COMPILER DESIGN

UNIT-4

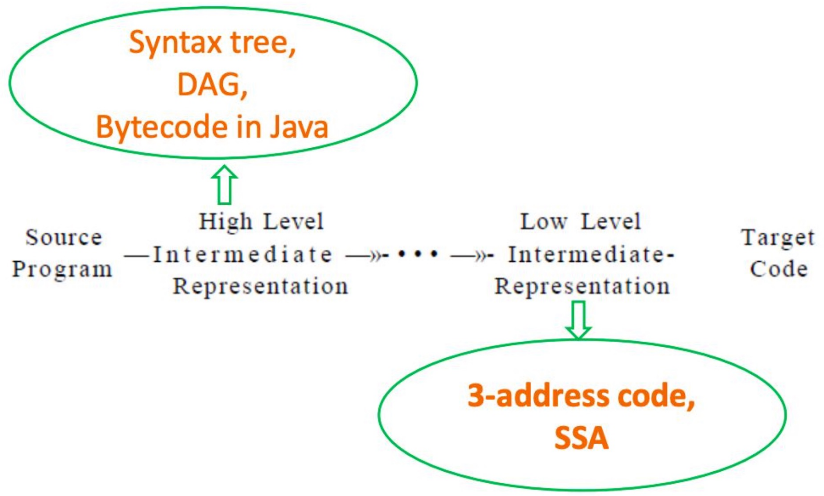
Intermediate Code Generation

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

Intermediate Code

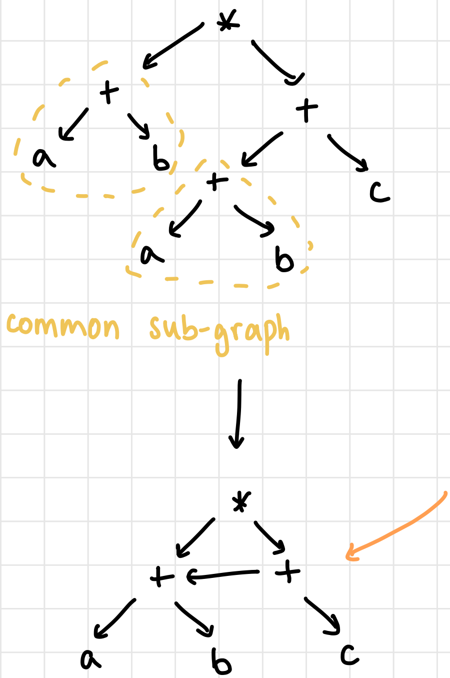
- Translate source → intermediate → machine
- Input: annotated syntax tree
- Ways to classify IR
  1. High level or low level
  2. Language specific or independent
  3. Graphical or linear



Directed Acyclic Graph

- No cycles
- Reuse sub-expression trees
- Exterior nodes: names / identifiers / constants
- Interior nodes: operators

Q: Draw a DAG for (a+b) * (a+b+c)



if node exists
    return existing node
else
    create new node

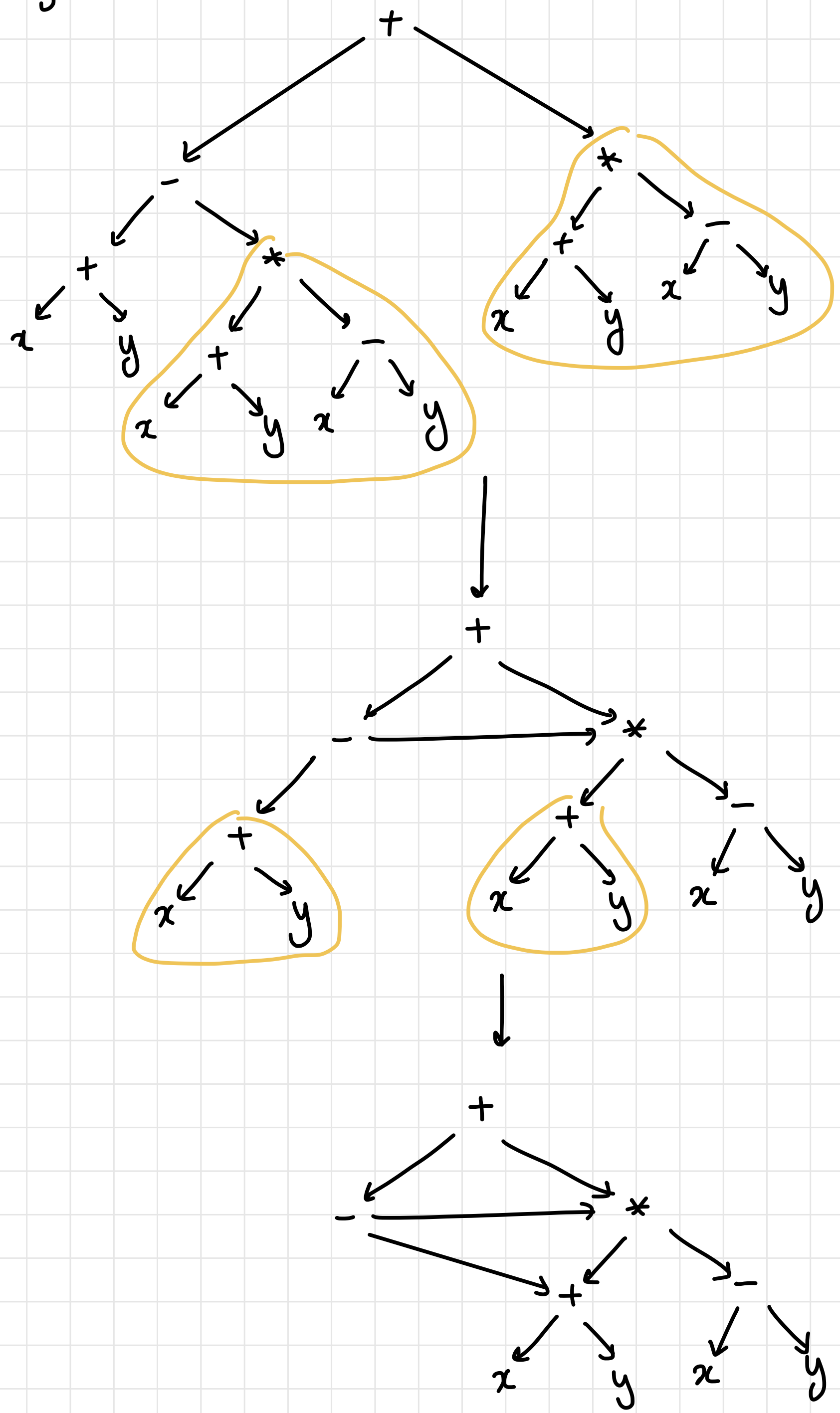
Q: Using grammar below, construct DAG for the given expression

$E \rightarrow E+T \mid E-T \mid T$
$T \rightarrow T*F \mid T/F \mid F$
$F \rightarrow (E) \mid [E] \mid id$

Expression:

