used for stack-based computers, now Java

- Example:

  x - 2 * y becomes

  push x
  push 2
  push y
  multiply
  subtract

Implicit names take up no space, where explicit ones do!

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Useful where code is transmitted
over slow communication links (*the net*)

Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:

$\mathbf{x \leftarrow y}$ ***op*** $\mathbf{z}$

With 1 operator (*op*) and, at most, 3 names (x, y, & z)

$t \leftarrow 2 * y$
$z \leftarrow x - t$

Example:

$z \leftarrow x - 2 * y$

becomes

$t \leftarrow 2 * y$
$z \leftarrow x - t$

*

Advantages:

- Resembles many real machines
- Introduces a new set of names
- Compact form

Three Address Code: Quadruples

Naïve representation of three address code

- Table of k * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

RISC assembly code

load	1	y	
loadi	2	2	
Mult	3	2	1
load	4	x	
sub	5	4	3

Quadruples

Expression is x - 2 * y

Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	loadI	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Implicit names occupy no space

Three Address Code: Indirect Triples

- List first triple in each statement
- Implicit name space
- Uses more space than triples, but easier to reorder

Stmt List	Implicit Names	Indirect Triples		
(100)	(100)	load	y	
(105)	(101)	loadI	2	
	(102)	mult	(100)	(101)
	(103)	load	x	
	(104)	sub	(103)	(102)

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  - In the past compile-time space was critical
  - Today, speed may be more important

Two Address Code

- Allows statements of the form

  $x \leftarrow x\ \underline{op}\ y$

  Has 1 operator ($\underline{op}$) and, at most, 2 names (x and y)

Example:

$z \leftarrow x - 2 * y$ becomes

$$t_1 \leftarrow 2$$
$$t_2 \leftarrow \text{load } y$$
$$t_2 \leftarrow t_2 * t_1$$
$$z \leftarrow \text{load } x$$
$$z \leftarrow z - t_2$$

- Can be very compact

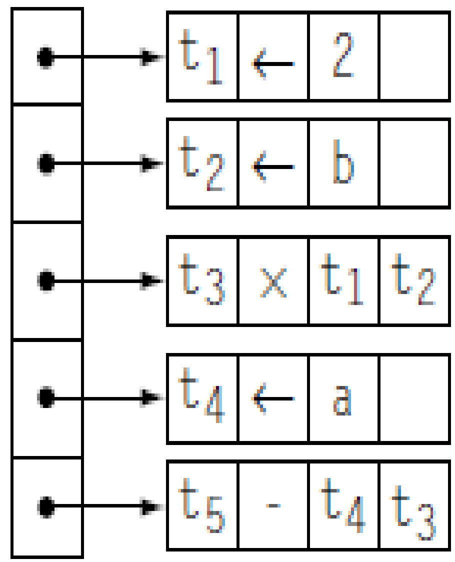
Problems

- Machines no longer rely on destructive operations
- Difficult name space
  - Destructive operations make reuse hard
  - Good model for machines with destructive ops (PDP-11)

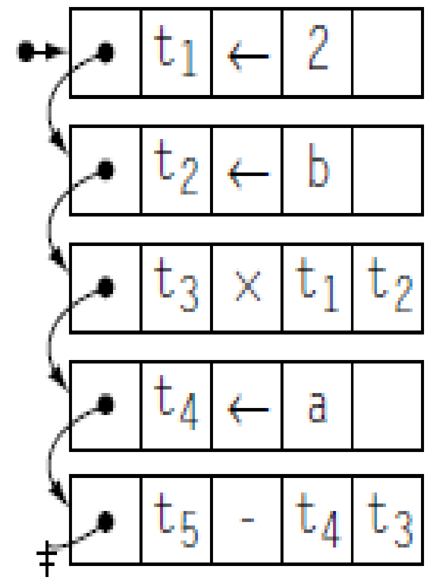
Implementations of Three-Address Code for $a - 2 \times b$

Target	*Op*	*Arg*$_1$	*Arg*$_2$
t_1	$\leftarrow$	2	
t_2	$\leftarrow$	b	
t_3	$\times$	t_1	t_2
t_4	$\leftarrow$	a	
t_5	-	t_4	t_3

(a) Simple Array



(b) Array of Pointers



(c) Linked List

Cost in rearranging

- The first operation loads a constant into a register; on most machines this translates directly into an immediate load operation.

- The second and fourth operations load values from memory, which on most machines might incur a multicycle delay unless the values are already in the primary cache.

- To hide some of the delay, the instruction scheduler might move the loads of b and a in front of the immediate load of 2.

- Lets analyze how it is possible in three different schemes of implementation.

Cost in Simple Array

- In the simple array scheme, moving the load of b ahead of the immediate load requires
  - saving the four fields of the first operation,
  - copying the corresponding fields from the second slot into the first slot, and
  - Overwriting the fields in the second slot with the saved values for the immediate load.

Cost in Array of Pointers

- The array of pointers requires the same three-step approach, except
  - that only the pointer values must be changed.

- Thus, the compiler
  - saves the pointer to the immediate load,
  - copies the pointer to the load of b into the first slot in the array, and
  - overwrites the second slot in the array with the saved pointer to the immediate load.

Cost in Linked List

- For the linked list, the operations are similar, except that the complier must save enough state to let it traverse the list.

Review Question

- Consider the expression $a \times 2 + a \times 2 \times b$. Translate it into stack machine code and into three address code.
- Compare and contrast the number of operations and the number of operands in each form.

Static Single Assignment Form (SSA)

SSA is used to encode information about both flow of control and flow of data values in program

- Main idea: each name defined exactly once introduce ϕ-functions to make it work. ϕ-function behavior depends on context.

