EECS 483: Compiler Construction

Lecture 5:
Parameterized Blocks, Booleans and Tail Calls

January 29, 2025

Reminders

- Assignment 1 is due on Friday, the 30th.

  Office hours Wednesday and Friday (Yuchen) and Thursday (Max)
- Next assignment to be released on Monday, February 2nd.

Learning Objectives

- Understand the need for join points, parameterized blocks and code generation

- Understand design choices when incorporating multiple data types into a language

- How to efficiently compile tail-calls to SSA parameterized blocks and assembly code.

Conditionals and Continuations

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

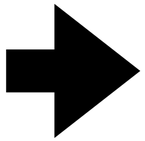
We need to also account for the **continuation** of the if expression!

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
```



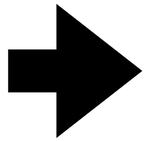
```
thn%uid:
  ... thn code
els%uid':
  ... els code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Copying Continuations

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

+

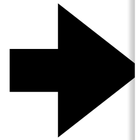
```
... continuation code
```



```
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
```

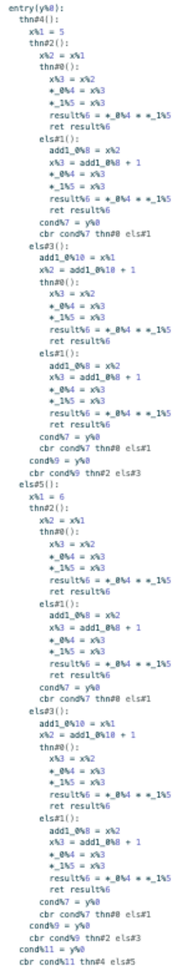
Exponential Blowup in Copying Continuations

```
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?



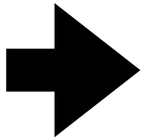
Join Points

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

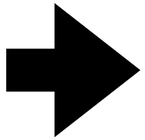
```
def main(y):
  let x = (if y: 5 else: 6) in
  x * x
```



```
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
```

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

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

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.

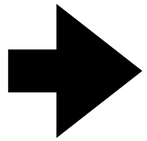
We can adapt our scope checker from the Snake language AST to the SSA programs. Gives us a "linting" pass that can help us find bugs if we accidentally made ill-formed SSA programs. If we implemented our compiler correctly, this should always succeed, but can be helpful for debugging.

Compiling Conditionals by Copying Continuations

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

+

```
... continuation code
```



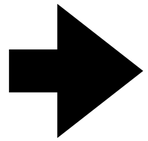
```
thn%uid:
  ... thn code
  ... continuation code
els%uid':
  ... els code
  ... continuation code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Generating Joins

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

+

```
... continuation code
```

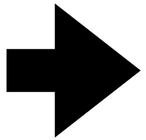


```
jn%uid''(x): ; continuation parameter
  ... continuation code
thn%uid:
  ... thn code
  br jn%uid''(thn_res)
els%uid':
  ... els code
  br jn%uid''(els_res)
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

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
```

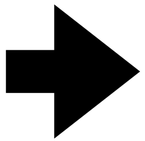
In compiling the conditional branch, need to know where the arguments for the label are stored. Keep track of this information in an environment you build up as you see sub-block declarations.

Alternate Approach: "SSA Destruction"

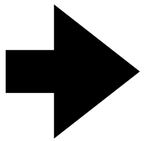
```
l(x1,x2,x3):
  ...
br l(imm1,imm2,imm3)
```

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

more intermediate IRs =~ more opportunities for optimization



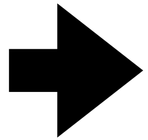
```
l(x1,x2,x3):
  ...
x1 = imm1
x2 = imm2
x3 = imm3
br l
```



```
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?

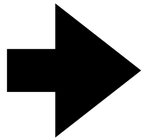
```
cbr x l1 l2
```



```
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)
```

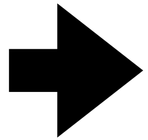