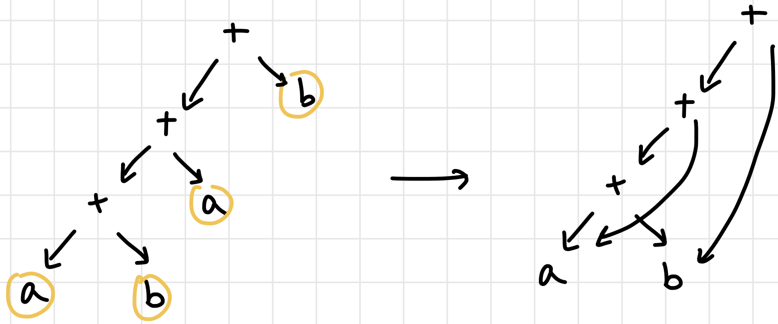
$((x+y)-((x+y)*(x-y))) + ((x+y)*(x-y))$

Syntax tree



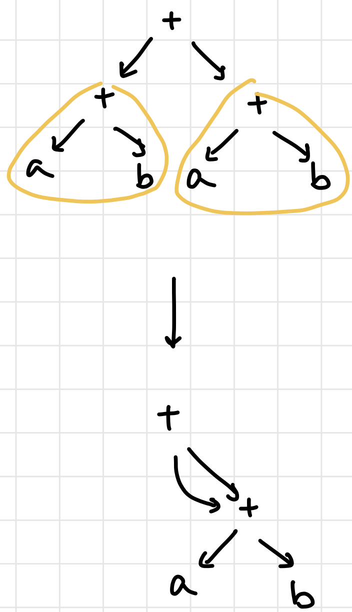
Q: For same grammar, construct DAG for

a+b+a+b



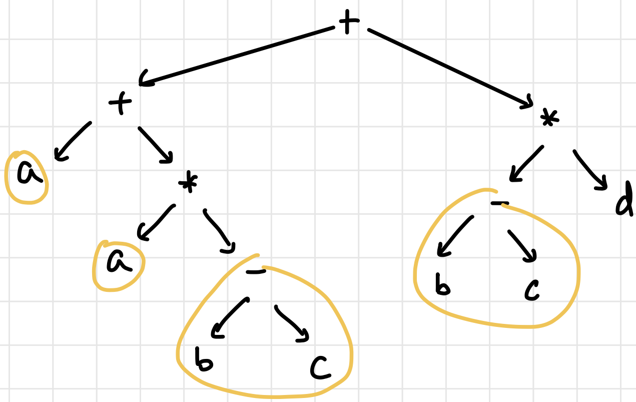
Q: For same grammar, construct DAG for

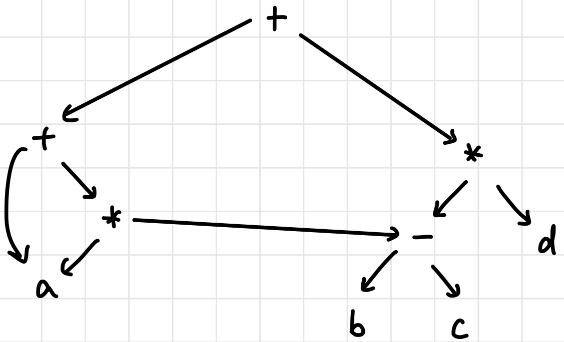
a + b + (a+b)



Q: For same grammar, construct DAG for

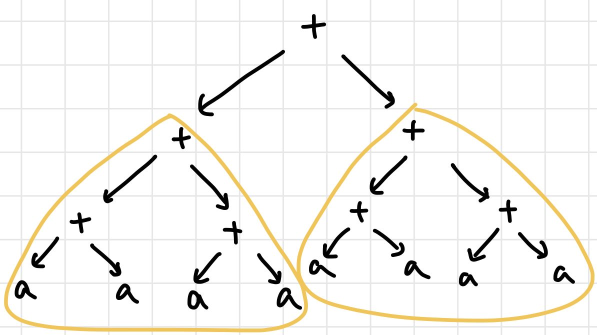
$$a + a * (b - c) + (b - c) * d$$

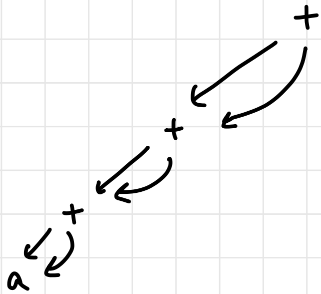




Q: For same grammar, construct DAG for

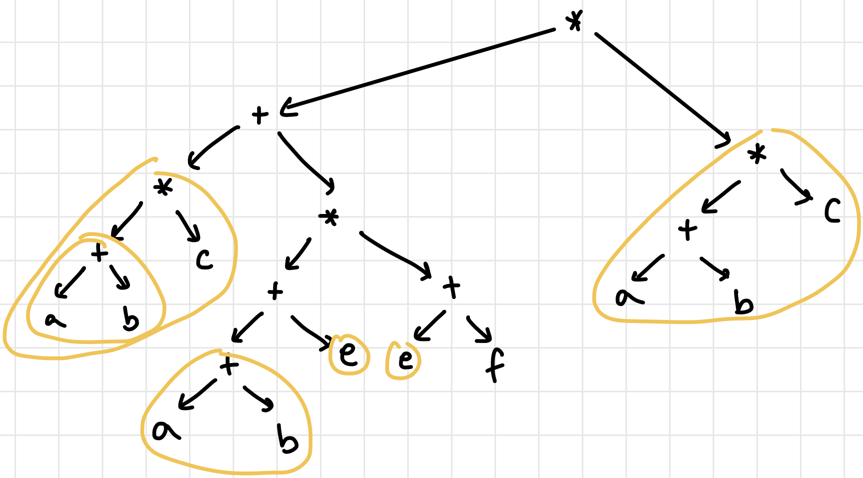
$$(((a+a)+(a+a)) + ((a+a)+(a+a)))$$

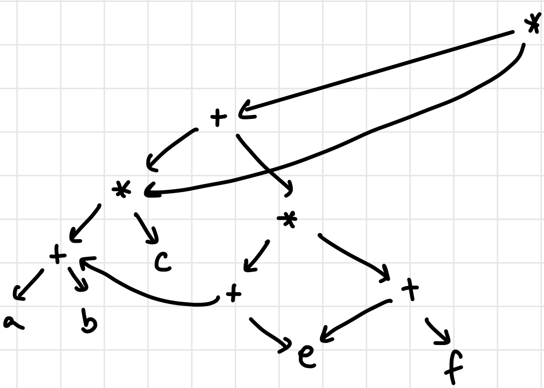




Q: For same grammar, construct DAG for

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





Three Address Code (TAC)

- Linearized representation of DAG
- At most one operator on RHS of instruction
- Each instr upto 3 addresses
  - Name (ID)
  - Constant (number)
  - Temporary (intermediate)

Format of TAC Instructions

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

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

Q: Generate TAC for the following

(i) $a+b*c-d/b*c$

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

(ii) $x = *p + \&y$

```
t1 = *p
t2 = &y
t3 = t1+t2
x = t3
```

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

```
t1 = y+1
param t1
t2 = call (f, 1)
t3 = t2+2
x = t3
```

(iv) $x = foo(2*x+3, y+10, g(i), h(3,j))$

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

(v) $x = f(g(i), h(3,j))$

```
param i
t1 = call g,1
param 3
param j
t2 = call h,2
param t1
param t2
t3 = call f,2
x = t3
```

(vi) alpha = (65 <= c && c <= 90) || (97 <= c && c <= 122)

```
t1 = 65 <= c
iffalse t1 goto L1
t2 = c <= 90
iffalse t2 goto L1
L0: alpha = true
goto next
L1: t3 = 97 <= c
 if t3 goto L0
 t4 = c <= 122
 if t4 goto L0
 alpha = false
 next:
```

