NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 13: Intermediate representations and SSA

2020-02-21

Michael Engel

Overview

- More on intermediate representations
  - Efficient implementation
  - Translation an AST into linear IR
  - Static single assignment (SSA) form

TAC example

Intermediate code

- TAC resembles a RISC-like ***register machine***
  - Operands have to be loaded into registers
  - Operations (other than load/store) operate on register values
  - Results are delivered in registers
- Limited constraints for naming/allocating registers compared to real machines

TAC code for `a - 2 × b`

```
t1 ← 2
t2 ← b
t3 ← t1 × t2
t4 ← a
t5 ← t4 - t3
```

ARM assembler code for `a - 2 × b`

```
MOV  R1, #2      // R1=2
LDR  R2, =b
LDR  R2, [R2]    // R2=b
MULU R3, R0, R2  // R3=2*b
LDR  R4, =a
LDR  R4, [R4]    // R4=a
SUB  R5, R4, R3  // R5=R4-R3=a-2*b
```

Three-address code (TAC)

Intermediate code

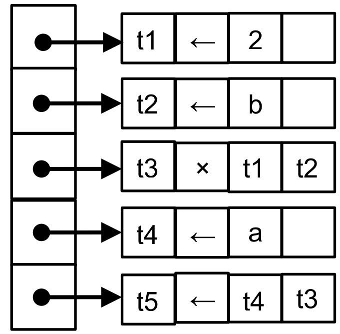
- Most operations in TAC have the form `i = j op k`
  - one operator (`op`), two operands (`j` and `k`) and one result (`i`)
  - some operators will need fewer arguments
    - e.g. immediate loads and jumps
  - sometimes, an op with more than three addresses is needed
- Three-address code is reasonably compact
  - most ops consist of four items: an operation and three names
  - both the operation and the names are drawn from limited sets
  - operations typically require 1 or 2 bytes
  - names are typically represented by integers or table indices
    - in either case, 4 bytes is usually enough
- Data structure choices affect the costs of operations on IR

Representing Linear IRs

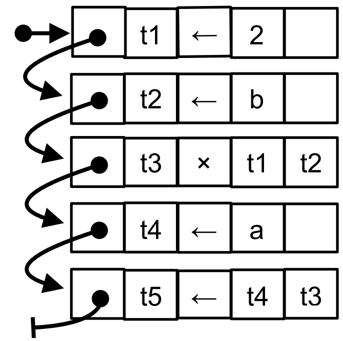
Intermediate code

Target	Op	Arg1	Arg2
t1	←	2	
t2	←	b	
t3	×	t1	t2
t4	←	a	
t5	-	t4	t3

Simple array



Array of pointers



Linked List

```
t1 ← 2
t2 ← b
t3 ← t1 × t2
t4 ← a
t5 ← t4 - t3
```

TAC code for
a - 2 × b

- **Simple array**: most simple form
  - short array to represent each basic block
  - often, the compiler writer places the array inside CFG node
- **Array of pointers** groups quadruples into a block
  - the pointer array can be contained in a CFG node
- **Linked list** links the quadruples together to form a list
  - requires less storage in the CFG node
  - at the cost of restricting accesses to sequential traversals

Tradeoffs of different repres.

- Use case: optimization of code
- Example: rearranging the code in this block
  - What are the costs incurred for each representation?
- Op 1 loads a constant into a register
  - on most machines this translates directly into an immediate load operation
- Ops 2 and 4 load values from memory
  - on most machines this 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**
  - What is the cost of doing this?

```
1 t1 ← 2
2 t2 ← b
3 t3 ← t1 × t2
4 t4 ← a
5 t5 ← t4 - t3
```

Tradeoffs of different repres.

Simple array: move 2 ahead of 1

```
1 t1 ← 2
2 t2 ← b
3 t3 ← t1 × t2
4 t4 ← a
5 t5 ← t4 - t3
```

