EECS 483: Compiler Construction

Lecture 5:
Conditionals 2

January 29, 2025

Conditionals and Continuations

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

We need to also account for the **continuation** of the if expression!

The continuation is what should happen **after** the result of the expression is computed. Now that result might be computed in either branch.

So the continuation needs to be run after **either branch**

Compiling Conditionals by Copying Continuations

```
if cond:
  thn
else:
  els
```

→

```
thn%uid:
  ... thn code
els%uid':
  ... els code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Copying Continuations

```
if cond:
  thn
else:
  els
```

+

```
... continuation code
```

➡

```
thn%uid:
  ... thn code
  ... continuation code
els%uid':
  ... els code
  ... continuation code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Copying Continuations

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

→

```
entry(y%5):
  thn%0:
    x%2 = 5
    res%3 = x%2 * x%2
    ret res%3
  els%1:
    x%4 = 6
    res%3 = x%4 * x%4
    ret res%3
  cbr y%5 thn%0 els%1
```

Compiling Conditionals by Copying Continuations

Strategy:

Make basic blocks for thn and els, giving them unique label names, compiling them recursively

Compile cond, do a conditional branch on the result, using the label names generated for thn and els

For continuations: copy them into both branches

For next time:

The strategy we've described today does create "correct" code.

Why is the strategy completely infeasible in practice?

Exponential Blowup in Copying Continuations

```
def main(y):
  let x = if y: 5 else: 6 in
  let x = if y: x else: add1(x) in
  let x = if y: x else: add1(x) in
  x * x
```

If we copy the continuation each time we perform an if, how many times does the

x * x

code appear in the generated ssa program?

```
entry(y%0):
  thn#4():
    x%1 = 5
    thn#2():
      x%2 = x%1
      thn#0():
        x%3 = x%2
        *_0%4 = x%3
        *_1%5 = x%3
        result%6 = *_0%4 * *_1%5
        ret result%6
      els#1():
        add1_0%8 = x%2
        x%3 = add1_0%8 + 1
        *_0%4 = x%3
        *_1%5 = x%3
        result%6 = *_0%4 * *_1%5
        ret result%6
      cond%7 = y%0
      cbr cond%7 thn#0 els#1
    els#3():
      add1_0%10 = x%1
      x%2 = add1_0%10 + 1
      thn#0():
        x%3 = x%2
        *_0%4 = x%3
        *_1%5 = x%3
        result%6 = *_0%4 * *_1%5
        ret result%6
      els#1():
        add1_0%8 = x%2
        x%3 = add1_0%8 + 1
        *_0%4 = x%3
        *_1%5 = x%3
        result%6 = *_0%4 * *_1%5
        ret result%6
      cond%7 = y%0
      cbr cond%7 thn#0 els#1
    cond%9 = y%0
    cbr cond%9 thn#2 els#3
  els#5():
    x%1 = 6
    thn#2():
      x%2 = x%1
      thn#0():
        x%3 = x%2
        *_0%4 = x%3
        *_1%5 = x%3
        result%6 = *_0%4 * *_1%5
        ret result%6
      els#1():
        add1_0%8 = x%2
        x%3 = add1_0%8 + 1
        *_0%4 = x%3
        *_1%5 = x%3
        result%6 = *_0%4 * *_1%5
        ret result%6
      cond%7 = y%0
      cbr cond%7 thn#0 els#1
    els#3():
      add1_0%10 = x%1
      x%2 = add1_0%10 + 1
      thn#0():
        x%3 = x%2
        *_0%4 = x%3
        *_1%5 = x%3
        result%6 = *_0%4 * *_1%5
        ret result%6
      els#1():
        add1_0%8 = x%2
        x%3 = add1_0%8 + 1
        *_0%4 = x%3
        *_1%5 = x%3
        result%6 = *_0%4 * *_1%5
        ret result%6
      cond%7 = y%0
      cbr cond%7 thn#0 els#1
    cond%9 = y%0
    cbr cond%9 thn#2 els#3
  cond%11 = y%0
  cbr cond%11 thn#4 els#5
```

Compiling Conditionals by Copying Continuations

Why is the strategy completely infeasible in practice?

Copying continuation: code size is exponential in the number of sequenced if-expressions

Generated code should be usually be linear in the size of the input program

Most compiler passes should be linear in the size of the input program

certain program analyses are not linear, and dominate compilation time

Not Copying Continuations

```
def main(y):
  let x = if y: 5 else: 6 in
  let x = if y: x else: add1(x) in
  let x = if y: x else: add1(x) in
  x * x
```

Copying the continuation is infeasible because it causes an exponential blowup in code size.