Address Calculation for 1D Array

- Calculate A[i]

  A : base address
  W : size (in bytes) of single element
  i : index of element
  L_B : start index (default = 0)

$$A[i] = A + W(i - L_B)$$

Q: Generate TAC for the following

(i) a = b[i]   assume int, size = 4, start = 0

```
t1 = i * 4
t2 = b + t1
a = t2
```

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

```
t1 = i
L0: t1 = t1 + 1
t2 = 4 * i
t3 = a + t2     OR  a[t2]
t4 = t3 < v
if t4 goto L0
```

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

```
product = 0
i = 1
L0: t1 = 4*i
t2 = A[t1]
t3 = B[t1]
t4 = t2 * t3
t5 = product + t4
product = t5
t6 = i+1
i = t6
t7 = i < 20
if t7 goto L0
```

Address Calculation for 2D Arrays

1. Row major form
2. Column major form

m: # of rows
n: # of columns

1. Row major form $A_{m \times n}$

$$A[i][j] = A + w * [(n*i) + j]$$ → if start index ≠ 0, replace i with $i - L_r$ and j with $j - L_c$

2. Column major form $A_{m \times n}$

$$A[i][j] = A + W * [(m*j) + i]$$

- Assumptions:
  - Row-major
  - int (size = 4)
  - m x n matrix

Q: Generate TAC (c : 5x5 array)

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0;
```

```
i=0

L0: t1 = i<n
iffalse t1 goto endout
j=0
L1: t2= j<n
iffalse t2 goto endin
t3 = i*5
t4 = t3+j
t5 = 4*t4
c+t5 = 0
t6 =j+1
j = t6
goto L1
endin
t7 = i+1
i = t7
goto L0
endout
```

Q: Generate TAC (c: 10x10 array)

```
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        c[i][j] = a[i][j] + b[i][j]
```

note: if i=1 to 10, start index = 1

```
      i=0
L0:   t1= i<10
      iffalse t1 goto endout
      j=0
L1:   t2= j<10
      iffalse t2 goto endin

      t3= 10*i
      t4 = t3+j
      t5= 4*t4

      t6= a[t5]
      t7= b[t5]
      t8 = t6+t7
      c[t5] = t8

      t9= j+1
      j= t9
      goto L1

      endin

      t10= i+1
      i = t10
      goto L0

      endout
```

Data Structures for TAC

- Structure with operands and operators as fields
- Array or linked list of records
- 3 types of record structures
  1. Quadruples (4 fields)
  2. Triples (3 fields)
  3. Indirect triples (triples + list of pointers to triples)

1. Quadruples

op	arg1	arg2	result

- op: operator
- arg1, arg2: two operands used
- result: result of expression

pointers to symbol table entries

Unary Operators

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

Functions

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

Jumps

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

Labels and Returns

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

Array indexing

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

Q: Write TAC and quadruple representation

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

TAC:

```
t1 = x == 0
iffalse t1 goto E
u=1
goto endif
E:
t2= x-1
param t2
t3= call fact, 1
t4= t3 * x
u= t4
endif :
t5= u
value = t5
```

op	arg1	arg2	result
==	x	0	t1
iffalse	t1		E
=	1		u
goto			endif
label			E
-	x	1	t2
param	t2		
call	fact	1	t3
*	t3	x	t4
=	t4		u
label			endif
=	u		t5
=	t5		value

2. Triples

op	arg1	arg2

- op: operator
- arg1, arg2: two operands used pointers to symbol table entries

- Avoid temporary names in symbol table
  - instead, use serial no. of statement computing its value

- Problem: code immovability
  - not very efficient in optimizing compilers

Jumps and labels

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

Array indexing

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

Q: Write triple representation for TAC

```
t1 = -b
t2 = t1 * d
t3 = t1 + c
t4 = -b
t5 = t4 * d
t6 = t3 + t5
a = t6
```

	op	arg1	arg2
(1)	-	b	
(2)	*	(1)	d
(3)	+	(1)	c
(4)	-	b	
(5)	*	(4)	d
(6)	+	(3)	(5)
(7)	=	a	(6)

3. Indirect Triples

op	arg1	arg2

- op: operator
- arg1, arg2: two operands used   pointers to symbol table entries

- Separate list of pointers to statement numbers is maintained
- Can re-order statement list to reorder code
- Requires less space than quadruples

Q: Write indirect triple representation for TAC

```
t1 = -b
t2 = t1 * d
t3 = t1 + c
t4 = -b
t5 = t4 * d
t6 = t3 + t5
a = t6
```

	Stmt no
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)
(6)	(17)

Stmt no	op	arg1	arg2
(11)	-	b	
(12)	*	(0)	d
(13)	+	(0)	c
(14)	-	b	
(15)	*	(3)	d
(16)	+	(2)	(4)
(17)	=	a	(5)

Q: If above code changes to the following, how will indirect triple representation change?