Original

$x \leftarrow \ldots$
$y \leftarrow \ldots$
while $(x < k)$
  $x \leftarrow x + 1$
  $y \leftarrow y + x$

SSA-form

$x_0 \leftarrow \ldots$
$y_0 \leftarrow \ldots$
if $(x_0 >= k)$ goto next
loop: $x_1 \leftarrow \phi(x_0,x_2)$
  $y_1 \leftarrow \phi(y_0,y_2)$
$x_2 \leftarrow x_1 + 1$
$y_2 \leftarrow y_1 + x_2$
if $(x_2 < k)$ goto loop
next: …

Strengths of SSA-form

- Sharper analysis
- ϕ-functions give hints about placement
- (sometimes) faster algorithms

Constraints:
1)Each definition name has distinct name
2) Each use refers to a single definition

Problems

1. Convert the following to SSA

Original	SSA
$a := b + c$	$a_1 := b_1 + c_1$
$b := c + 1$	$b_2 := c_1 + 1$
$d := b + c$	$d_1 := b_2 + c_1$
$a := a + 1$	$a_2 := a_1 + 1$
$e := a + b$	$e_1 := a_2 + b_2$

Problems

2. Convert the following to SSA

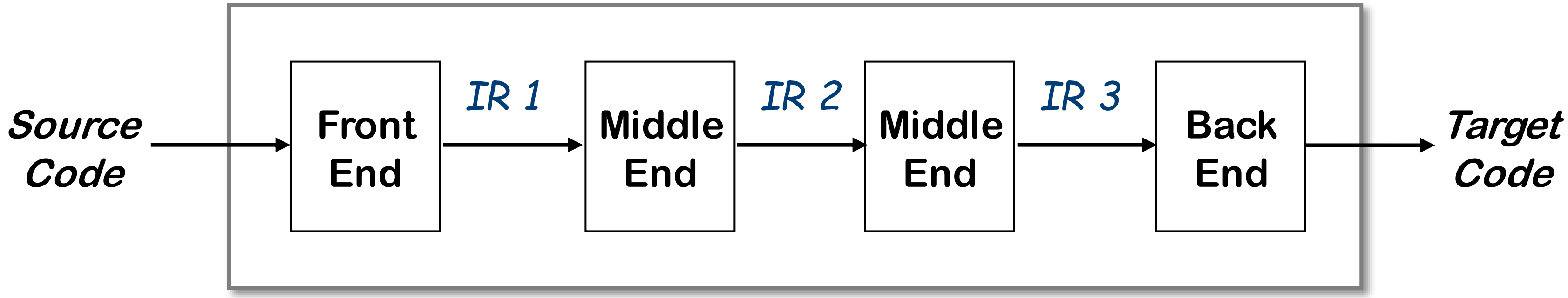
Original

```
if B then
  a := b
else
  a := c
end
  … a …
```

SSA

if B then
  a_1 := b
else
  a_2 := c
End
$a_3 := \Phi(a_1, a_2)$
  … a_3 …

Using Multiple Representations



- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - WHIRL intermediate format
    - Consists of 5 different *IR*s that are progressively more detailed and less abstract

Memory Models

Two major models

- Register-to-register model
  - Keep all values that can legally be stored in a register in registers
  - Ignore machine limitations on number of registers
  - Compiler back-end must insert loads and stores

- Memory-to-memory model
  - Keep all values in memory
  - Only promote values to registers directly before they are used
  - Compiler back-end can remove loads and stores

- Compilers for RISC machines usually use register-to-register
  - Reflects programming model
  - Easier to determine when registers are used

The Rest of the Story…

Representing the code is only part of an *IR*

There are other necessary components

- Symbol table
- Constant table
  - Representation, type
  - Storage class, offset
- Storage map
  - Overall storage layout
  - Overlap information
  - Virtual register assignments