Target	Op	Arg1	Arg2
t1	←	2	
t2	←	b	
t3	×	t1	t2
t4	←	a	
t5	-	t4	t3

move

Target	Op	Arg1	Arg2
t2	←	b	
t2	←	b	
t3	×	t1	t2
t4	←	a	
t5	-	t4	t3

save

t1	←	2	

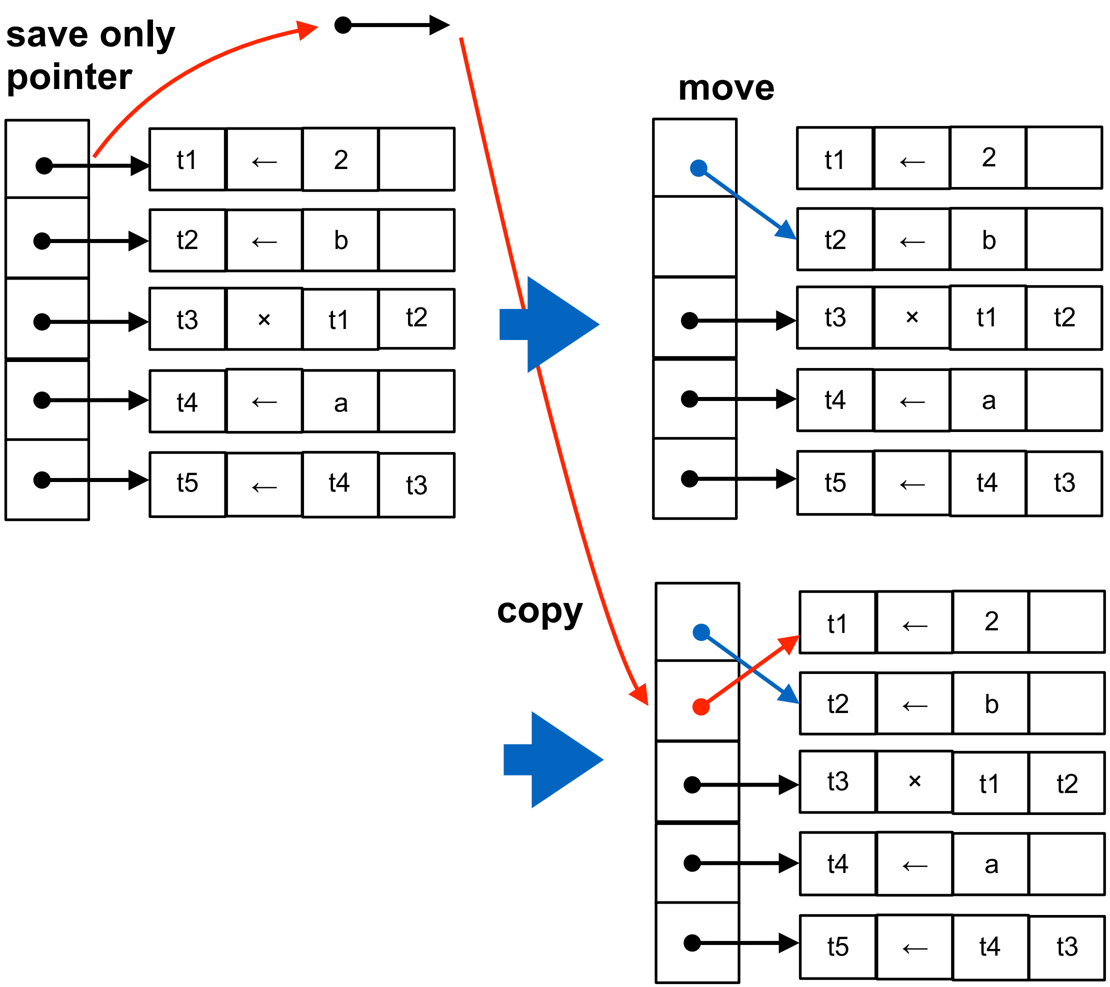
copy

Target	Op	Arg1	Arg2
t2	←	b	
t1	←	2	
t3	×	t1	t2
t4	←	a	
t5	-	t4	t3

Tradeoffs of different repres.

Intermediate code

Array of pointers: move 2 ahead of 1

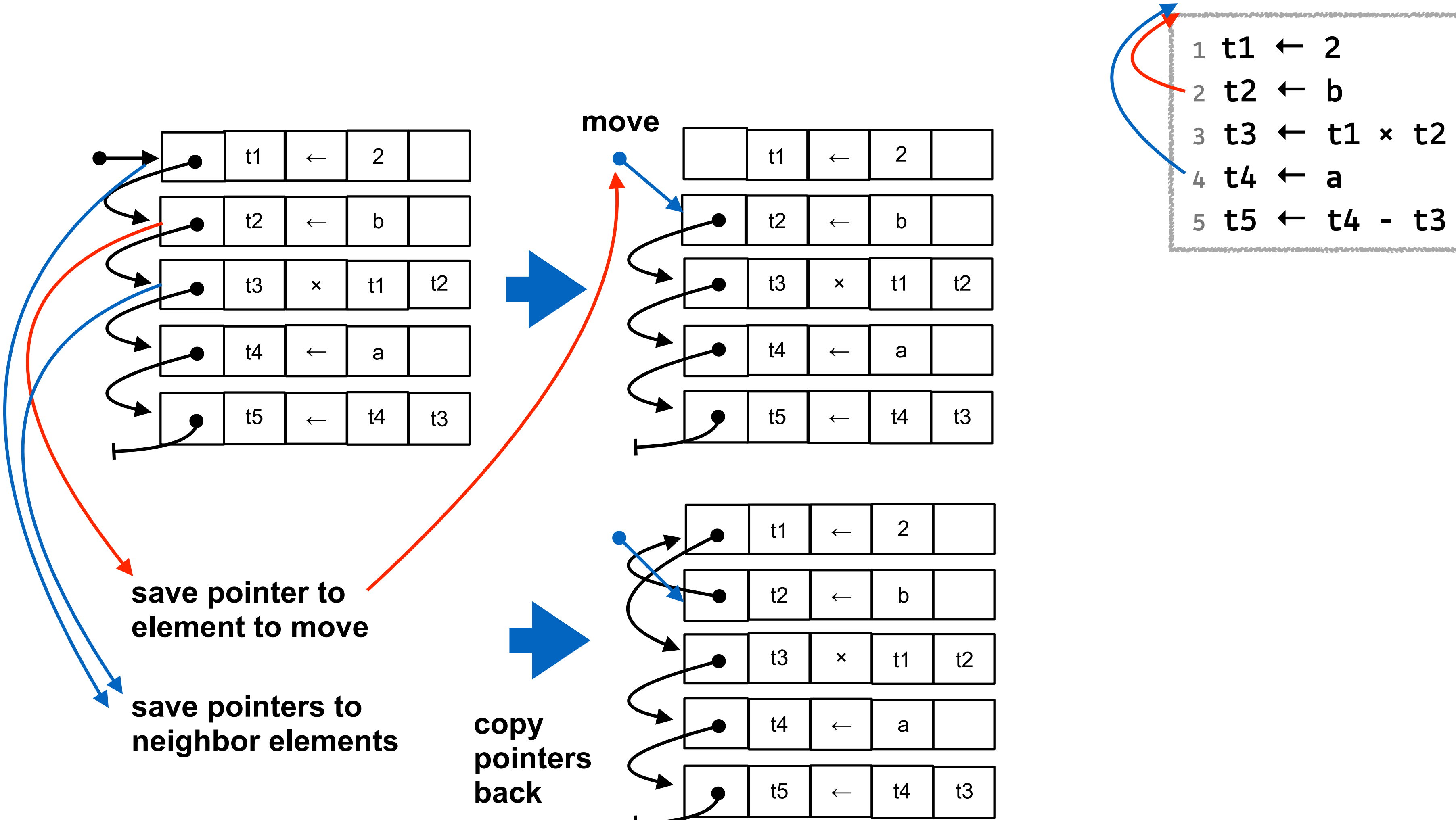



```
1 t1 ← 2
2 t2 ← b
3 t3 ← t1 × t2
4 t4 ← a
5 t5 ← t4 - t3
```

Tradeoffs of different repres.