```
t1 = -b
t2 = t1 * d
t3 = t1 + c
t4 = t1
t5 = t4 * d
t6 = t3 + t5
a = t6
```

	Stmt no
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(11)
(4)	(15)
(5)	(16)
(6)	(17)

Stmt no	op	arg1	arg2
(11)	-	b	
(12)	*	(0)	d
(13)	+	(0)	c
(14)	-	b	
(15)	*	(3)	d
(16)	+	(2)	(4)
(17)	=	a	(5)

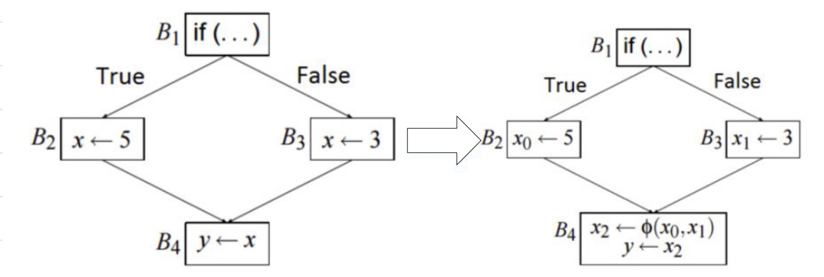
STATIC SINGLE ASSIGNMENT

- Each variable assigned only once; can be used multiple times
- IR variables split into versions (subscripts)

ϕ - Function

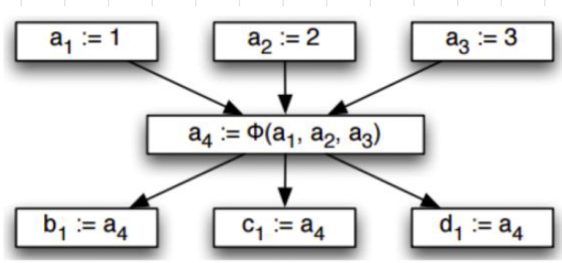
- control flow cannot be predicted
- ϕ functions — meet points for branched values
- $\phi(a1, a2, \ldots)$ – no. of arguments is no. of incoming flow edges
- Return value corresponds to control flow path

Example – Control Flow Graph

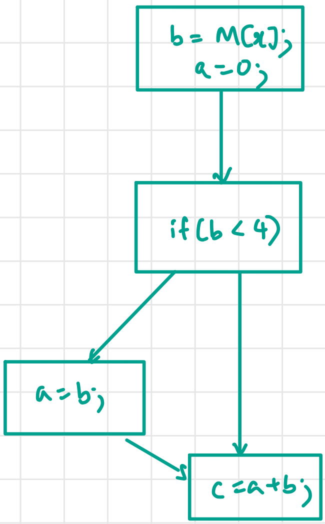


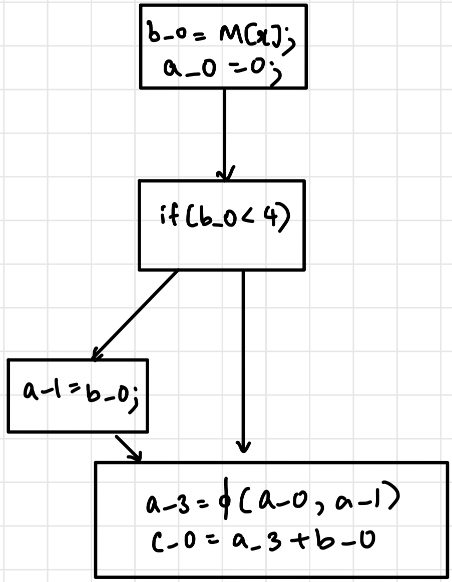
Example – Control Flow Graph

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



Q: Draw CFG





CFG Generation

- Rules
  1. Nodes: basic blocks — sequential statements (no branch)
  2. Nodes are numbered
  3. First basic block: initial block
  4. Directed

- Each instruction assigned only to one basic block

Rules for Determining Basic Blocks

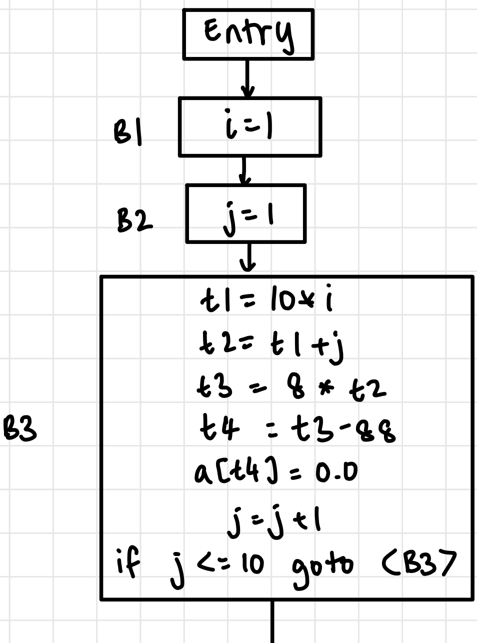
- Identify leaders
  - First TAC in IR
  - Target of jump
  - Instruction immediately after jump

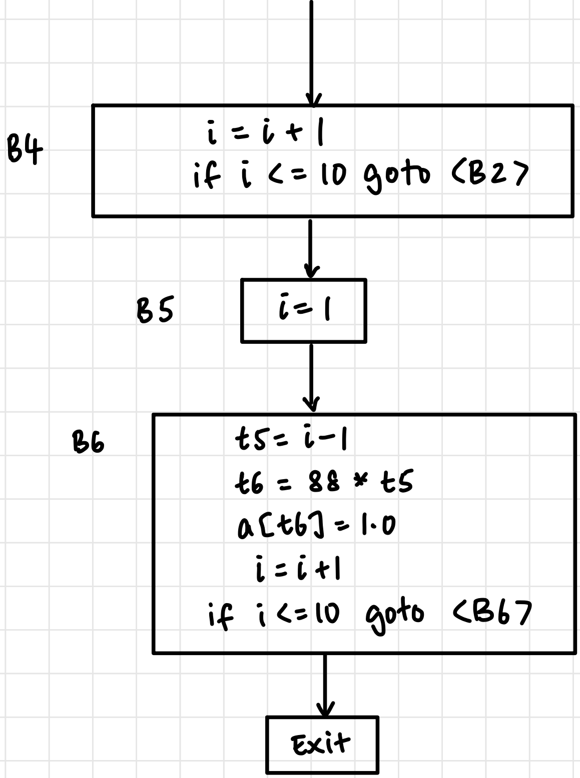
- Each basic block contains leader to next leader, excluding next leader

Q: Identify leaders and construct CFG

```
1.  i = 1                       first TAC is leader
2.  j = 1                       target of 11
3.  t1 = 10 * i                 target of 9
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                  follows jump
11.  if i <= 10 goto (2)
12.  i = 1                      follows jump
13.  t5 = i - 1                 target of 17
14.  t6 = 88 * t5
15.  a[t6] = 1.0
16.  i = i + 1
17.  if i <= 10 goto (13)
```

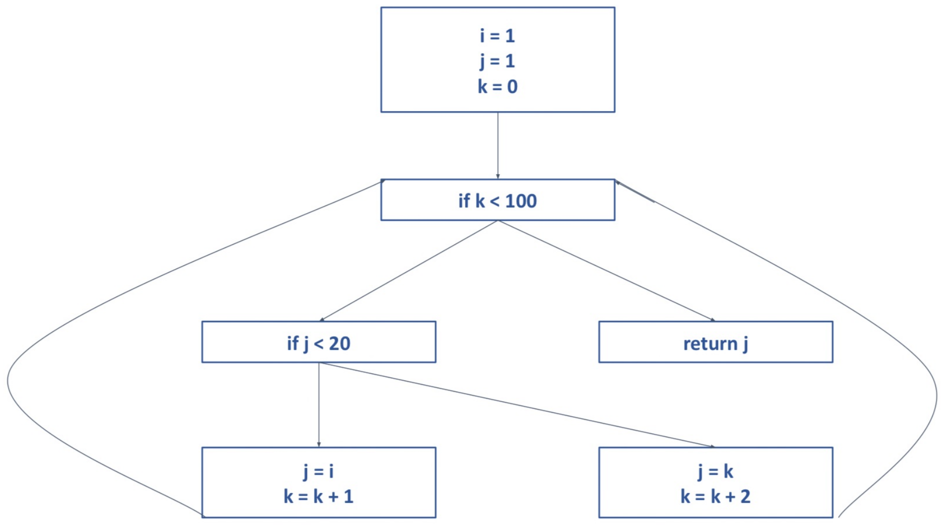
CFG

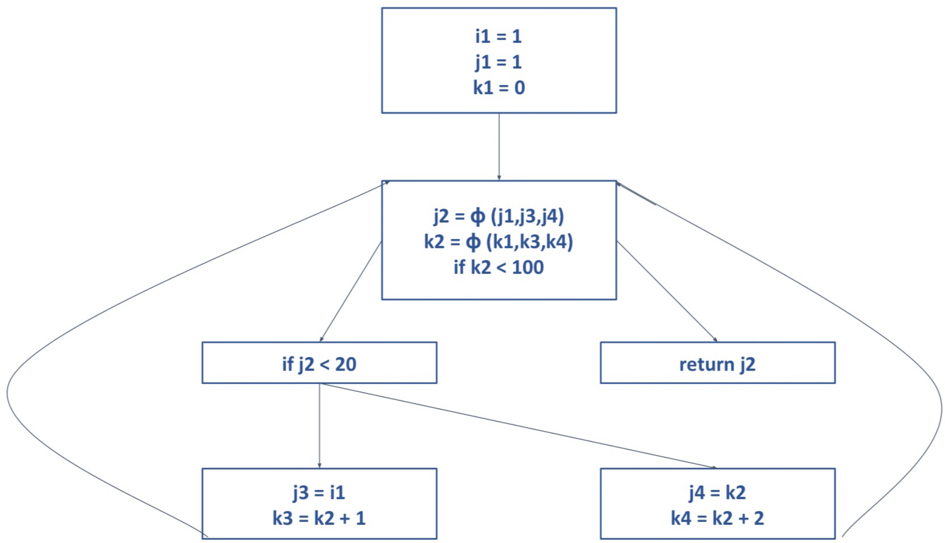




Q: convert to SSA

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

return j
```


i = 1
j = 1
k = 0
if k < 100
if j < 20
return j
j = i
k = k + 1
j = k
k = k + 2


i1 = 1
j1 = 1
k1 = 0
j2 = ϕ (j1,j3,j4)
k2 = ϕ (k1,k3,k4)
if k2 < 100
if j2 < 20
return j2
j3 = i1
k3 = k2 + 1
j4 = k2
k4 = k2 + 2