Intermediate Representations Problems

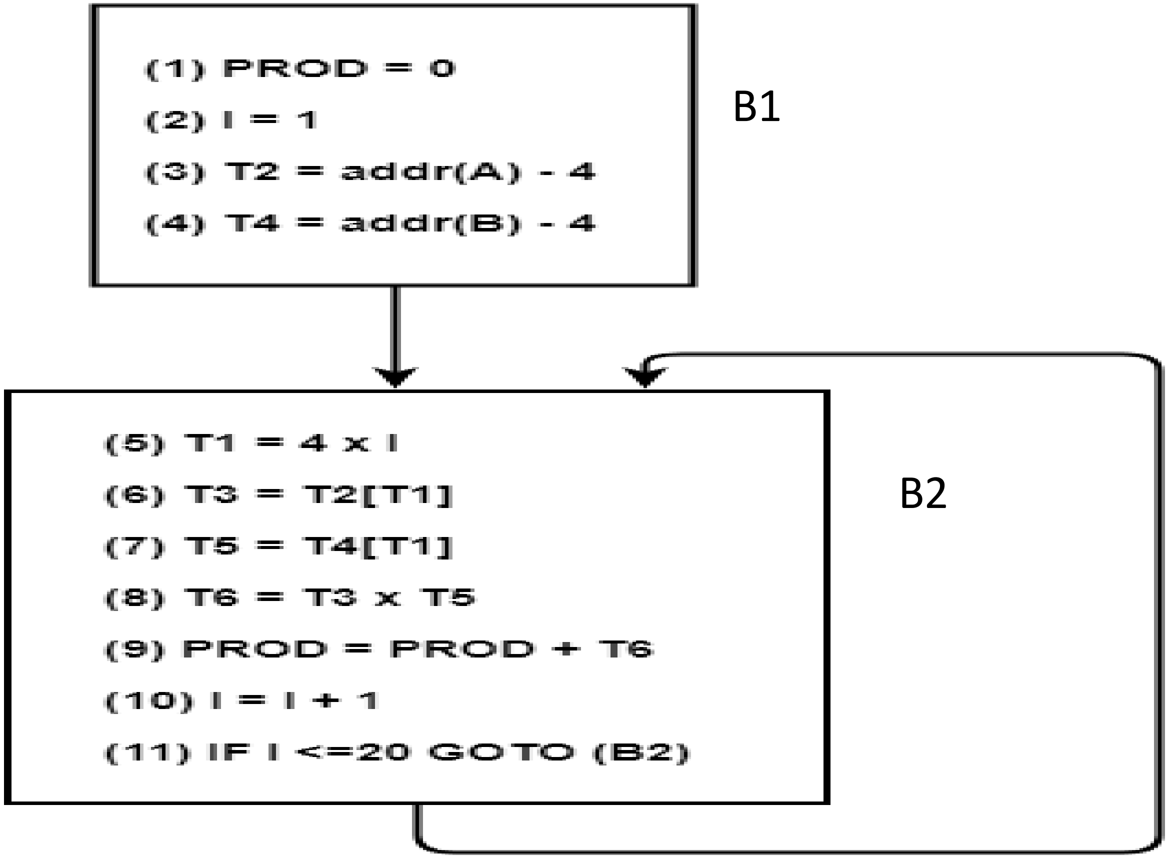
Chapter-5

Amrita School of Engineering, Bengaluru
Amrita Vishwa Vidyapeetham

Problems

1. Convert the following to CFG

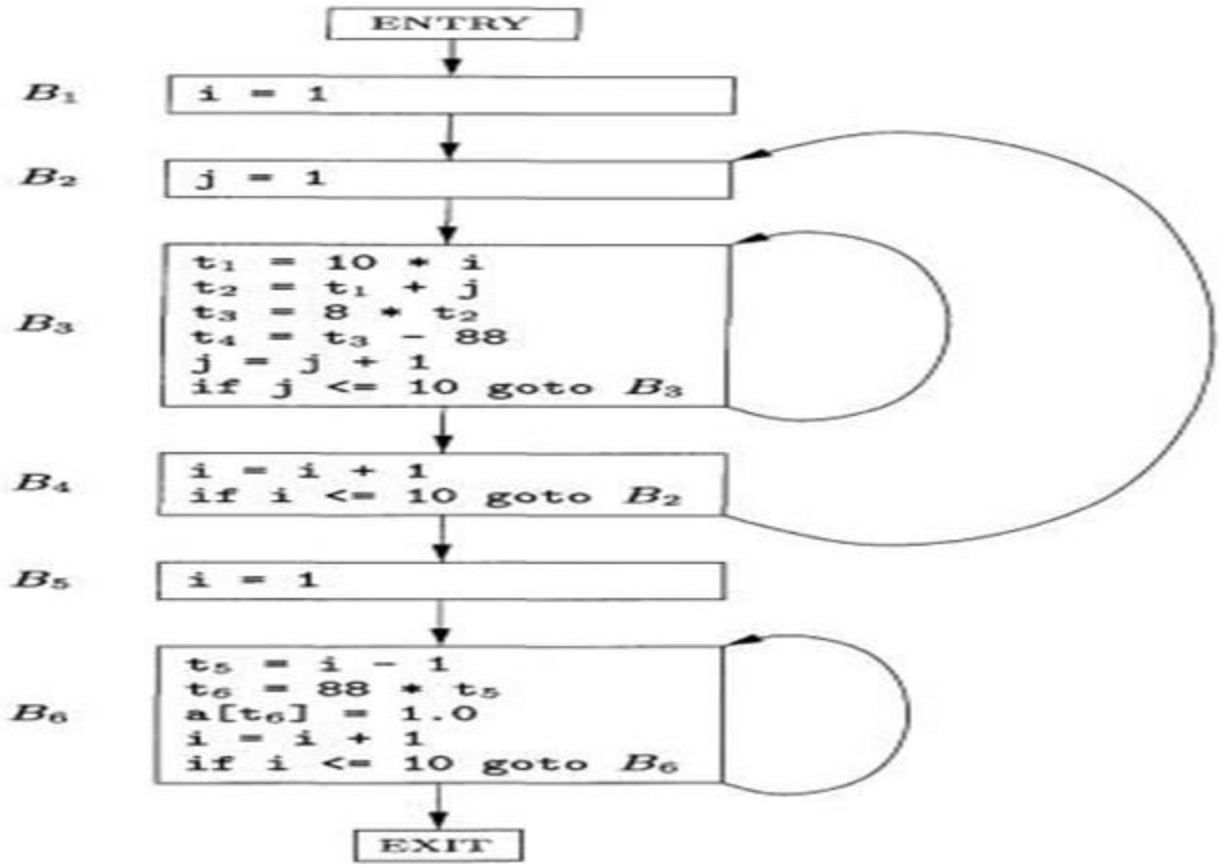
(1) PROD = 0
(2) I = 1
(3) T2 = addr(A) – 4
(4) T4 = addr(B) – 4
(5) T1 = 4 x I
(6) T3 = T2[T1]
(7) T5 = T4[T1]
(8) T6 = T3 x T5
(9) PROD = PROD + T6
(10) I = I + 1
(11) IF I <=20 GOTO (5)