But it **does** produce functionally correct code because it correctly identifies that the two branches share the same continuation. The best we can do with our version of SSA.

Need to add something to SSA to allow us to express that two pieces of code share the same continuation.

Join Points

How would we write this manually in assembly code without copying?

Make a new block and jump to that same block at the end of each of the branches. This "shares" the continuation without copying, using the fact that we can copy the **reference** to the code, its label, for cheap.

Join Points

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

➡

```
entry:
        cmp rdi, 0
        jne thn#0
        jmp els#1
thn#0:
        mov rax, 5
        jmp jn#2
els#1:
        mov rax, 6
        jmp jn#2
jn#2:
        imul rax, rax
        ret
```

Join Points

How can we extend our IR to express join points?

Join points are just a new kind of block?

- Make a block for the join point
- Add a new **uncdonditional** branch, like an assembly **jmp** to our IR.

Join Points

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

```
entry(y%0):
  jn#2:
    ...?
    result%4 = x%1 * x%1
    ret result%4
  thn#0:
    thn_res%6 = 5
    ...?
    br jn#2
  els#1:
    els_res%7 = 6
    ...?
    br jn#2
  cond%5 = y%0
  cbr cond%5 thn#0 els#1
```

Our ordinary blocks aren't enough: Join points aren't just code blocks, they are **continuations.** We don't just need to execute

x * x

We also need to **assign to x** differently depending on the branch

Solution 1: Assign to x in both branches

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

```
entry(y%0):
  jn#2:
    result%4 = x%1 * x%1
    ret result%4
  thn#0:
    x%1 = 5
    br jn#2
  els#1:
    x%1 = 6
    br jn#2
  cond%5 = y%0
  cbr cond%5 thn#0 els#1
```

Pros: easy to generate assembly code

Con: breaks the "static single assignment property"

It's not clear in the join point where x is defined, makes program analysis, optimization much harder

Solution 2: ϕ nodes

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

➡

```
entry(y%0):
  jn#2:
    x%1 = ϕ(thn_res%6, els_res%7)
    result%4 = x%1 * x%1
    ret result%4
  thn#0:
    thn_res%6 = 5
    br jn#2
  els#1:
    els_res%7 = 6
    br jn#2
  cond%5 = y%0
  cbr cond%5 thn#0 els#1
```

Solution 2: ϕ nodes

A ϕ node is a "ϕony" operation that allows SSA format to express join points without breaking the SSA property.

```
x = ϕ(x1,x2,x3,...)
```

The semantics is a little strange...The ϕ node is an assignment to x, but which variable it assigns depends on where we **just** branched **from.**

ϕ nodes require some syntactic restrictions:

- they can only appear at the beginning of a basic block (so that we just branched).
- need to make sure that the variables on the rhs are actually defined before they reach the ϕ node.
- need to pick some kind of ordering, so we actually know which variable corresponds to which branch

Solution 2: ϕ nodes

A ϕ node is a "ϕony" operation that allows SSA format to express join points without breaking the SSA property.

```
x = ϕ(x1,x2,x3,...)
```

Pros: maintains the SSA property, popular in SSA literature, used in long-established industrial SSA-based compilers (LLVM, GCC, Hotspot)

Cons: strange semantics, strange code generation (the move happens in the predecessor block!), difficult to enforce syntactic restrictions

Solution 3: Parameterized Blocks

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

```
entry(y%0):
  jn#2(x%1):
    result%4 = x%1 * x%1
    ret result%4
  thn#0():
    br jn#2(5)
  els#1():
    br jn#2(6)
  cond%5 = y%0
  cbr cond%5 thn#0() els#1()
```

Represent the **continuation** directly in the syntax: a block can have **parameters** just like a continuation has an input variable.

Directly allow us to turn continuations into blocks

Solution 3: Parameterized Blocks

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

```
entry(y%0):
  jn#2(x%1):
    result%4 = x%1 * x%1
    ret result%4
  thn#0():
    br jn#2(5)
  els#1():
    br jn#2(6)
  cond%5 = y%0
  cbr cond%5 thn#0() els#1()
```

Represent the **continuation** directly in the syntax: a block can have **parameters** just like a continuation has an input variable.