OPTIMIZATION

- Make code consume less resources and deliver high speed
- Good code optimization
  1. Semantics preserving
  2. Speed up programs on avg
  3. Worth the effort

Techniques

1. Control Flow Analysis:
   - Identify loops in CFG – room for improvement

2. Data Flow Analysis:
   - Information about how variables are used in a program

- Best place to optimize – at source code level

Optimization
- Machine Independent
  - I/P: intermediate code
  - Based on CFG
    - Global: apply to CFG
    - Local: apply to block
- Machine Dependent
  - I/P: target code

Machine Independent Code Optimizations

1. Constant folding
2. Constant propagation
3. Common subexpression elimination (CSE)
4. Copy propagation
5. Dead code elimination (DCE)
6. Strength reduction
7. Packing temporaries
8. Loop optimizations

1. Constant Folding

- Evaluate constant expressions at compile time instead of runtime

$$i = 30 * 20 + 10;$$

- Algebraic identities (true regardless of variable values)

$$0 * x = 0$$
$$1 * x = x$$

- Concatenation of string literals / constant strings

"alpha" + "bet" = "alphabet"

2. Constant Propagation

- Substituting values of known variables at compile time

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

- Propagating a

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

- Find b

```
int a=10;
int b= 6
return b+200* 10
```

- Fold b

```
int a=10;
int b=6;
return 6 +200* 10
```

- Constant fold

```
int a=10;
int b =6;
return 2006;
```

- Remove dead code

```
return 2006;
```

3. Common Subexpression Elimination

- Replace instances of identical expressions with a single variable holding the computed value
- Values in subexpression have not changed

Q: Eliminate common subexpressions

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

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

Q: Eliminate common subexpressions

t6 = 4 * i	→ retain	t6 = 4*i
x = a[t6]		x = a[t6]
t7 = 4 * i	→ eliminate	t7 = t6
t8 = 4 * j	→ retain	t8 = 4*j
t9 = a[t8]		t9 = a[t8]
a[t7] = t9		a[t7] = t9
t10 = 4 * j	→ eliminate	t10 = t8
a[t10] = x		a[t10] = x
goto B2		goto B

4. Copy Propagation

- Replace occurences of targets of direct assignments with their values

$u = v$

use v instead of u whenever possible

Q: Copy propagate

$y = x$
$z = 3 + y$

dead code

$y = x$
$z = 3 + x$ $\longrightarrow$ $z = 3 + x$

5. Dead Code Elimination

- Code that does nothing useful / is never executed

Q: Eliminate dead code

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

6. Strength Reduction

- Replace expensive operation with cheaper one

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

7. Packing Temporaries

- Replace distinct temporaries with a single one when no longer required

Q: Pack temprraries

t1 = a + a	t1 = a + a
t2 = t1 + b	t1 = t1 + b
c = t2 * t2	c = t1 * t1

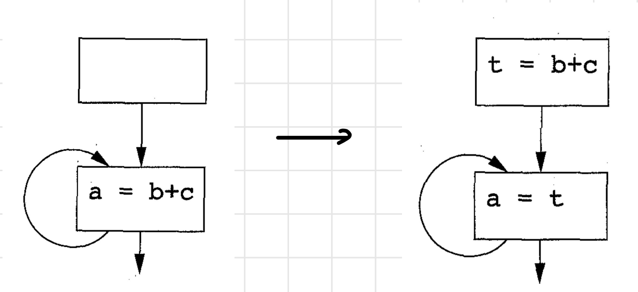
8. Loop Optimizations

- Increase execution speed and reduce overhead

- Types
  (a) Loop invariant detection
  (b) Code motion
  (c) Loop unrolling / unwinding
  (d) Induction variable detection

(a) Loop invariant detection

- Variable whose value is constant for the duration of the loop



a = 5 invariant

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

(b) Code Motion

- Decrease amount of code (by identifying loop invariants)

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

(c) Loop Unwinding

- Reduce no. of iterations by replicating body of loop
- Reduce # of jumps & condition tests

- Requires # of iterations to be known at compile time

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

→

```
for (i=0; i<50; ++i) {
    g();
    g();
}
```

(d) Induction Variable Detection

- Var that gets increased/decreased by fixed amount at every iteration of loop

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

→

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

Q: Detect induction variables

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

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

LOCAL CODE OPTIMIZATION

- Apply techniques on code snippets

Q: Optimize

$$\begin{array}{lcl}
a = x^2 & & a = x*x \\
b = 3 & & b = 3 \\
c = x & \xrightarrow{\text{strength reduction}} & c = x \\
d = c*c & & d = c*c \\
e = b*2 & & e = b \ll 1 \\
f = a+d & & f = a+d \\
g = e*f & & g = e*f
\end{array}$$

$$\begin{array}{lcl}
a = x*x & & a = x*x \\
b = 3 & & b = 3 \\
c = x & & c = x \\
d = c*c & \xrightarrow{\text{constant prop.}} & d = c*c \\
e = b \ll 1 & & e = 3 \ll 1 \\
f = a+d & & f = a+d \\
g = e*f & & g = e*f
\end{array}$$

$$\begin{array}{lcl}
a = x*x & & a = x*x \\
b = 3 & & b = 3 \\
c = x & & c = x \\
d = c*c & \xrightarrow{\text{constant prop.}} & d = c*c \\
e = 3 \ll 1 & & e = 6 \\
f = a+d & & f = a+d \\
g = e*f & & g = 6*f
\end{array}$$