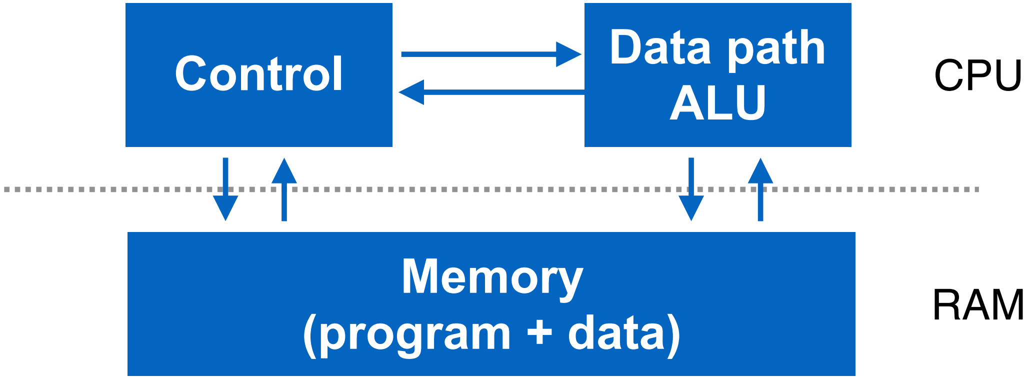
Intermediate code

Linked list: move 2 ahead of 1



A closer look at TAC

- Most modern computers (still) try to look like a von Neumann machine (even though they are far more complex internally)
- A von Neumann machine has three main components:
  - Control unit
  - Data path + ALU
  - Unified memory for instructions and data
- A clock controls the execution of instructions
  - Instruction fetch (from memory, addressed py PC)
  - Operand fetch (from memory addresses encoded in instr.)
  - Execute the instruction
  - Write back the results



Instruction classes

We need

- Instructions for control unit
- Data for data unit/ALU
  - Instructions and data are in memory
    - we can use ***symbolic names*** for these instead of numeric addresses:
      - ***Labels*** for instructions
      - ***Names*** for variables
- We can categorize instructions:

Math, logic, data movement →

Binary operations
Unary operations
Copy operations
Load/store operations

Unconditional jumps
Conditional jumps
Procedure calls

← **Control flow**

TAC is a low-level IR

Intermediate code

"Three address" since each operation deals with at most three addresses in memory (+ the instruction itself):

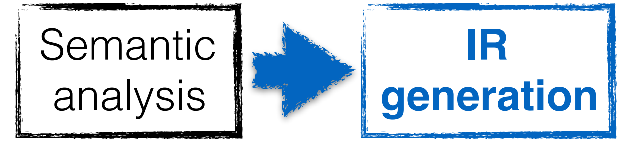
• Binary operations:	`a = b OP c`	OP is ADD, MUL, SUB, …
• Unary operations:	`a = OP b`	OP is NEG, MINUS, …
• Copy:	`a = b`	
• Load/store:	`x = &y`	address of y
	`x = *y`	value at addres y
	`x[i] = y`	address + offset

Control flow in TAC

Control flow is equally simple:

- Label: `L:` named address of next instruction
- Unconditional jump: `jump L` go to L and get next instruction
- Conditional jump:
  - `if x goto L` go to L if x is TRUE
  - `ifFALSE x goto L` go to L if x is FALSE
  - `if x<y goto L` comparison operators
  - `if x>=y goto L` comparison operators
  - `if x!=y goto L` comparison operators
- Call and return:
  - `param x` x is parameter in next call
  - `call L` similar to jump
  - `return` ...to where we came from

Translating to TAC



Translation of binary operators:

we make use of the recursive nature of our AST

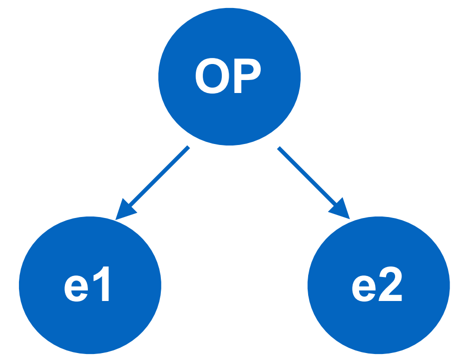
- No matter how complex the contents of expressions `e1` and `e2` are, this can be translated from

```
t = T[e1 OP e2]
```

into

"T" = "translation"

```
t1 = T[e1]
t2 = T[e2]
t3 = t1 OP t2
```

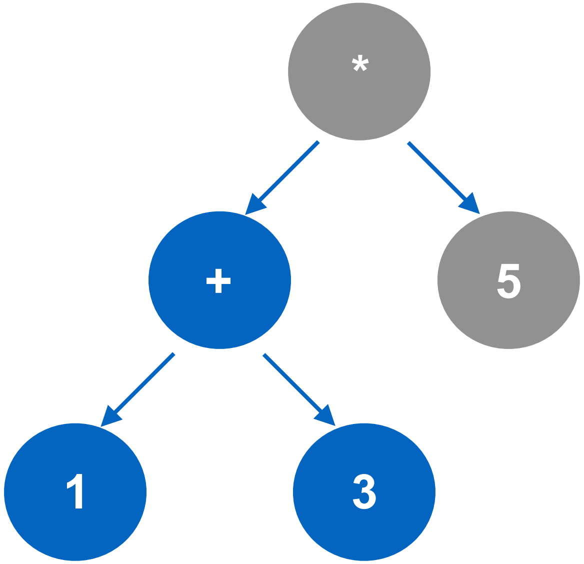


- First, (recursively) translate `e1` and store its result
- then, (recursively) translate `e2` and store its result
- finally, combine the two stores results using `OP`

Linearizing the program

We traverse the AST in depth-first order:

```
t1 = 1
t2 = 3
t3 = t1 + t2
```

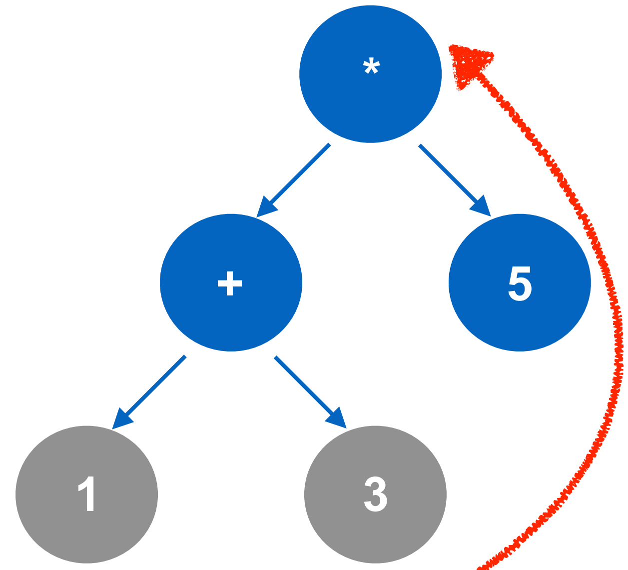


Linearizing the program

Semantic analysis → **IR generation**

We traverse the AST in depth-first order:

```
t1 = 1
t2 = 3
t3 = t1 + t2
```



Then we continue further up the tree:

- The result of the "+" operation is in `t3`

```
t4 = t3
t5 = 5
t6 = t4 * t5
```

- The final result can be copied:

```
t  = t6
```

Nested expressions

Semantic analysis → **IR generation**

Combine the local parts which represent sub-trees:

```
t1 = 1
t2 = 3
t3 = t1 + t2
```

T[1+3]

```
t4 = t3
t5 = 5
t6 = t4 * t5
```

T[t3*5]

```
t  = t6
```

T[(1+3)*5]

t = T[(1+3)*5]

Statement sequences

Straightforward, since they are already sequenced:

```
T[ s1; s2; s3; … ]
```

becomes

```
T[s1]
T[s2]
T[s3]
…
```

Simply translate one statement after the other and append their translations in order

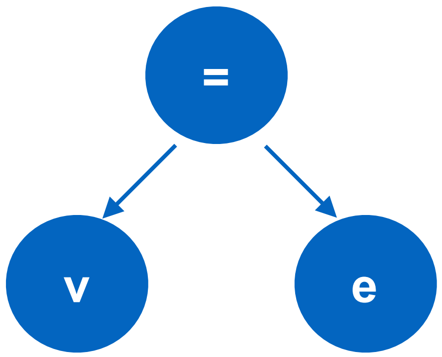
Assignments

Straightforward, since they are already sequenced:

```
T[ v=e ]
```

requires us to

- obtain the result of `e`
- put the result into `e`




```
T[ v=e ] -> t = T[e]
            v = t
```

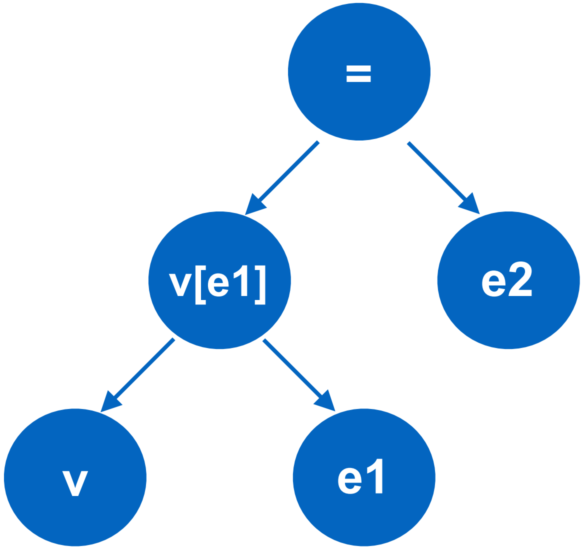
Array assignment

We need to also calculate the index (address offset)

```
T[ v[e1]=e2 ]
```

requires us to

- compute the index expression `e1`
- compute the expression `e2`
- put the result into `v[e1]`




```
T[ v[e1]=e2 ] -> t1 = T[e1]
                 t2 = T[e2]
                 v[t1] = t2
```

Conditionals

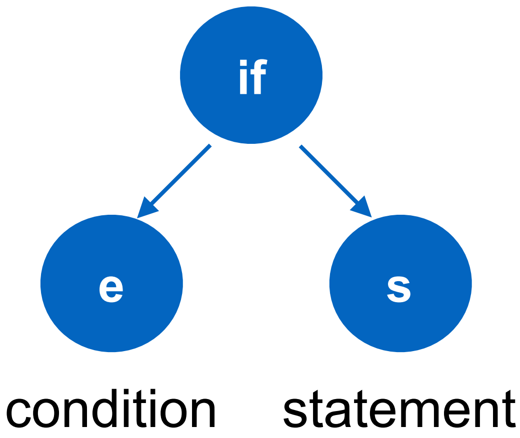
These require control flow

```
T[ if(e) then s ]
```

becomes

```
    t1 = T[e]
    ifFALSE t1 goto Lend
    T[s]
Lend:
```

(translation of next statement follows here)



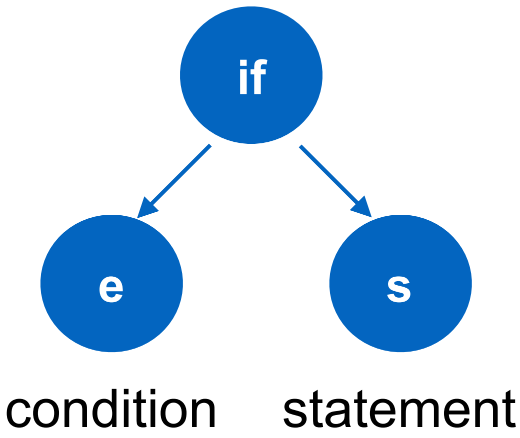
Conditionals

If e is true, control goes through s
If e is false, control skips past it

```
    t1 = T[e]
    ifFALSE t1 goto Lend
    T[s]
Lend:
```

t1 = true

t1 = false



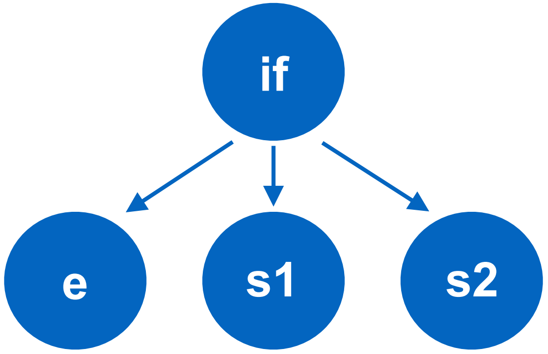
Conditionals + else

Easy to derive:

```
t1 = T[e]
ifFALSE t1 goto Lelse
T[s1]
jump LEnd
Lelse:
T[s2]
Lend:
```

t1 = **true**

t1 = **false**



While loops

The condition has to be checked at the beginning of each iteration:

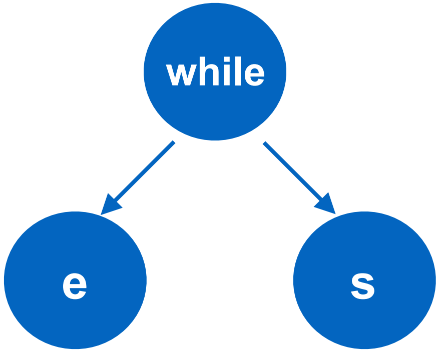
```
T[while(e) do s]
```

becomes

```
Ltest:
    t1 = T[e]
    ifFALSE t1 goto Lend
    T[s]
    jump Ltest
Lend:
```

`t1` **= true**

`t1` **= false**



Different kinds of loop

For and repeat loops can be transformed into while loops:

```
for (i=0; i<10; i++) {
    dosomething();
}
```

⟺

```
i=0;
while (i<10) {
    dosomething();
    i = i+1;
}
```

```
do {
    dosomething();
} while(x);
```

⟺

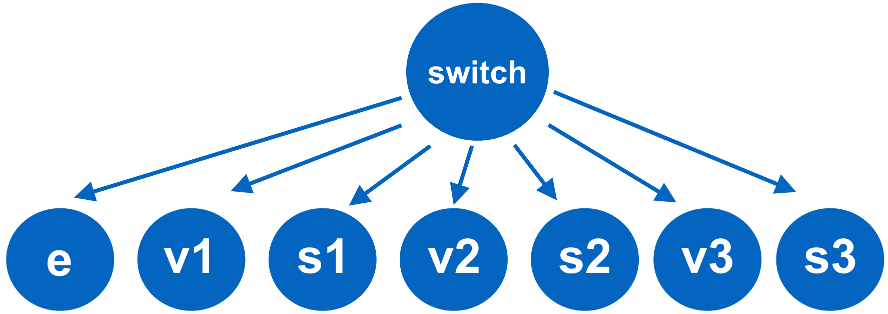
```
dosomething();
while (x) {
    dosomething();
}
```

Switch


```
T[switch(e){ case v1:s1; … case vn:sn]
```

can become

```
    t = T[e]
    ifFALSE (t=v1) goto L1
    T[s1]
L1: ifFALSE (t=v2) goto L2
    T[s2]
L2: …
    ifFALSE (t=vn) goto Lend
    T[sn]
Lend:
```

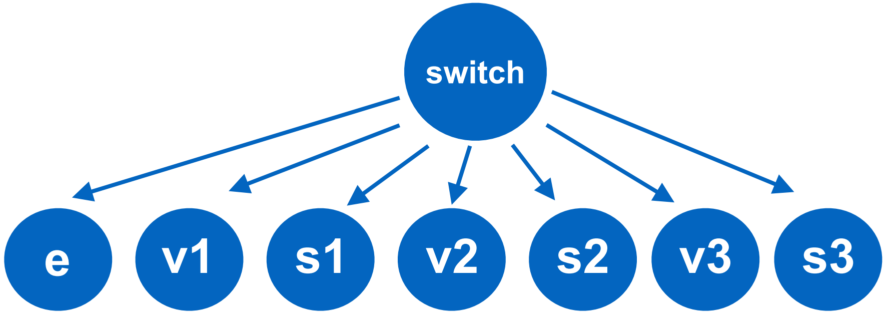


Switch using jump table


```
T[switch(e){ case v1:s1; … case vn:sn]
```

can also become

```
    t = T[e]
    jump table[t]
Lv1:T[s1]
Lv2:T[s2]
…
Lvn:T[sn]
Lend:
```



This models the C-like "fall-through" behavior without a break at the end of the case. Otherwise, we would have to insert "jump Lend" here!

Here, the compiler has to provide a ***jump table*** which maps the conditions `v1`, `v2`, … `vn` to their respective labels `Lv1`, `Lv2`, … `Lvn`

Using labels

Semantic analysis → **IR generation**

Labels must be unique

- This can be handled by numbering the statements that generate them:

```
if (e1) then s1;
if (e2) then s2;
```

becomes

```
    t1 = T[e1]
    ifFALSE t1 goto LEnd1
    T[s1]
Lend1:
    t2 = T[e2]
    ifFALSE t2 goto LEnd2
    T[s2]
Lend2:
```

Nested statements

`if (e1) then if (e2) then a=b` **requires a bit of care:**

outer if (#1)

```
t1 = T[e1]
ifFalse (t1) goto Lend1
t2 = T[e2]
ifFalse (t2) goto Lend2
t3 = b
a  = t3
Lend2:
Lend1:
```

inner if (#2)

Statement

Static Single-Assignment Form

Intermediate code

- Static single-assignment form (SSA) is a naming discipline that many modern compilers use to encode information about both the flow of control and the flow of data values in the program
  - names correspond uniquely to specific definition points in the code
  - each name is defined by one operation
  - hence the name static single assignment
- SSA abstracts from processor registers
  - helps to name intermediate values during compilation
- Each use of a name as an argument in some operation encodes information about where the value originated
  - each textual name refers to a specific definition point

Static Single-Assignment Form

Intermediate code

- A program is in SSA form when it meets two constraints:
  (1) each definition has a distinct name; and
  (2) each use refers to a single definition
- Transforming an IR program to SA form:
  - compiler inserts ϕ functions at points where different control-flow paths merge
  - it then renames variables to make the single-assignment property hold

```
x ← …
y ← …
while (x < 100)
    x ← x + 1
    y ← y + x
```

→

```
      x0 ← …
      y0 ← …
      if (x0 >= 100) goto next
loop: x1 ← ϕ(x0,x2)
      y1 ← ϕ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x
      if (x0 < 100) goto loop
next: x3 ← ϕ(x0,x2)
      y3 ← ϕ(y0,y2)
```