Directly allow us to turn continuations into blocks

Solution 3: Parameterized Blocks

```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```

➡

```
entry(y%0):
  jn#2(x%1):
    result%4 = x%1 * x%1
    ret result%4
  thn#0():
    br jn#2(5)
  els#1():
    br jn#2(6)
  cond%5 = y%0
  cbr cond%5 thn#0() els#1()
```

Represent the **continuation** directly in the syntax: a block can have **parameters** just like a continuation has an input variable.

Directly allow us to turn continuations into blocks

ϕ Nodes vs Parameterized Blocks

A parameterized block adds "arguments" to our basic blocks

```
l(x1,x2,x3):
```

These arguments are like other variables, they are in scope for the block, but not outside of it.

Branching to a parameterized block means providing arguments to it

```
br l(y1,y2,y3)
```

Pros: maintains the SSA property, simple code generation, simple well-formedness condition, used in newer SSA-based compilers (Swift, MLIR, MLton)

Cons: separates the different join points syntactically in the SSA program, need to translate most SSA papers from phi node notation

ϕ Nodes vs Parameterized Blocks

```
entry(y%0):
  jn#2:
    x%1 = ϕ(thn_res%6, els_res%7)
    result%4 = x%1 * x%1
    ret result%4
  thn#0:
    thn_res%6 = 5
    br jn#2
  els#1:
    els_res%7 = 6
    br jn#2
  cond%5 = y%0
  cbr cond%5 thn#0 els#1
```

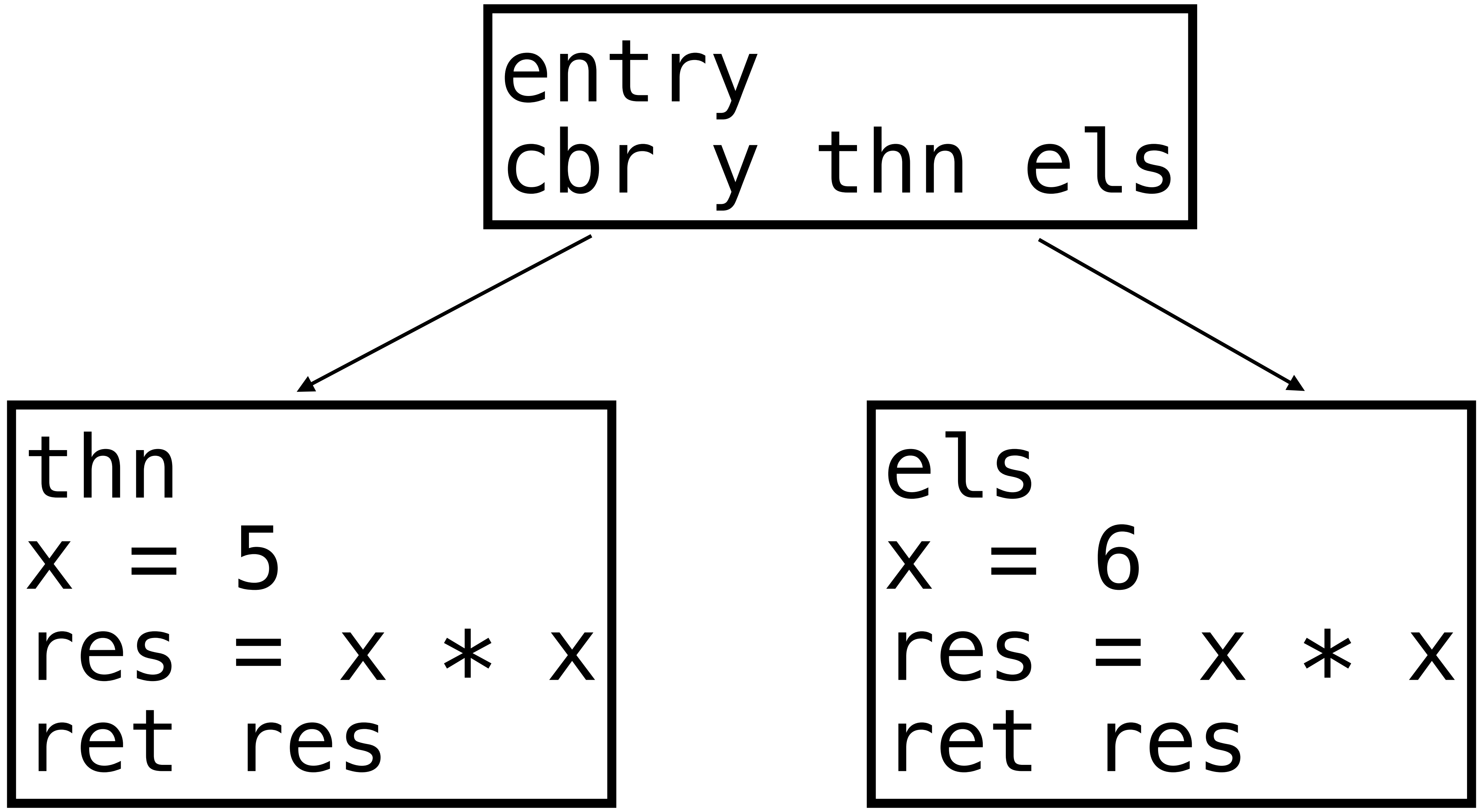
```
entry(y%0):
  jn#2(x%1):
    result%4 = x%1 * x%1
    ret result%4
  thn#0():
    br jn#2(5)
  els#1():
    br jn#2(6)
  cond%5 = y%0
  cbr cond%5 thn#0() els#1()
```

ϕ nodes put assignment in the block itself, parameterized blocks put the "asignment in the predecessor

Control Flow Graph

We can visualize SSA programs using **control-flow graphs**.