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

$\xrightarrow{\text{copy prop.}}$

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

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

$\xrightarrow{\text{CSE}}$

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

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

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

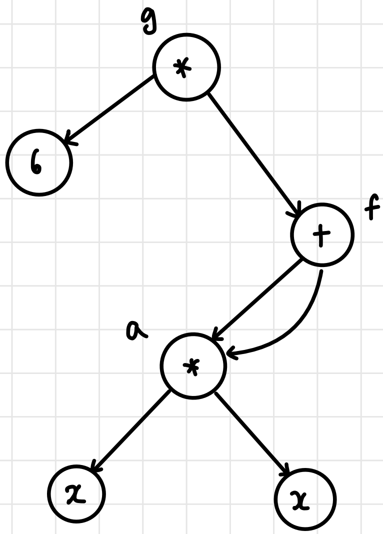
$\xrightarrow{\text{DCE}}$

```
a = x*x



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

LOCAL OPTIMIZATION USING DAG

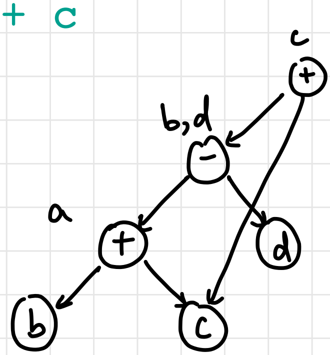
- Take prev example

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



Q: Local optimization using DAG

$a = b + c$
$b = a - d$
$c = b + c$
$d = b$

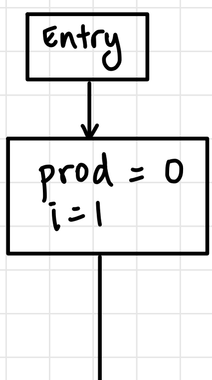


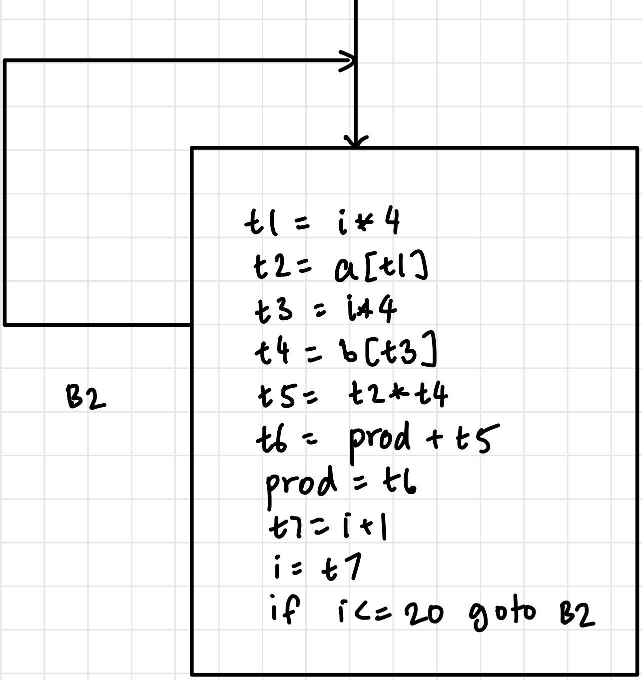
Q: Local optimization using DAG

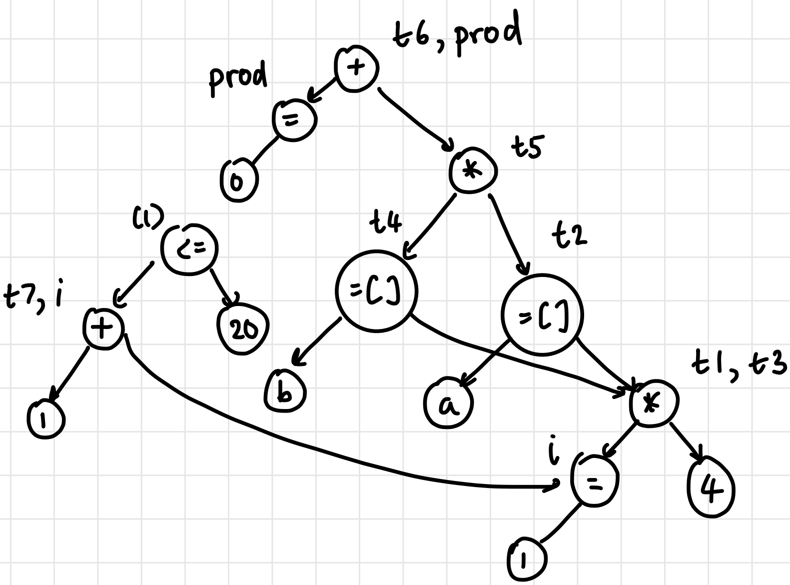
```
prod = 0;
i = 1;
do
  prod = prod + a[i] * b[i];
  i = i + 1;
while (i <= 20);
```

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

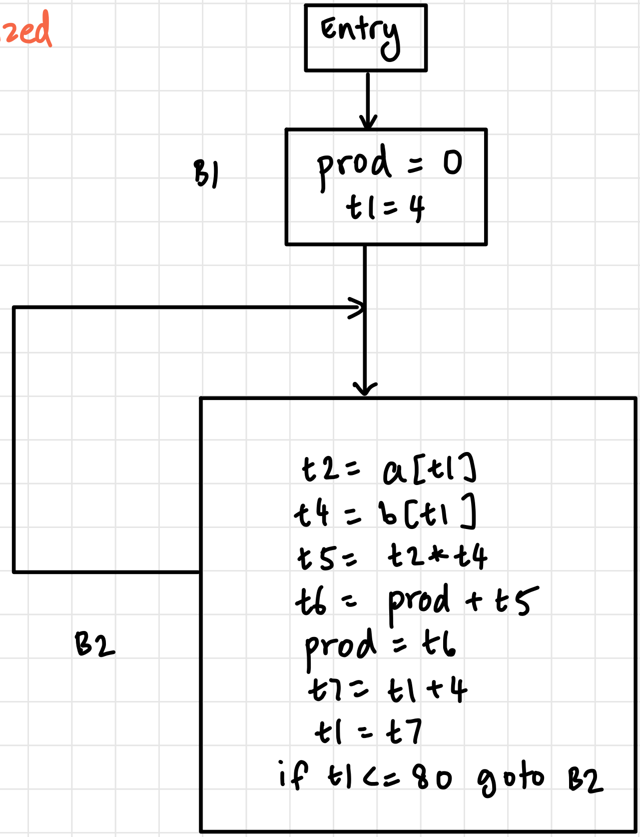
B1



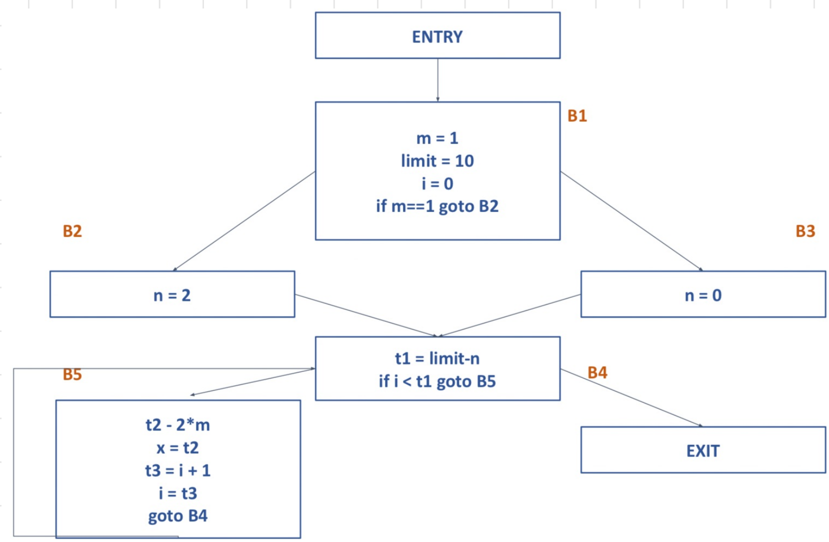

t1 = i*4
t2= a[t1]
t3 = i*4
t4 = b[t3]
t5= t2*t4
t6 = prod + t5
prod = t6
t7= i+1
i= t7
if i<= 20 goto B2
B2


t6, prod
prod
+
=
0
*
t5
(1)
<=
t4
t2
=[]
=[]
t7, i
+
20
b
a
*
t1, t3
1
i
=
4
1

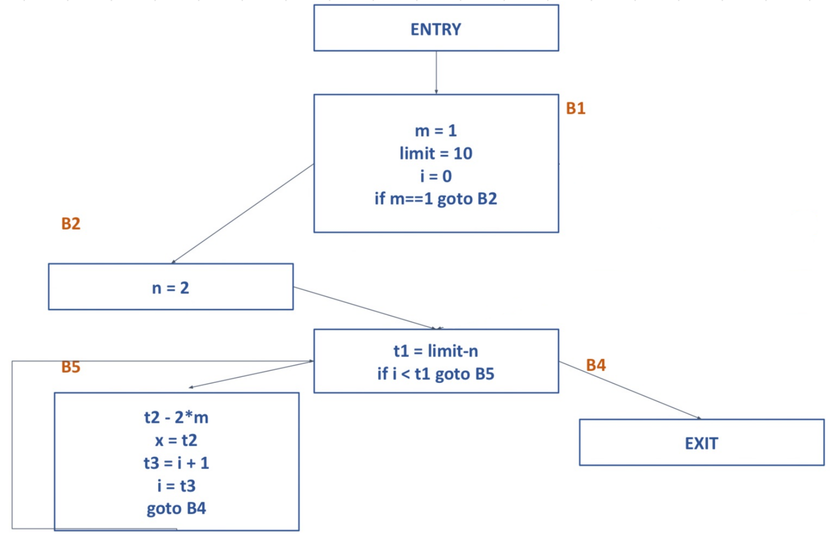
Optimized



Q: Optimize CFG