```
mov rax, [rsp - offset(x)]
cmp rax, 0
mov rax, [rsp - offset(y1)]
mov [rsp - offset(v1)], rax
mov rax, [rsp - offset(y1)]
mov [rsp - offset(v1)], rax
jne l1
mov rax, [rsp - offset(z)]
mov [rsp - offset(w)], rax
jmp l2
```

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 reject type mixing: 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 "strong dynamic typing": integers and booleans are different and disjoint, error at runtime if we encounter one where the other is required

4. Dynamic coercions"weak dynamic typing": variables can be any type, insert coercions on inputs to typed operations

For now we will implement d**ynamic coercions,** later in the semester we will implement **dynamic checks**

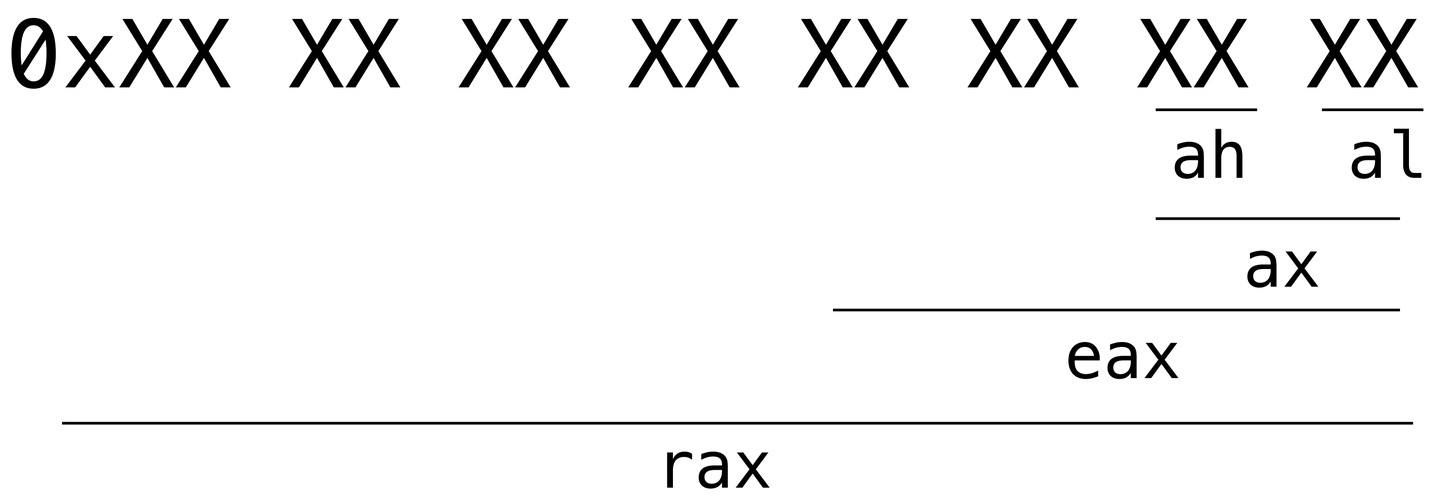
x86 Instructions: setcc

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.



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

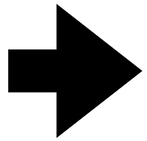
```
true  ➡  1

false ➡  0

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

SSA to x86

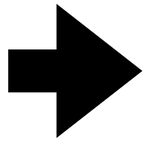
```
x = intToBool(y)
```



```
mov rax, [rsp - off(y)]
cmp rax, 0
mov 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.

Extending the Snake Language

So far:

- Adder: straightline sequence of operations
- Boa so far: control-flow DAGs

Next:

- cyclic control-flow graphs
- computational power: finite automata

Cyclic Control Flow in Assembly and SSA

live code

Extending the Snake Language

What source-level programming features would allow us to express cyclic control-flow graphs?

1. Functional: recursive functions, tail calls

2. Imperative: while/for loops, mutable variables

We'll look at these each in turn and study how to compile them to SSA.

Extending the Snake Language

What source-level programming features would allow us to express cyclic control-flow graphs?

1. Functional: recursive functions, tail calls

2. Imperative: while/for loops, mutable variables

We'll look at these each in turn and study how to compile them to SSA.

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?

Extending the Snake Language

⟨*decls*⟩:
- ⟨decl⟩
- ⟨decls⟩ `and` ⟨decl⟩

⟨*decl*⟩:
- `def` ***IDENTIFIER*** `(` ⟨ids⟩ `)` `:` ⟨expr⟩
- `def` ***IDENTIFIER*** `(` `)` `:` ⟨expr⟩

⟨*ids*⟩:
- ***IDENTIFIER***
- ***IDENTIFIER*** `,` ⟨ids⟩

⟨*expr*⟩:
- `...`
- ***IDENTIFIER*** `(` ⟨exprs⟩ `)`
- ***IDENTIFIER*** `(` `)`
- ⟨decls⟩ `in` ⟨expr⟩

⟨*exprs*⟩: ⟨expr⟩ | ⟨expr⟩ `,` ⟨exprs⟩

Extending the Snake Language