```
entry(y%5):
  thn%0:
    x%2 = 5
    res%3 = x%2 * x%2
    ret res%3
  els%1:
    x%4 = 6
    res%3 = x%4 * x%4
    ret res%3
  cbr y%5 thn%0 els%1
```

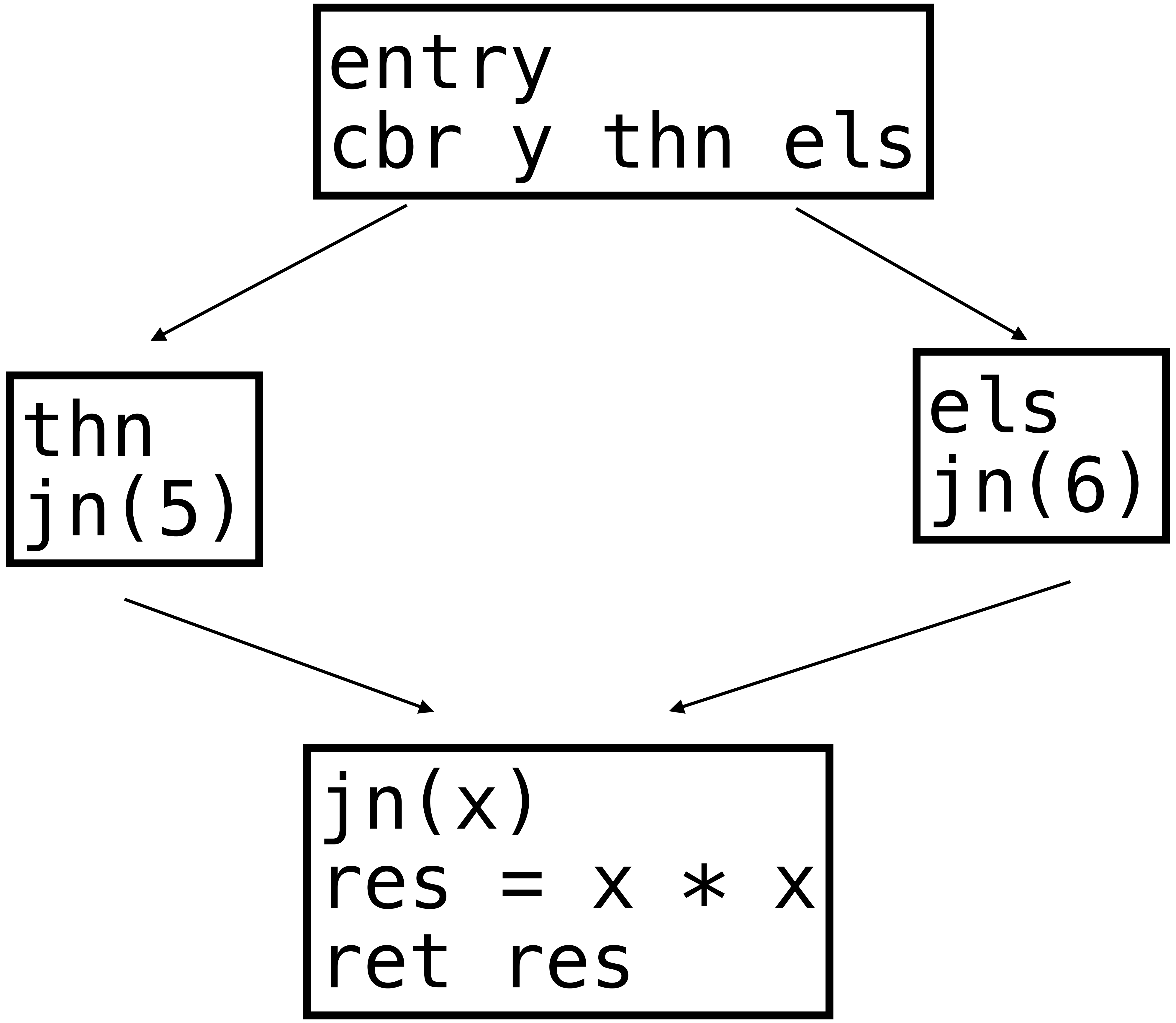


Nodes of CFG: basic blocks
edges are branches

Control Flow Graph

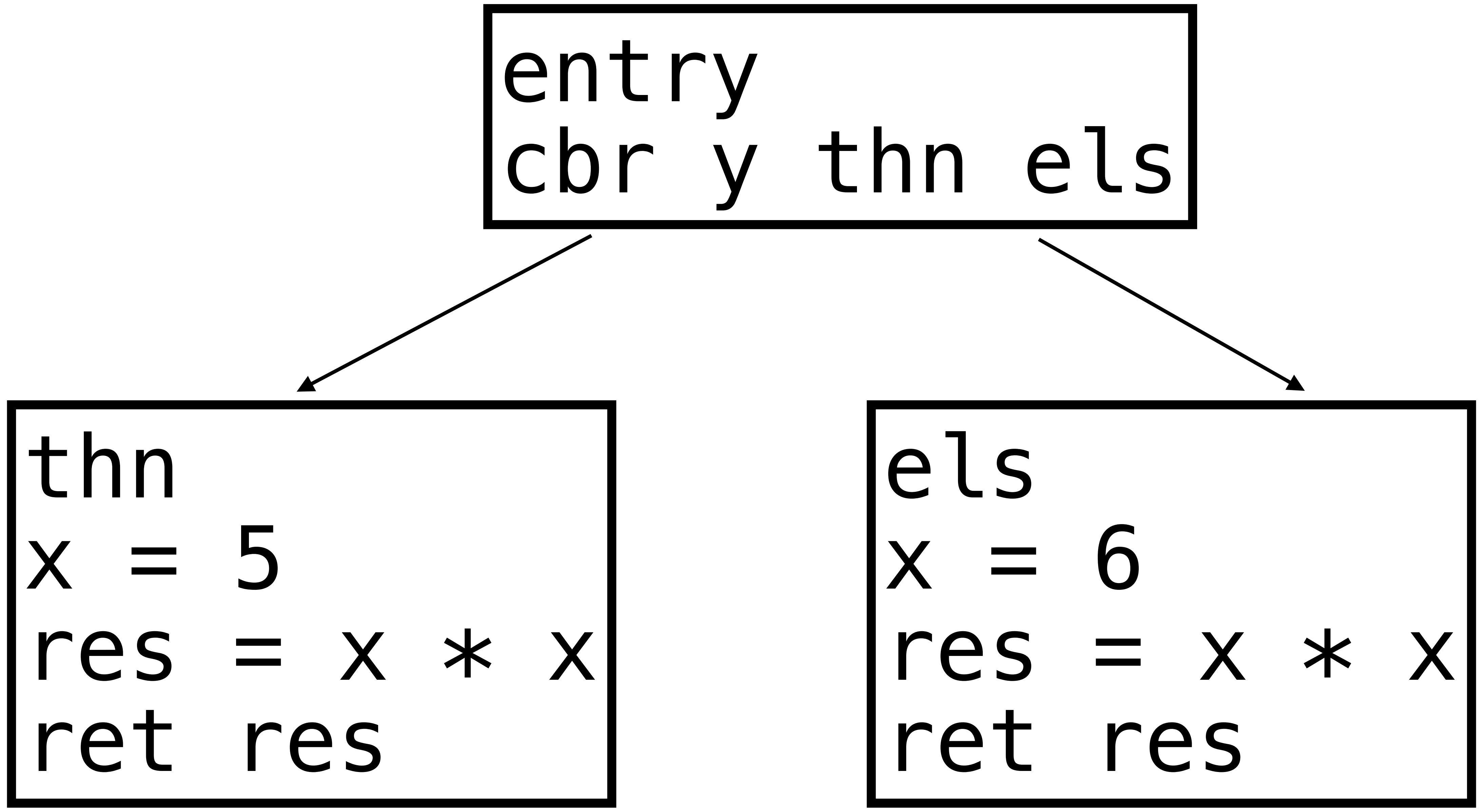
We can visualize SSA programs using **control-flow graphs**.
Join point: multiple predecessors