1. if m==1 is always true → remove B3 (unreachable)



2. move B2 to B1

NEXT-USE ALGORITHM

- Two pass algorithm
- Pass 1:
  - Scan forward over basic block
  - For every variable x in BB, set x.live = FALSE and x.next_use = NONE
- Pass 2:
  - Scan backwards over basic block
  - For every tuple (i) x = y op z, copy live/NU info from x, y, z's ST entries into tuple data and update ST entries as follows

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

Q: Go through NUA for the following code

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

FIRST PASS

Statement	ST Info						Instruction Info					
	Live			Next Use			Live			Next Use		
	x	y	z	x	y	z	x	y	z	x	y	z
(1) $x = y + z$	F	F	F				F	F	F			
(2) $z = x * 5$	F	F	F				F	F	F			
(3) $y = z - 7$	F	F	F				F	F	F			
(4) $x = z + y$	F	F	F				F	F	F			

SECOND PASS

Step 1 : Start with (4)

tuple data

Statement	ST Info						Instruction Info					
	Live			Next Use			Live			Next Use		
	x	y	z	x	y	z	x	y	z	x	y	z
(1) $x = y + z$	F	F	F				F	F	F			
(2) $z = x * 5$	F	F	F				F	F	F			
(3) $y = z - 7$	F	F	F				F	F	F			
(4) $x = z + y$	F	T	T	-	4	4	F	F	F			

Step 2 : Copy ST info to tuple data

Statement	ST Info						Instruction Info					
	Live			Next Use			Live			Next Use		
	x	y	z	x	y	z	x	y	z	x	y	z
(1) $x = y + z$	F	F	F				F	F	F			
(2) $z = x * 5$	F	F	F				F	F	F			
(3) $y = z - 7$	F	F	F				F	T	T	-	4	4
(4) $x = z + y$	F	T	T	-	4	4	F	F	F			

Step 3: Continue for (3), (2), (1)