Problems

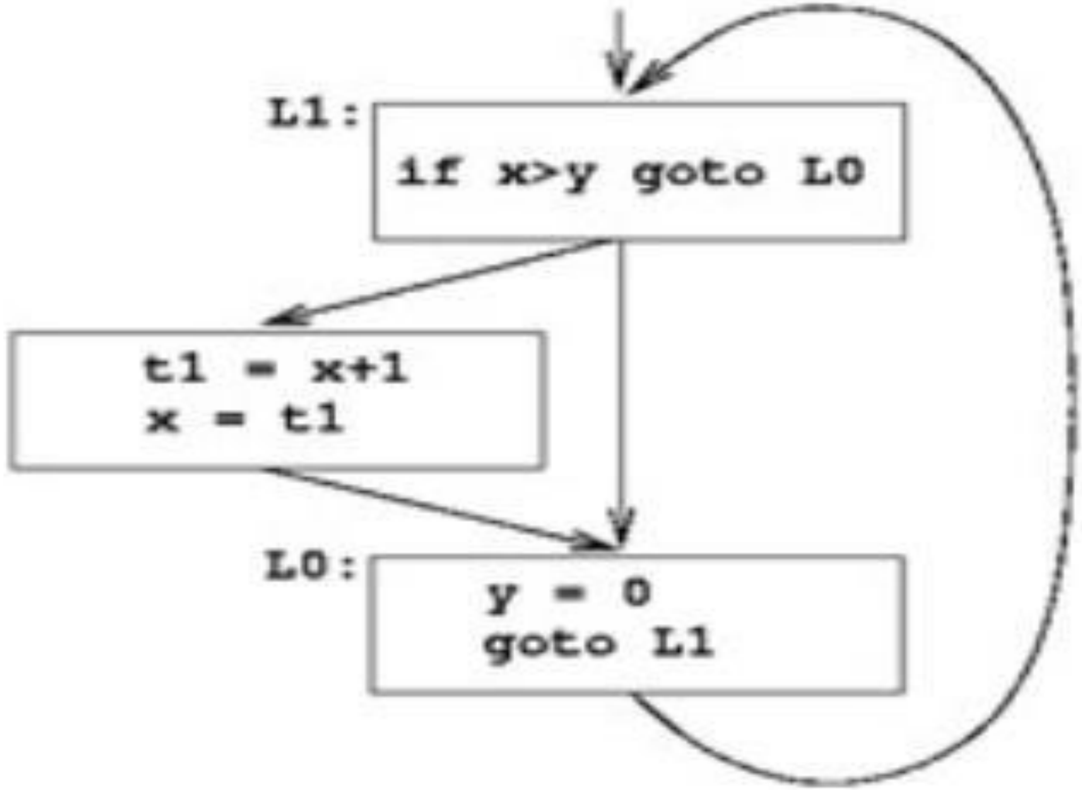
2. Convert the following to CFG

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



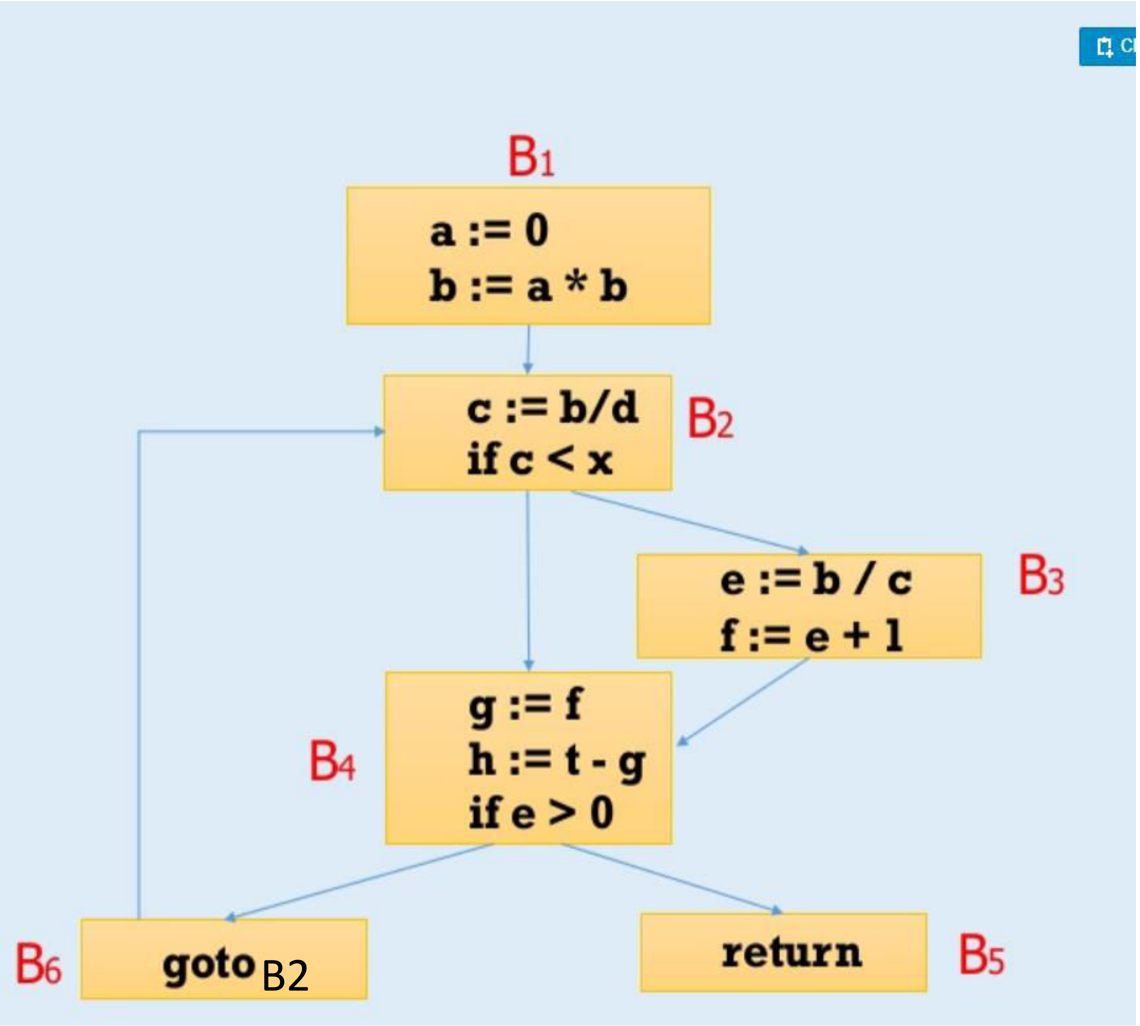
3. Convert the following to CFG

```
L1: if x > y goto L0
    t1 = x+1
    x = t1
L0: y = 0
    goto L1
```



4 Convert the following to CFG

```
    a := 0
    b := a * b
L1: c := b/d
    if c < x goto L2
    e := b / c
    f := e + 1
L2: g := f
    h := t - g
    if e > 0 goto L3
goto L1
L3: return
```



5. Convert the following to CFG

```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

Problems

Convert the following to 3-address code

1.
```
for(i = 1; i<=10; i++)
 {
  a[i] = x * 5;
 }
```

2.
```
for(i = 0; i<10 ; i++){
      b[i] = i*i;
 }
```

3.
```
int a[10], b[10], dot_prod, i;

dot_prod = 0;

for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Source Code

4

```
        ⋮
y = p(a, b+1)
        ⋮
int p(x,z){
  return x+z;
}
```

Problems

1. Convert the following stmt to 3-address code

```
for(i = 1; i<=10; i++)
 {
  a[i] = x * 5;
 }
```

3-address code

```
        i=1
L2 :  If i>10 goto L1
        t1=x  * 5
        a[i]=t1
        t2=i+1
        i=t2
        goto L2
L1:
```

Problems

2. Convert the following stmt to 3-address code

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

3-address code

```
i=0
L2 :  If  i>=10 goto L1
        t1=i*i
        b[i]=t1
        t2=i+1
        i=t2
        goto L2
L1:
```

Problems

3. Convert the following 3-address code

```
int a[10],b[10],i,dot_prod=0
for(i=0;i<10;i++)
        dot_prod+=a[i]*b[i]
```

3-address code

```
        dot_prod=0
         i=0
L2 :   If  i>=10 goto L1
          t1=a[i]
          t2=b[i]
          t3=t1 * t2
          t4= dot_prod + t3
          dot_prod=t4
           i = i+1
          goto L2
L1:
```

Problems

4. Convert the following to 3-address code

Source Code

```
:
y = p(a, b+1)
:
int p(x,z){
  return x+z;
}
```

3-address code

```
.........
t1=a
t2=b+1
call p, t1, t2
.........
funcbegin p
    param x
    param z
    t3=x+z
    return t3
funcend
```

Problems

1. Convert the following to SSA

Original	SSA
a := b + c	a_1 := b_1 + c_1
b := c + 1	b_2 := c_1 + 1
d := b + c	d_1 := b_2 + c_1
a := a + 1	a_2 := a_1 + 1
e := a + b	e_1 := a_2 + b_2