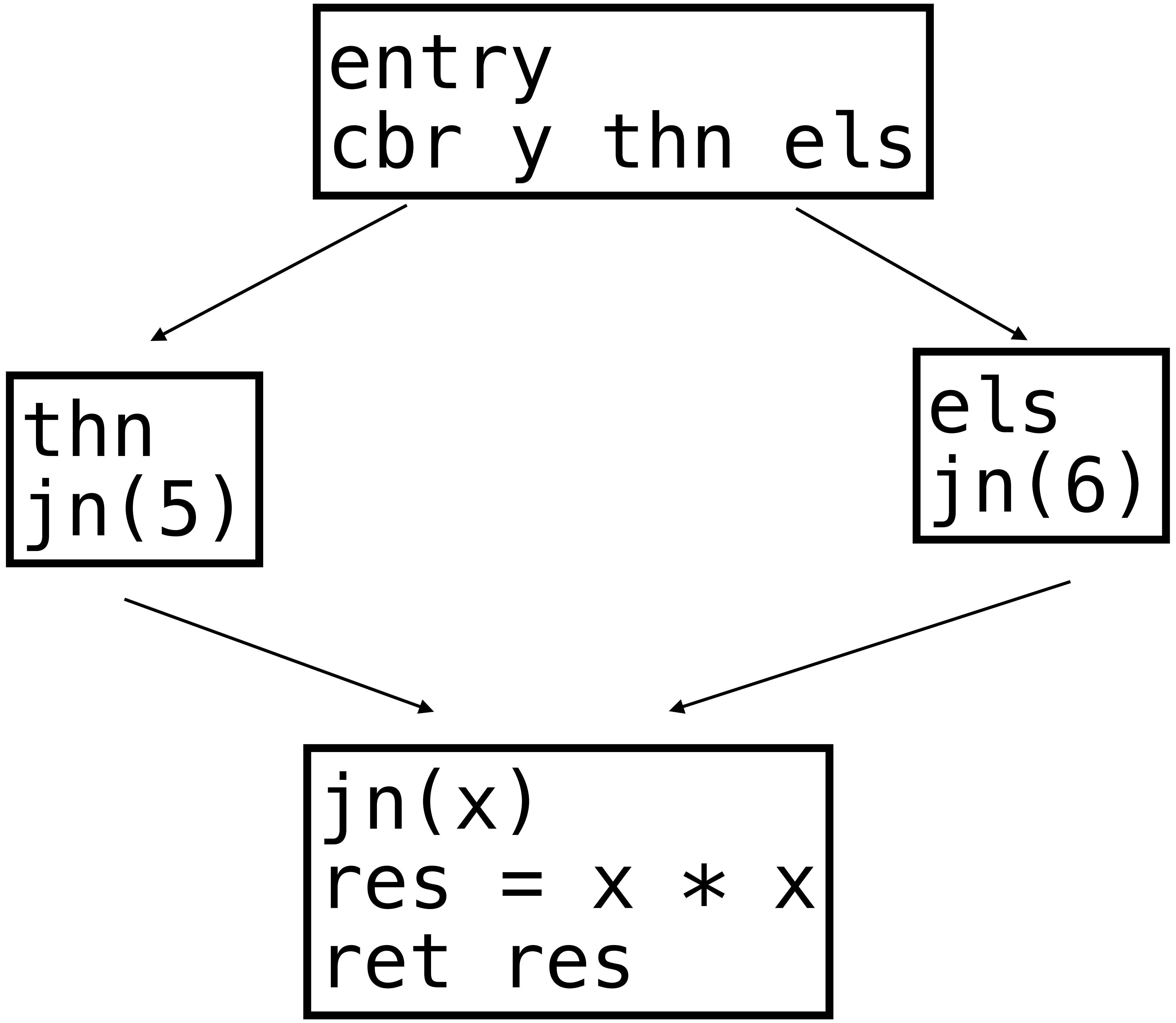
```
entry(y%0):
  jn#2(x%1):
    result%4 = x%1 * x%1
    ret result%4
  thn#0():
    br jn#2(5)
  els#1():
    br jn#2(6)
  cond%5 = y%0
  cbr cond%5 thn#0() els#1()
```



Control Flow Graph

Join points are needed to express **sharing**. Conditional code like our source produces a DAG. DAGs can be simulated with trees, but with an exponential blowup!





Control Flow Graph

A common way to think about SSA programs is in terms of **control-flow graphs**.

With branching, but no join points, we can express control-flow **trees**.

Join points allow us to express control-flow **DAGs** which can be exponentially more compact than trees**.**

If we remove the acyclicity requirement, we can express **loops** and even more exotic control flow. Revisit this next week

SSA Abstract Syntax