```
pub enum Expr {
    ...
    FunDefs {
        decls: Vec<FunDecl>,
        body: Box<Expr>,
    },
    Call {
        fun_name: Fun,
        args: Vec<Expr>,
    },
}

pub struct FunDecl {
    pub name: String,
    pub parameters: Vec<String>,
    pub body: Expr,
}
```

Examples

recursion

Function definitions are recursive: the function is in scope within its own body as well as in the body of the continuation of its definition

```
def fac(x):
  def loop(x, acc):
    if x == 0:
      acc
    else:
      loop(x - 1, acc * x)
  in
  loop(x, 1)
in
fac(10)
```

Examples

mutual recursion

Function definitions separated by an and are **mutually recursive.** Mutually recursive functions are all in scope of each other.

```
def even(x):
  def evn(n):
    if n == 0:
      true
    else:
      odd(n - 1)
  and
  def odd(n):
    if n == 0:
      false
    else:
      even(n - 1)
  in
  if x >= 0:
    evn(x)
  else:
    evn(-1 * x)
in
even(24)
```

Examples

variable capture

Function definitions can access variables in scope at their definition site.

```
def pow(m, n):
  def loop(n, acc):
    if n == 0:
      acc
    else:
      loop(n - 1, acc * m)
  in
  loop(n, 1)
```

First-order vs Higher-order Functions

In first-order programming languages, we can have function **definitions** but functions cannot be passed around as values

In higher-order programming languages, functions can be passed as values, returned from functions/expressions etc.

For now: first-order, return to higher-order later in the semester.

Function Names

Since functions cannot be values, treat them as a separate namespace.

Allow shadowing of function names, like variable declarations. Similarly, resolve all function names to unique identifiers.

Arity-Checking

If functions are first-order, we can always resolve a function call to its definition site. So we can determine if the function is called with the right number of arguments statically. Produce an error if the function is called with the wrong number of arguments

Arity-Checking

If functions are first-order, we can always resolve a function call to its definition site. So we can determine if the function is called with the right number of arguments statically. Produce an error if the function is called with the wrong number of arguments

```
def f(x,y,z):
...
in
...
f(a,b)
```

Overloading

Should we allow this call?

shadowing: the inner f wins

but we can resolve the disambiguity based on static information

```
def f(x,y,z):
...
in
def f(x,y):
...
in
f(x,y,z)
```

Functions as Blocks

When can a function call be compiled to a branch with arguments?

When it is in **tail position**, i.e., the result of the called function is immediately returned by the caller.

If this is the case, the call can be compiled directly to a branch.

Otherwise it is a **true call** and implementing it requires storing data on the call stack. Revisit this next week

Tail Position

```
def fac(x):
  def loop(x, acc):
    if x == 0:
      acc
    else:
      loop(x - 1, acc * x)
  in
  loop(x, 1)
in
fac(10)
```

```
def factorial(x):
  if x == 0:
    1
  else:
    x * factorial(x - 1)
in
factorial(6)
```

Tail Position

When is an expression in **tail position?**

- It depends on the **context**, not the expression itself

Tail Position

```
pub struct Prog<Var, Fun> {
    pub param: (Var, SrcLoc),
    pub main: Expr<Var, Fun>,
}
```

The **main** expression is in tail position, as its result is the result of the main function

Tail Position