Problems

2. Convert the following to SSA

Original

```
if B then
  a := b
else
  a := c
end
  … a …
```

SSA

if B then

  $a_1 := b$

else

  $a_2 := c$

End

$a_3 := \Phi(a_1, a_2)$

  … a_3 …

Problems

3. Convert the following to SSA

$a = b - c$

$d = a + d$

$a = d + e$

$d = c * f$

$d = a * d$

SSA

$a1 = b1 - c1$

$d2 = a1 + d1$

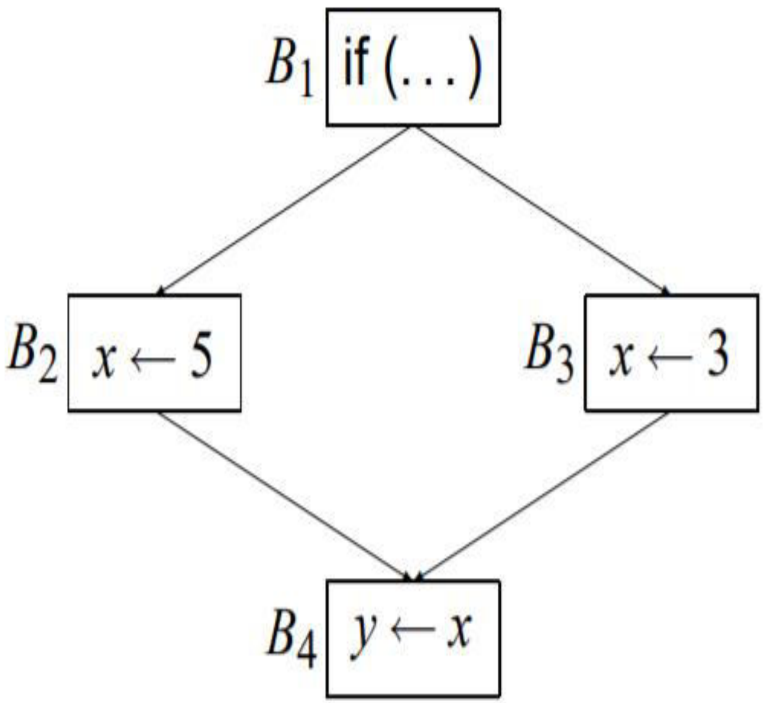
$a2 = d2 + e1$

$d3 = c1 * f1$

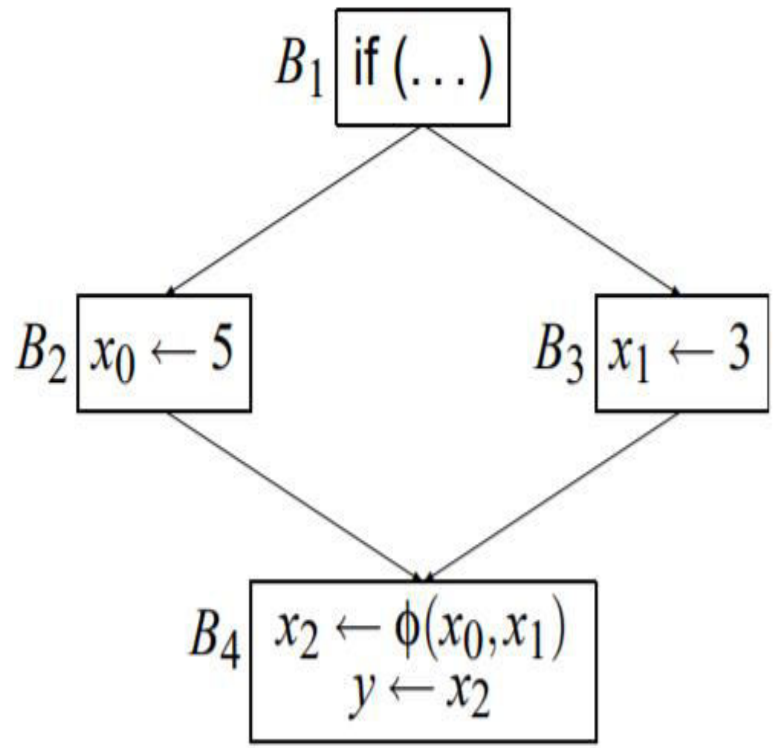