```
pub enum BlockBody {
    Terminator(Terminator),
    Operation {
        dest: VarName,
        op: Operation,
        next: Box<BlockBody>
    },
    SubBlocks {
        blocks: Vec<BasicBlock>,
        next: Box<BlockBody>
    },
}
```

```
pub enum Terminator {
    Return(Immediate),
    Branch(Branch),
    ConditionalBranch {
        cond: Immediate,
        thn: Label,
        els: Label
    },
}
```

```
pub struct BasicBlock {
    pub label: Label,
    pub params: Vec<VarName>,
    pub body: BlockBody,
}
```

```
pub struct Branch {
    pub target: Label,
    pub args: Vec<Immediate>,
}
```

Well-formedness of SSA Programs

A benefit of sub-blocks and parameterized blocks is that we have a similar notion of **scope** that we do in our Snake language.

Sub-blocks declare the names of blocks: those blocks should only be used within the body of the sub-block declaration

Operations and Basic blocks declare the names of variables: those should only be used within the body of the block after the declaration.

We can adapt our scope checker from the Snake language AST to the SSA programs. Gives us a "linting" pass that can help us find bugs if we accidentally made ill-formed SSA programs. If we implemented our compiler correctly, this should always succeed, but can be helpful for debugging.

Compiling Conditionals by Copying Continuations

```
if cond:
   thn
else:
   els
```

+

```
... continuation code
```

```
thn%uid:
  ... thn code
  ... continuation code
els%uid':
  ... els code
  ... continuation code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Generating Joins

```
if cond:
   thn
else:
   els
```

+

```
... continuation code
```

```
jn%uid''(x): ; continuation parameter
  ... continuation code
thn%uid:
  ... thn code
  br jn%uid''(thn_res)
els%uid':
  ... els code
  br jn%uid''(els_res)
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

If the continuation is small (i.e., just a ret), copying would be better

Code Generation for Branch with Arguments

```
l(x1,x2,x3):
  ...
br l(imm1,imm2,imm3)
```

➡

```
mov rax, imm1
mov [rsp - offset(x1)], rax
mov rax, imm2
mov [rsp - offset(x2)], rax
mov rax, imm3
mov [rsp - offset(x3)], rax
jmp l
```

In compiling the conditional branch, need to know where the arguments for the label are stored. Keep track of this information in an environment you build up as you see sub-block declarations.

Alternate Approach: "SSA Destruction"

```
l(x1,x2,x3):
  ...
br l(imm1,imm2,imm3)
```

Used in most industry SSA compilers to squeeze out the best possible code generation:

more intermediate IRs =~ more opportunities for optimization

➡

```
l(x1,x2,x3):
  ...
x1 = imm1
x2 = imm2
x3 = imm3
br l
```

➡

```
mov rax, imm1
mov [rsp - offset(x1)], rax
mov rax, imm2
mov [rsp - offset(x2)], rax
mov rax, imm3
mov [rsp - offset(x3)], rax
jmp l
```

Should Conditional Branches be allowed to have arguments?

```
cbr x l1 l2
```

➡

```
mov rax, [rsp - offset(x)]
cmp rax, 0
jne l1
jmp l2
```

Should Conditional Branches be allowed to have arguments?

```
l1(v1,v2):
...
l2(w):
...
cbr x l1(y1,y2) l2(z)
```

➡

```
mov rax, [rsp - offset(x)]
cmp rax, 0
mov rax, [rsp - offset(y1)]
mov [rsp - offset(v1)], rax
mov rax, [rsp - offset(y1)]
mov [rsp - offset(v1)], rax
jne l1
mov rax, [rsp - offset(z)]
mov [rsp - offset(w)], rax
jmp l2
```

unnecessary movs if the else branch is taken

Should Conditional Branches be allowed to have arguments?