Statement	ST Info						Instruction Info					
	Live			Next Use			Live			Next Use		
	x	y	z	x	y	z	x	y	z	x	y	z
(1) $x = y + z$	F	T	T	-	1	1	T	F	F	2	-	-
(2) $z = x * 5$	T	F	F	2	-	-	F	F	T	-	-	3
(3) $y = z - 7$	F	F	T	-	-	3	F	T	T	-	4	4
(4) $x = z + y$	F	T	T	-	4	4	F	F	F			

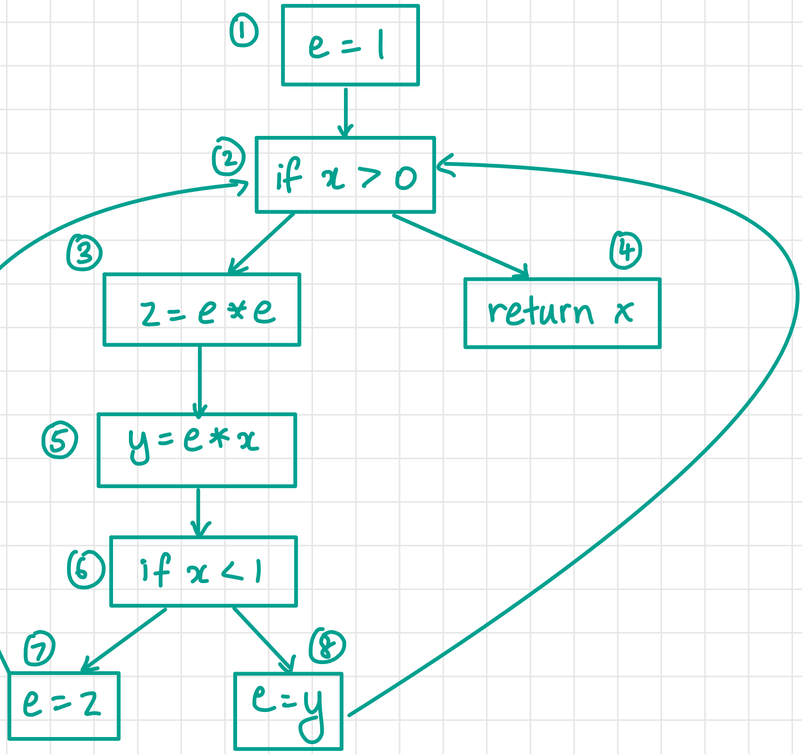
LIVE VARIABLE ANALYSIS

- Liveness associated with edges of CFG, not nodes
- Data flow values
  1. use[n]: set of variables used in node n
  2. def[n]: set of variables defined in node n
  3. in[n]: variables live on entry to node n
  4. out[n]: variables live on exit from node n
- Data flow equations for each basic block B

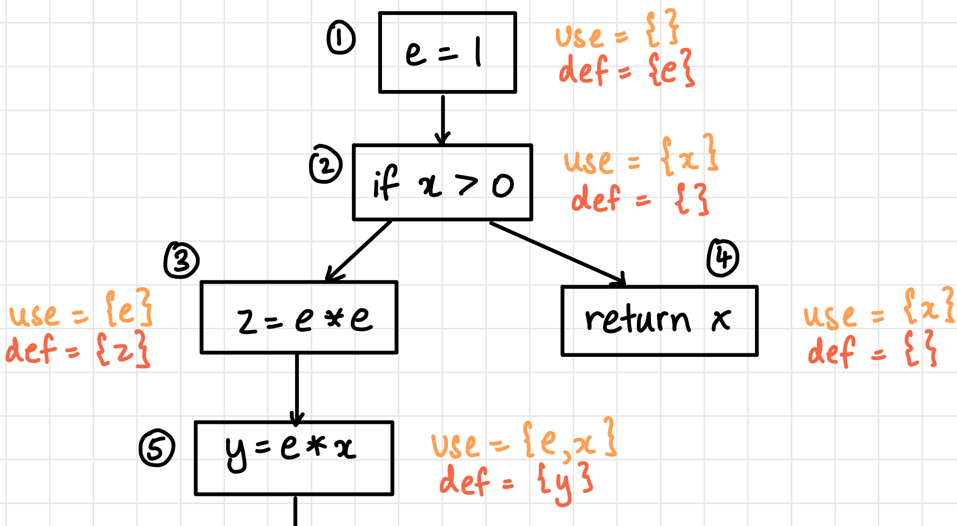
$$out[B] = \bigcup_{S \text{ is a successor of } B} (in[S])$$

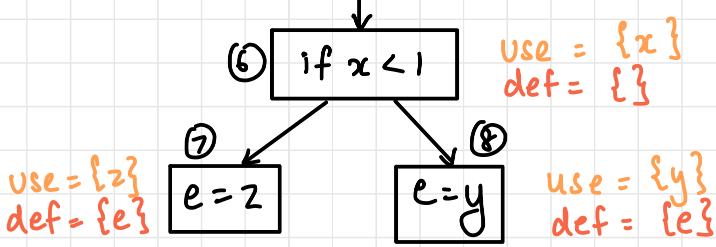
$$in[B] = use[B] \cup (out[B] - def[B])$$

Q: Compute liveness info for the CFG

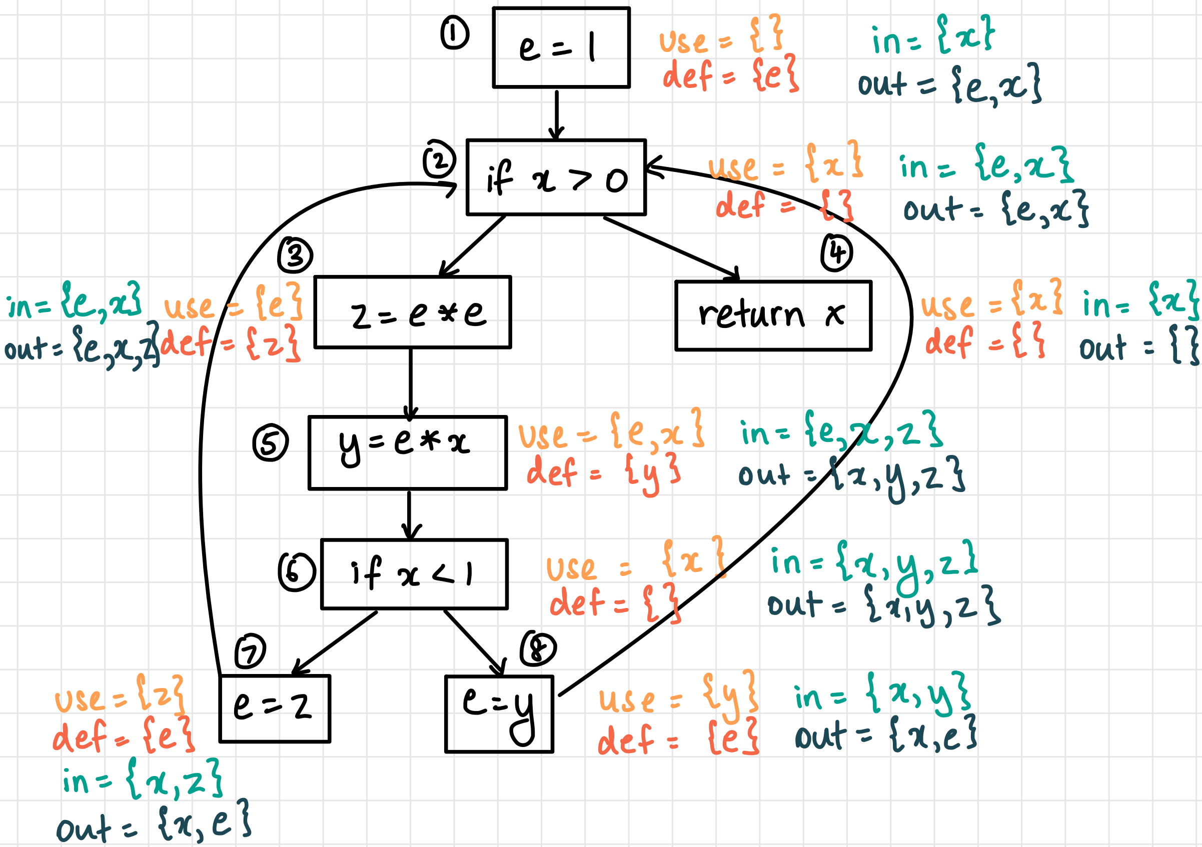


Step 1: compute use & def

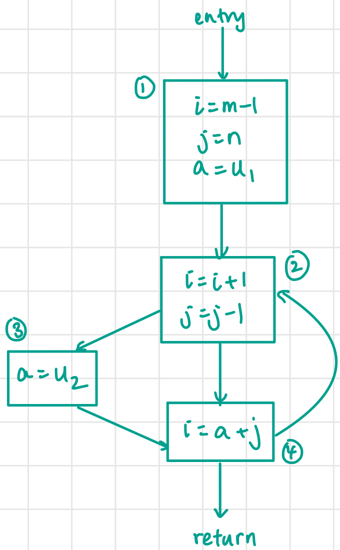


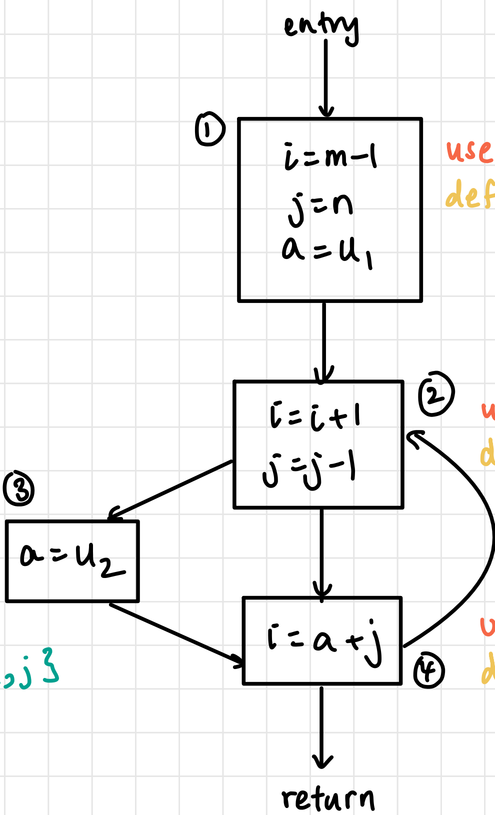


Step 2: Compute in and out



Q: Compute liveness information for CFG





1: use = $\{m, n, u_1\}$ in = $\{m, n, u_1, u_2\}$
def = $\{i, j, a\}$ out = $\{a, i, j, u_2\}$

2: use = $\{i, j\}$ in = $\{a, j, i, u_2\}$
def = $\{i, j\}$ out = $\{a, j, u_2\}$

3: use = $\{u_2\}$
def = $\{a\}$
in = $\{u_2, j\}$
out = $\{a, u_2, j\}$

4: use = $\{a, j\}$ in = $\{a, j, u_2\}$
def = $\{i\}$ out = $\{i, j, u_2, a\}$