$d4 = a2 * d3$

Problems

4. Convert the following to SSA

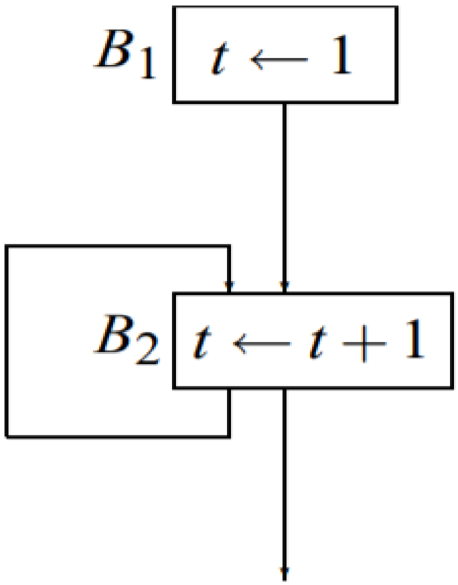


SSA

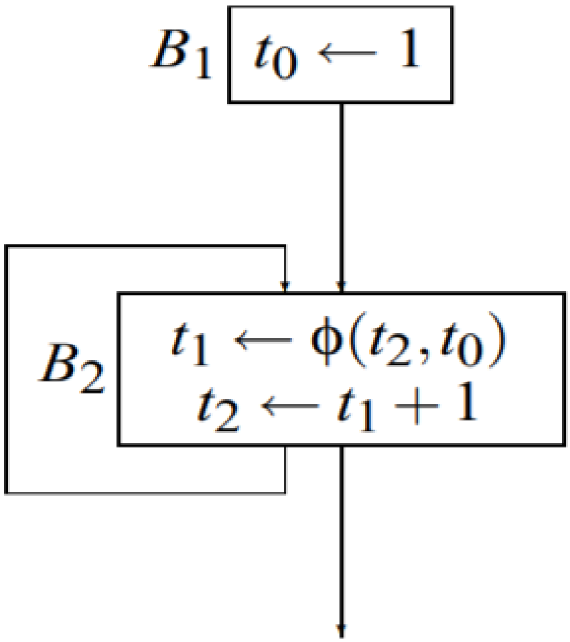


Problems

5. Convert the following to SSA



SSA

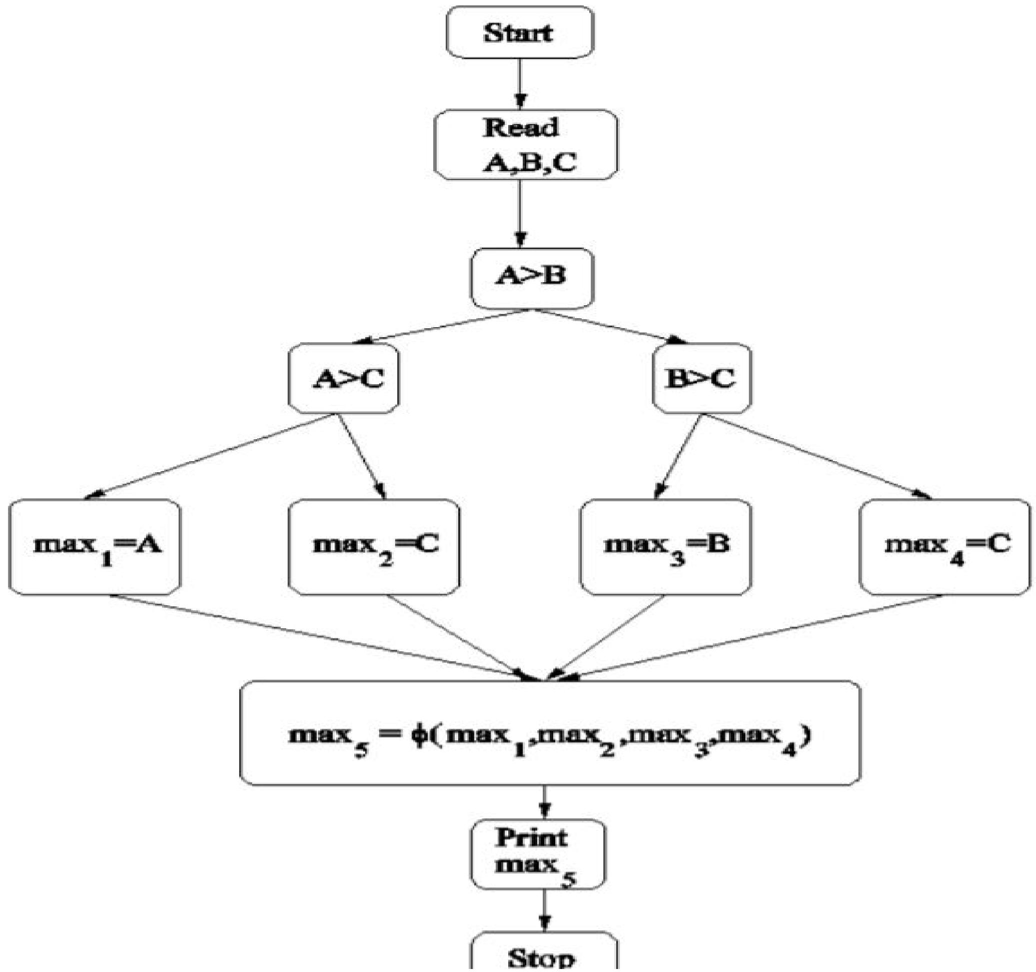


Problems

6. Convert the following to SSA

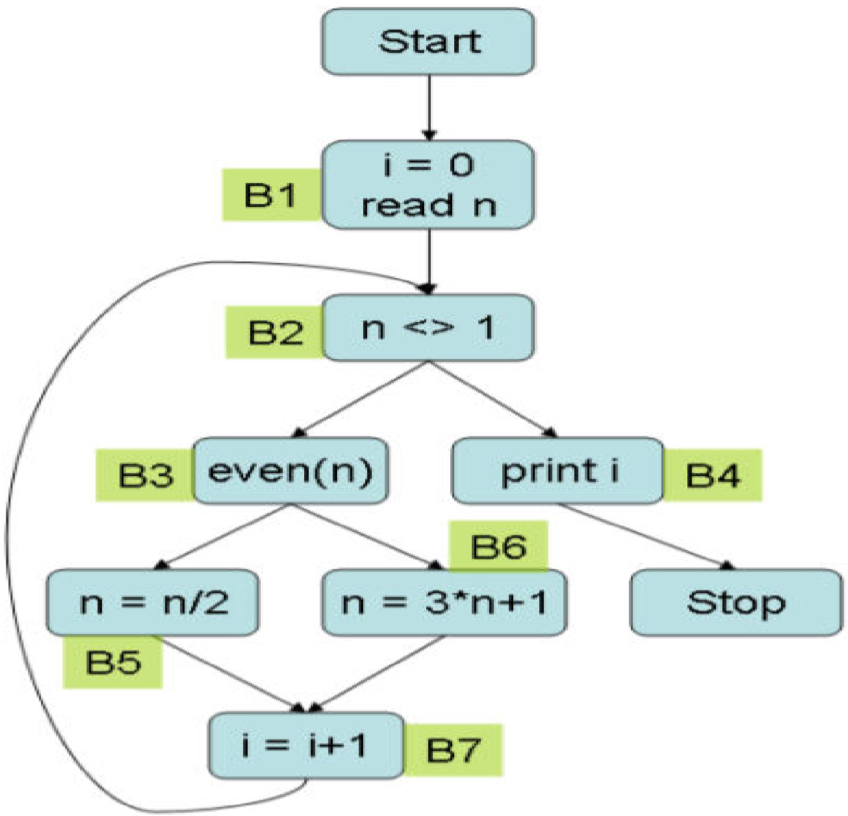
```
read A,B,C
if (A>B)
   if (A>C) max = A
   else max = C
else if (B>C) max = B
      else max = C
printf (max)
```