Translation of code into SSA form

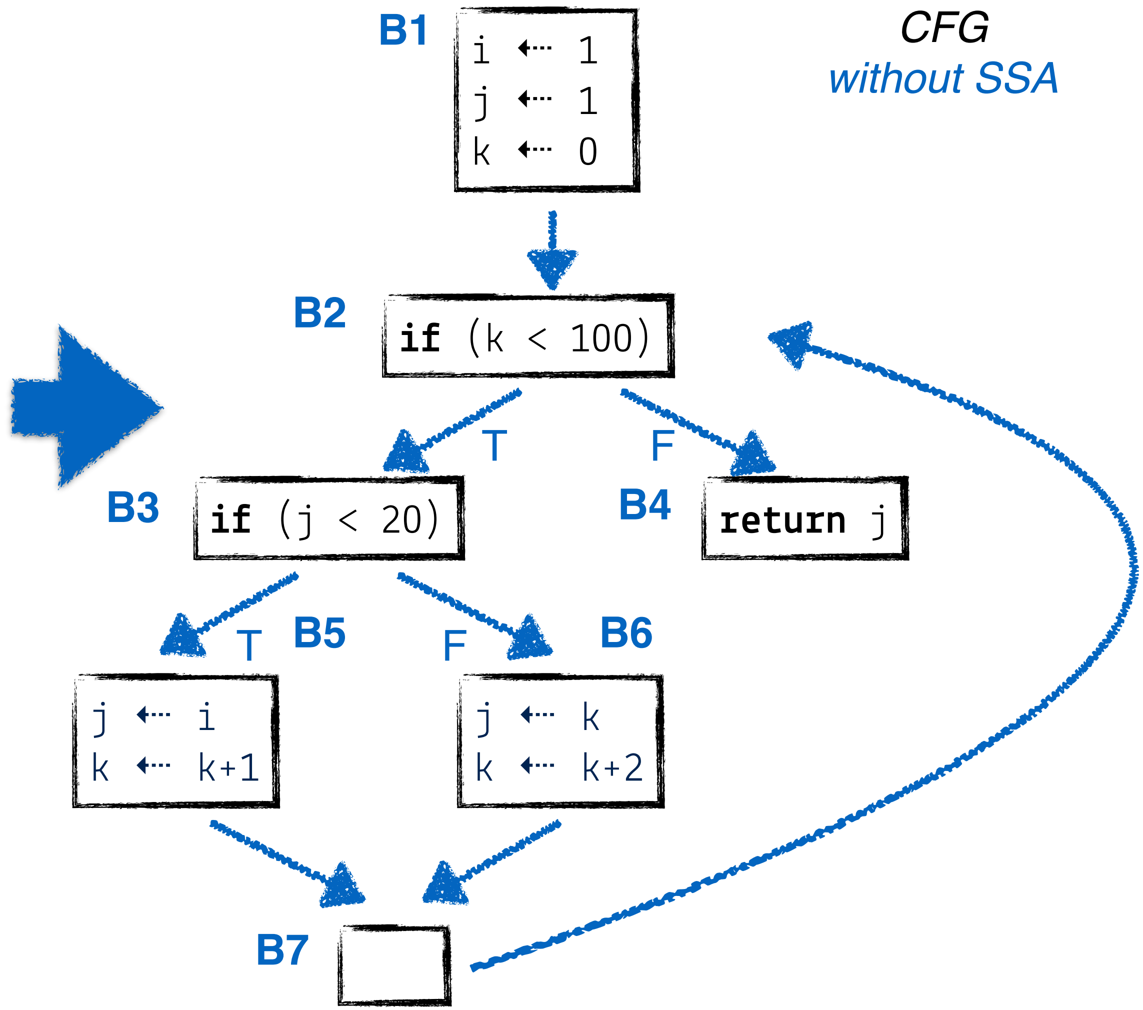
Intermediate code

Source code

```
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;
```

Example from [2]