```
l1(v1,v2):
...
l2(w):
...
cbr x l1(y1,y2) l2(z)
```

➡

```
l1(v1,v2):
...
l2(w):
...
l1b():
  l1(y1,y2)
l2b():
  l2(z)
cbr x l1b l2b
```

SSA-to-SSA transformation can eliminate them

Join Points

Summary:

Join points are needed when different code paths share a common continuation.

Express sharing by duplicating a reference to the continuation, rather than the code for the continuation itself

SSA handles join points using either ϕ nodes or block arguments. Equivalent approaches but different ergonomics.

Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?
2. What is the semantics of the language we are compiling?
3. How can we implement that semantics in assembly code?
4. How should we adapt our intermediate representation to new features?
5. How can we generate assembly code from the IR?

Snake v0.2: "Boa"

Last time we added conditionals, but we only have integer operations so far. Let's add logical operators to write more interesting programs.

Snake v0.2: "Boa"

⟨*prim1*⟩: ... | `!`

⟨*prim2*⟩: ... | `&&` | `||` | `<` | `<=` | `>` | `>=` | `==` | `!=`

⟨*expr*⟩: ... | `true` | `false`

Abstract Syntax

```
enum Prim {
    ...
    // unary
    Not
    // binary
    ...
    And,
    Or,
    Lt,
    Leq,
    Gt,
    Geq,
    Eq,
    Neq,
}
```

```
enum Expression {
    ...
    Bool(bool)
}
```

Examples

```
def main(x):
  if x >= 4 && x < 7 :
    x
  else:
    0
```

Semantics

```
true == 1

false == 0

if 5: ... else: ...

true * 7

5 || 3
```

Semantics

Multiple approaches to handling datatypes:

1. Statically rule these out: integers and booleans are considered different and disjoint, reject programs like these

2. Statically insert coercions: integers and booleans are different but related, add coercions back and forth when mixed

3. Dynamically checks: integers and booleans are different and disjoint, error at runtime if we encounter these programs

4. Dynamic coercions: variables can be any type, insert coercions on all boolean operations

x86 Instructions: setcc

setcc loc

Actually a family of instructions, where **cc** is a **condition code**

Semantics: sets the lowest bit of loc **to the result of the condition code**

Peculiarity: loc in this case needs to be a 1-byte register.

```
0xXX XX XX XX XX XX XX XX
                    ah  al
                    ------
                      ax
                ----------
                   eax
--------------------------
           rax
```

x86 Instructions: setcc

setcc loc

Actually a family of instructions, where **cc** is a **condition code**

Semantics: sets the lowest bit of loc **to the result of the condition code**

Peculiarity: loc in this case needs to be a 1-byte register.

```
mov rax, 0
setge al
```

sets rax to 1 if the condition code ge is set, otherwise 0

x86 Instructions: bitwise operators

and dest, src

or dest, src

bitwise and, or. Not quite what we want for logical operations

```
mov rax, 0xF0
mov rcx, 0x0F
and rax, rcx
```

`rax` is 0, not 1

Coercions and Representation

Booleans

- true is 1
- false is 0

Integers

- any 64-bit value

Integer to boolean: everything non-zero to 1, zero to 0

Boolean to integer: true to 1, false to 0

Implementing Coercions

Can implement coercions as the assembly or SSA level

1. Assembly level: coerce inputs to booleans before all logical operations

2. SSA level: add a coercion `intToBool` to SSA that is implemented by the assembly coercion

   advantage: can be removed by optimizations

   advantage: simplifies code generation

Lowering to SSA

```
true    →  1

false   →  0

x && y  →  b = intToBool(x)
           c = intToBool(y)
           res = b && c
```

SSA to x86

```
x = intToBool(y)
```

→

```
mov rax, [rsp - off(y)]
cmp rax, 0
setne al
mov [rsp - off(x)], rax
```

SSA to x86

```
x = y & z
```

→

```
mov rax, [rsp - off(y)]
mov r10, [rsp - off(z)]
and rax, r10
mov [rsp - off(x)], rax
```

Summary

Implement a **coercions** from integers to booleans before performing the operation

a) Implement the coercion in the code generation phase from SSA to x86, insert it into each operation

b) SSA remains untyped, oblivious to our high-level type distinctions: all values are just 64-bits.