SSA

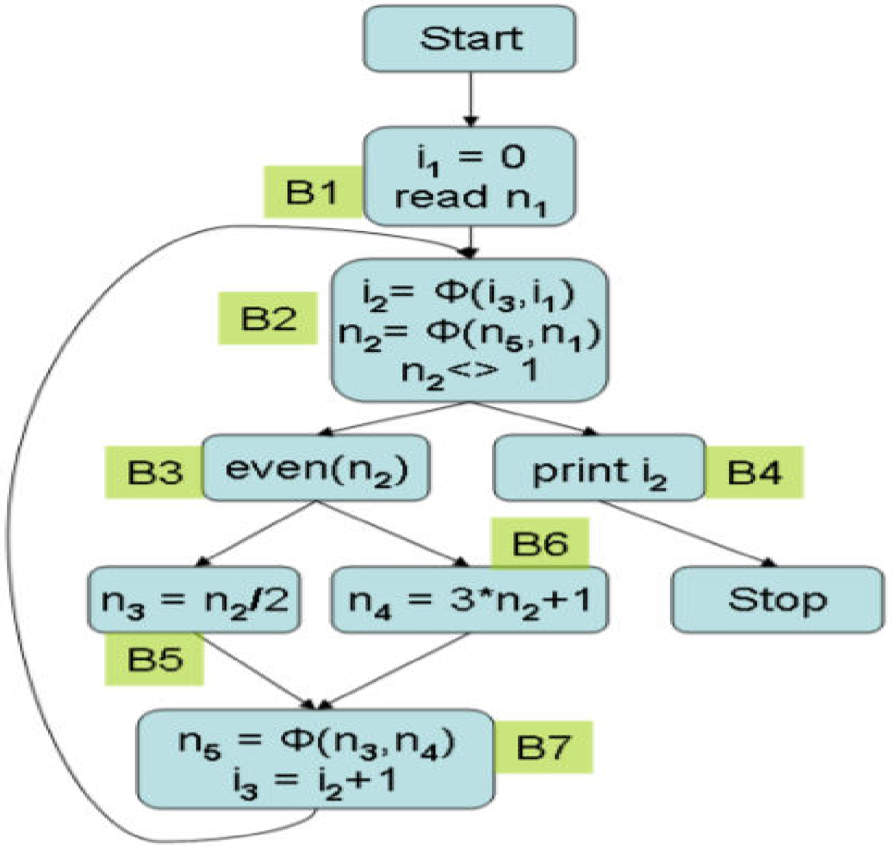


Problems

7. Convert the following to SSA



SSA

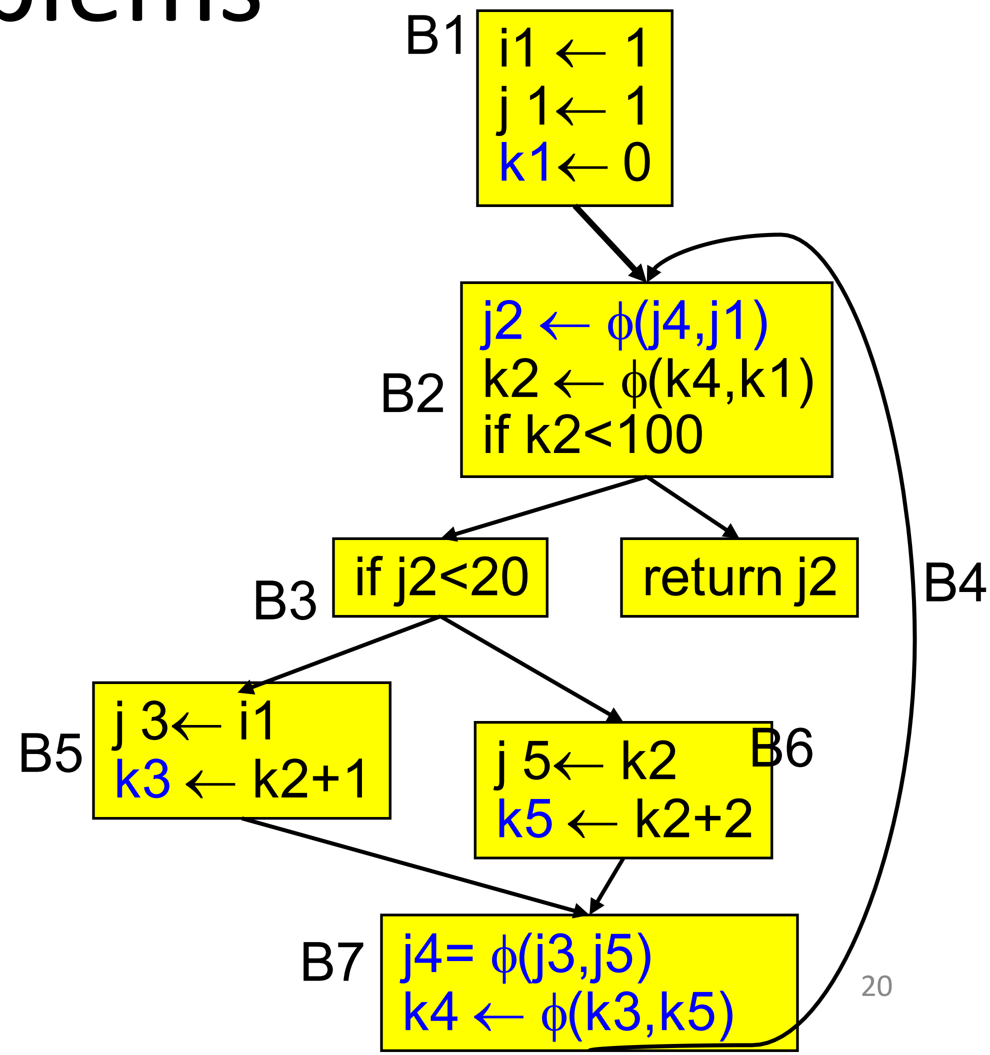


Problems

SSA

8. Convert the following to SSA

```
i=1;
j=1;
k=0;
 while(k<100)
        {
          if(j<20)
            {
              j=i;
              k=k+1;
            }
          else
            {
              j=k;
              k=k+2;
            }
        }
  return j;
}
```