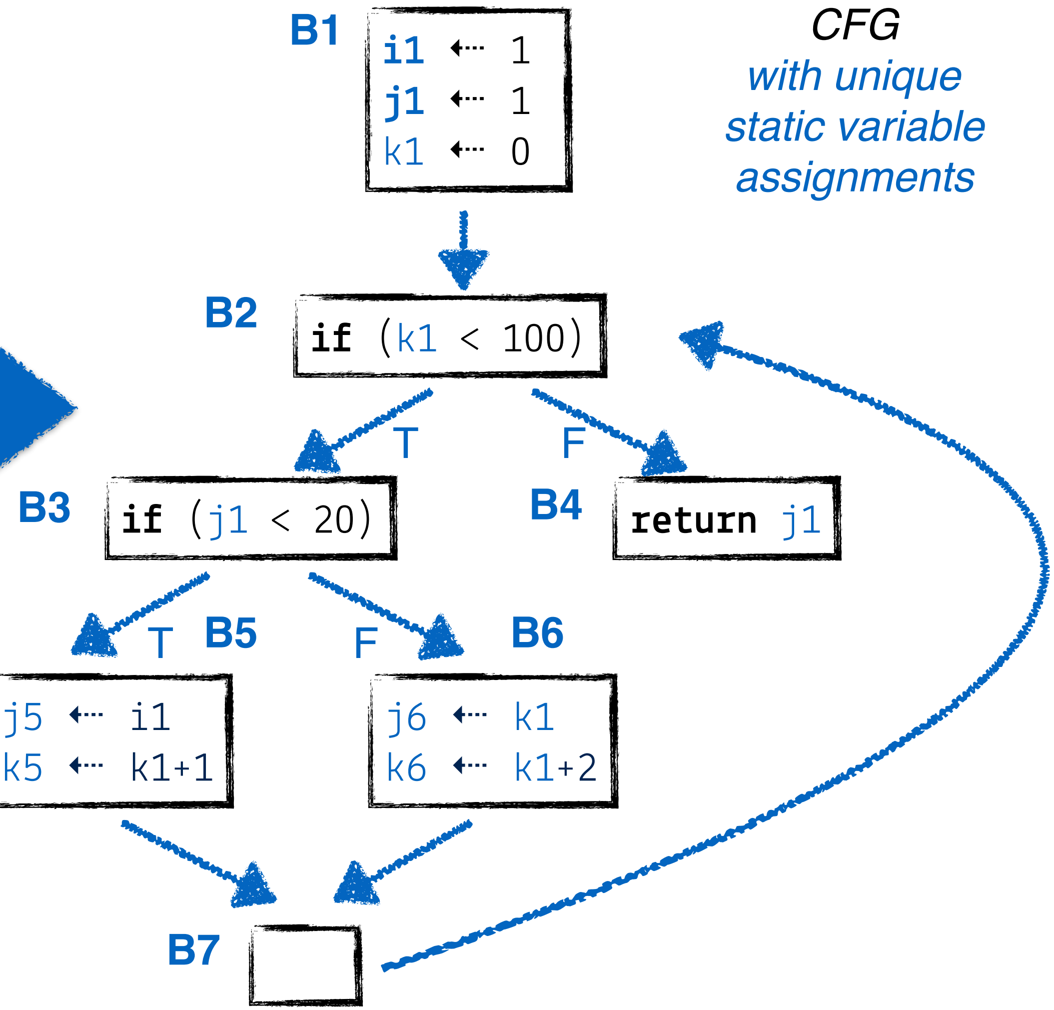
Unique Identifiers: Naive Approach

Intermediate code

Source code

```
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;
```



CFG
with unique static variable assignments

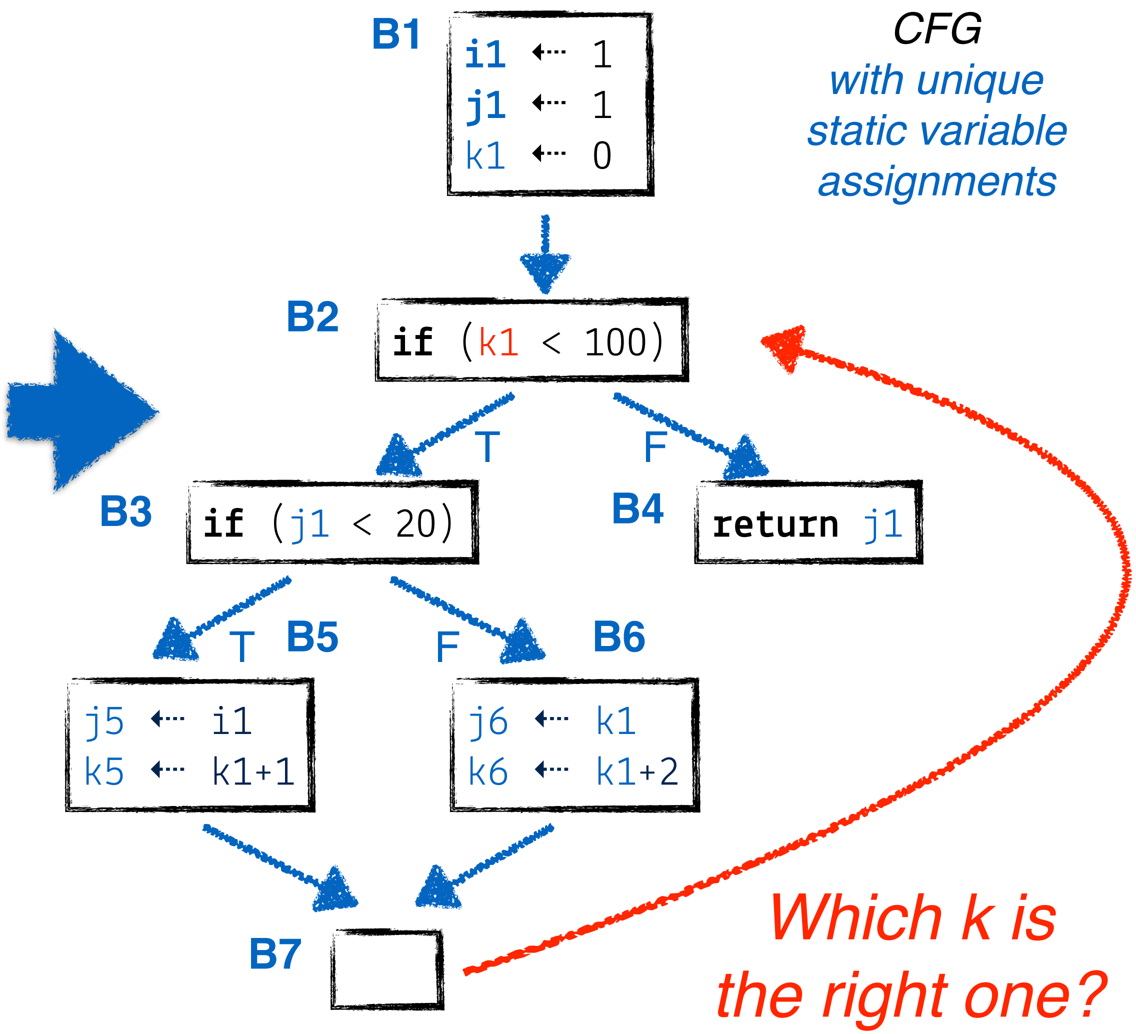
Problem with the Naive Approach

Intermediate code

Source code

```
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;
```



Fixing the Variable Problem

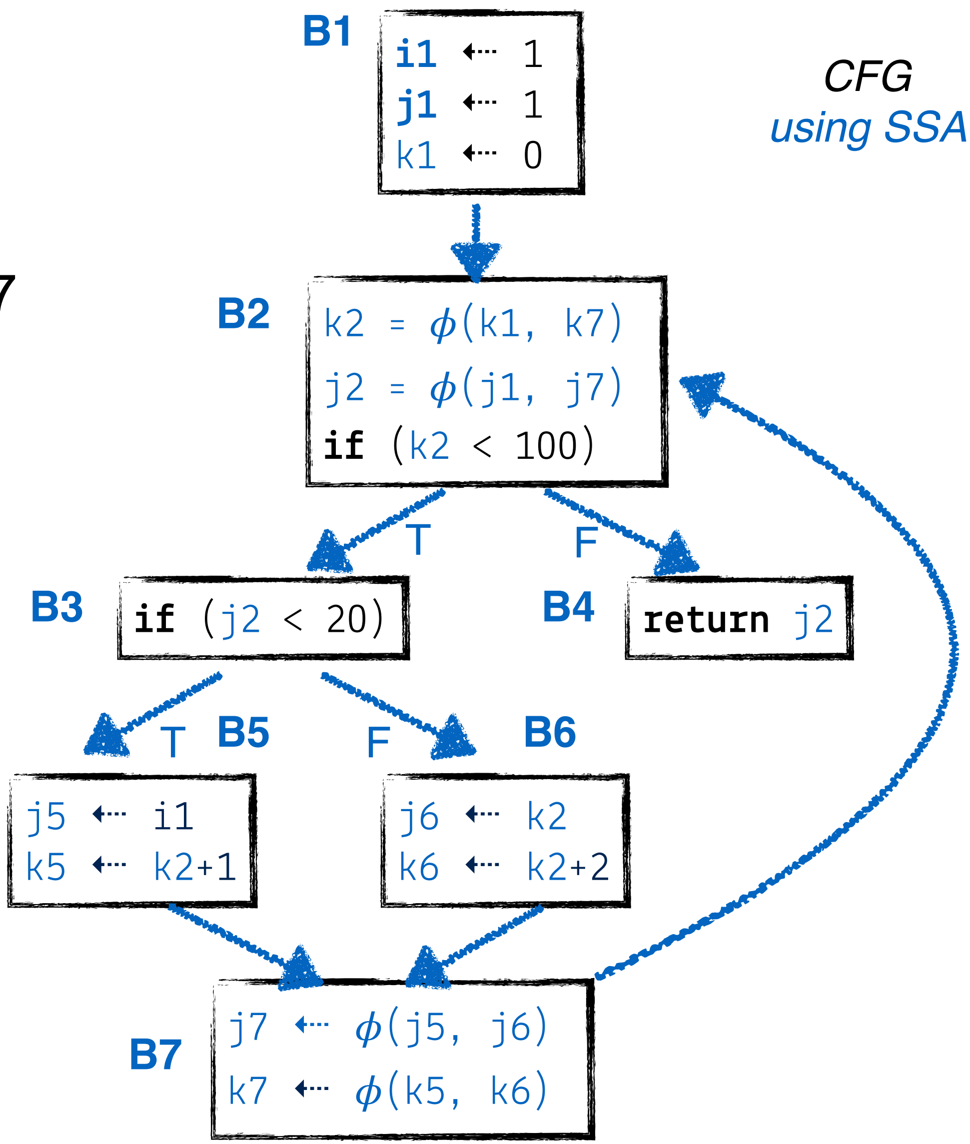
Intermediate code

"Which k is the right one?"
"It depends…"

- Basic block B2 can receive values for k from B1 and B7
- Similar for variable **j**
- Fix: introduce a *selector function ϕ (phi)* that copies the correct value to a new intermediate variable depending on the control flow:

```
k2 = ϕ(k1, k7)
j2 = ϕ(j1, j7)
```

CFG
using SSA