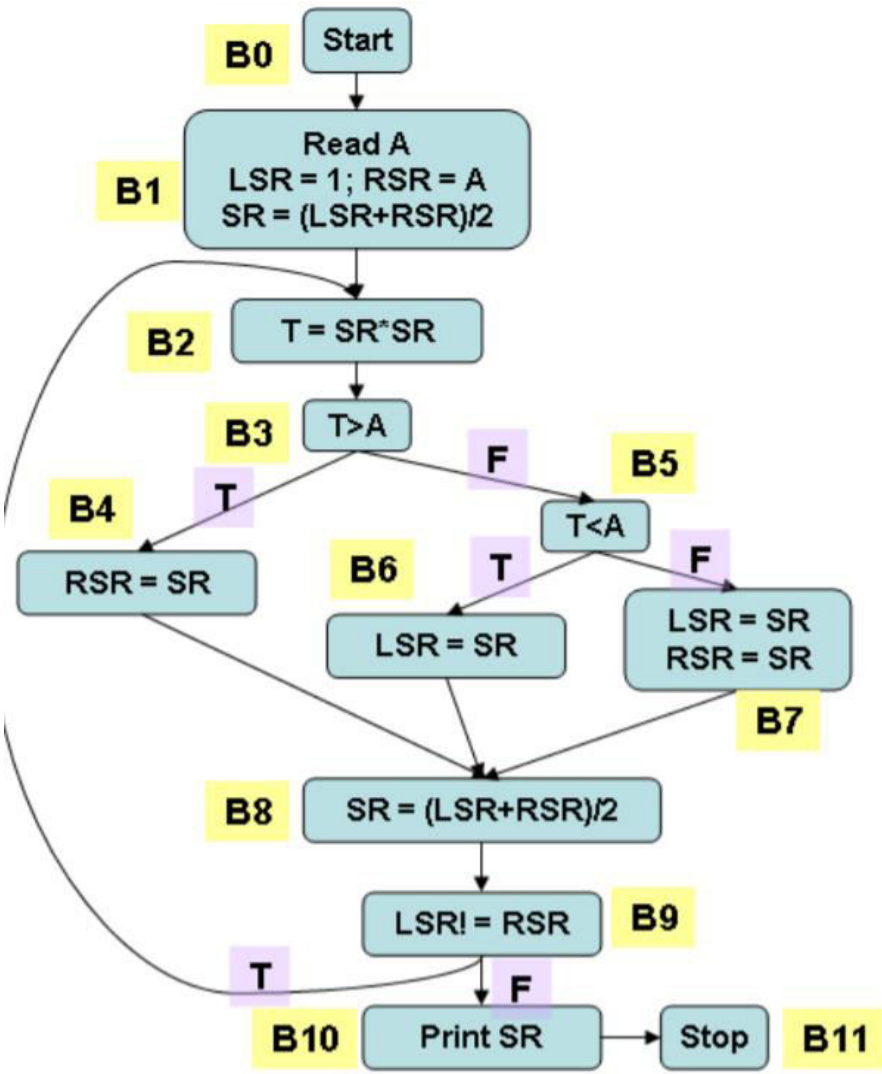


Problems

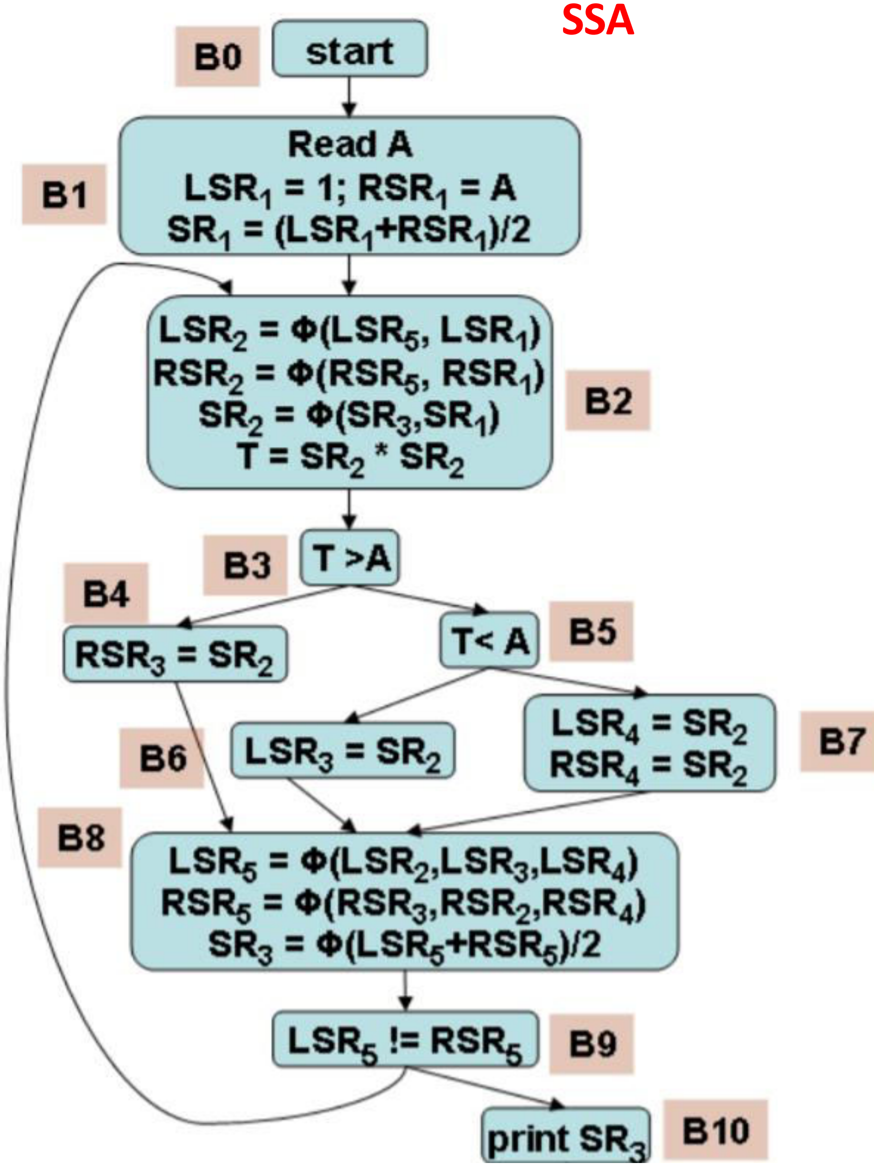
9. Convert the following to SSA

```
{ Read A; LSR = 1; RSR = A;
  SR = (LSR+RSR)/2;
  Repeat {
        T = SR*SR;
        if (T>A) RSR = SR;
        else if (T<A) LSR = SR;
              else { LSR = SR; RSR = SR}
        SR = (LSR+RSR)/2;
  Until (LSR ≠ RSR);
  Print SR;
}
```

CFG


B0
Start
B1
Read A
LSR = 1; RSR = A
SR = (LSR+RSR)/2
B2
T = SR*SR
B3
T>A
T
F
B4
RSR = SR
B5
T<A
T
F
B6
LSR = SR
B7
LSR = SR
RSR = SR
B8
SR = (LSR+RSR)/2
B9
LSR! = RSR
T
F
B10
Print SR
Stop
B11

SSA


B0
start
B1
Read A
LSR_1 = 1; RSR_1 = A
SR_1 = (LSR_1+RSR_1)/2
LSR_2 = Φ(LSR_5, LSR_1)
RSR_2 = Φ(RSR_5, RSR_1)
SR_2 = Φ(SR_3,SR_1)
T = SR_2 * SR_2
B2
B3
T >A
B4
RSR_3 = SR_2
T< A
B5
B6
LSR_3 = SR_2
LSR_4 = SR_2
RSR_4 = SR_2
B7
B8
LSR_5 = Φ(LSR_2,LSR_3,LSR_4)
RSR_5 = Φ(RSR_3,RSR_2,RSR_4)
SR_3 = Φ(LSR_5+RSR_5)/2
LSR_5 != RSR_5
B9
print SR_3
B10