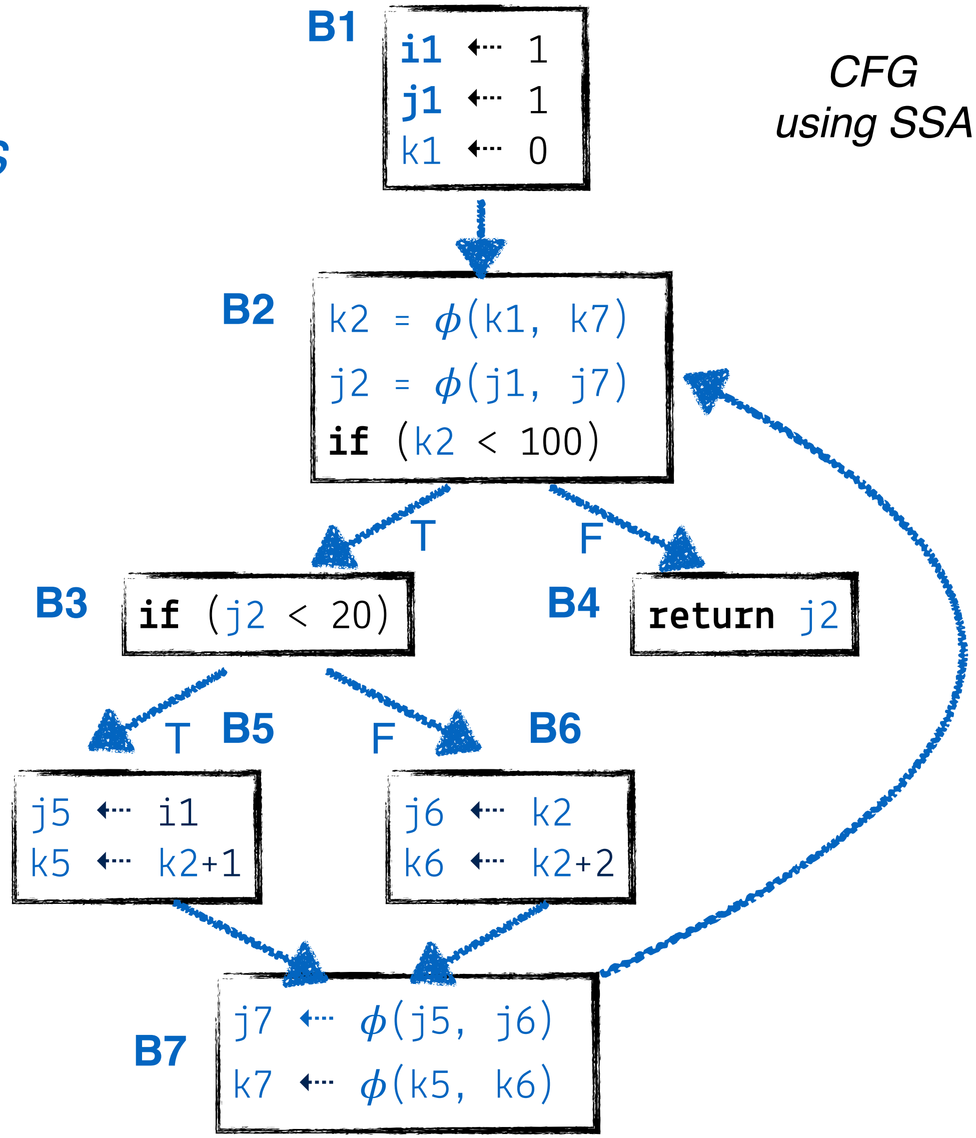


Placement of Phi Functions

Intermediate code

The minimal number and placement of phi functions is more complex than in this simple example

- Generation of *minimal SSA*
- Use of *dominance frontiers* to determine the basic block defining the current value of a variable
- See [3] for details



CFG using SSA

What's next?

Intermediate code

- The procedure abstraction

References

[1] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. K. Zadeck (1991). Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems. 13 (4): 451–490

[2] Andrew W. Appel (1998). SSA is Functional Programming. ACM SIGPLAN Not. 33, 4 (April 1998), 17-20

[3] Cooper, Keith D.; Harvey, Timothy J.; Kennedy, Ken (2001). A Simple, Fast Dominance Algorithm. Softw. Pract. Exper. 2001; 4:1–10