```
Prim {
    prim: Prim,
    args: Vec<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

```
Call {
    fun: Fun,
    args: Vec<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

The **args** of a prim or a call are **never** in tail position, as we always have to do something else after evaluating them (the prim/call)

Tail Position

```
Let {
    bindings: Vec<Binding<Var, Fun>>,
    body: Box<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

The expressions in the **bindings** are **never** in tail position, as we always have to do something else after evaluating them (the let body)

The **body** of the let is in tail position if the let itself is in tail position

Tail Position

```
If {
    cond: Box<Expr<Var, Fun>>,
    thn: Box<Expr<Var, Fun>>,
    els: Box<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

The expressions in the **cond** position is **never** in tail position, as we always have to do something else after evaluating them (the if)

The **thn** and **els** branches are in tail position if the if itself is in tail position

Tail Position

```
FunDefs {
    decls: Vec<FunDecl<Var, Fun>>,
    body: Box<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

The **body** of a fundef is in tail position if the FunDefs expression itself is in tail position

Tail Position

```
pub struct FunDecl<Var, Fun> {
    pub name: Fun,
    pub params: Vec<(Var, SrcLoc)>,
    pub body: Expr<Var, Fun>,
    pub loc: SrcLoc,
}
```

The **body** of a FunDecl is always in tail position

Function definitions to Blocks

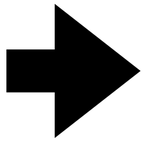
Compile each function definition directly to a corresponding block.

Compile mutually-recursive function definitions to mutually recursive blocks

Compile **tail** function calls to branch with arguments, with left-to-right evaluation order of arguments:

Tail calls to Branches

```
f(e1,e2,e3)
```



```
... ;; e1 code
x1 = ...
... ;; e2 code
x2 = ...
... ;; e3 code
x3 = ...
br f(x1,x2,x3)
```

No continuation to use
because call is assumed to be in tail position

Compiling Branch with Arguments

Semantically, a branch with arguments is a **simultaneous** move, all of the variables get updated at once.

This is not supported in our target architecture, in reality we have to sequentialize those moves into a sequence.

Compiling Branch with Arguments

Semantically, a branch with arguments is a **simultaneous** move, all of the variables get updated at once.

This is not supported in our target architecture, in reality we have to sequentialize those moves into a sequence.

Can cause correctness issues if we are not careful

Compiling Branch with Arguments

```
x = 7
f(a, b):
  z = x * a
  w = b + z
  ret w
y = x * 2
br f(5, y)
```

where is each variable stored?

```
x: rsp - 8
y: rsp - 16
a: rsp - 16
b: rsp - 24
z: rsp - 32
w: rsp - 40
```

Compiling Branch with Arguments

```
x = 7
f(a, b):
  z = x * a
  w = b + z
  ret w
y = x * 2
br f(5, y)
```

```
mov [rsp - 16], 5 ;; a = 5
mov rax, [rsp - 16]
mov [rsp - 24], rax ;; b = y
jmp f
```

Compiling Branch with Arguments

easy, sub-optimal solution

To ensure we don't overwrite memory we are about to use, we can introduce extra temporaries for the arguments.

Since we allocate variables based on their nested definitions, and the block we branch to is in scope, this guarantees that the new temporaries occur higher on the stack than their targets, so they won't be overwritten

Revisit this to get a more efficient allocation scheme when we perform register allocation

Compiling Branch with Arguments

easy, sub-optimal solution

```
x = 7
f(a, b):
  z = x * a
  w = b + z
  ret w
y = x * 2
a2 = 5
b2 = y
br f(a2, b2)
```

```
mov rax, [rsp - 24]
mov [rsp - 16], rax ;; a = a2
mov rax, [rsp - 32]
mov [rsp - 24], rax ;; b = b2
jmp f
```

Functional to SSA

Summary:

If a function is only ever tail-called locally, it can be compiled directly to an SSA block with arguments. Tail calls can then be compiled to branch with arguments

A tail call is a call to a function in tail position: the result of the function call is immediately returned.

Functional to SSA

It's easy to map functional code to an SSA code since SSA is essentially functional.

But, is that the **best** translation of the functional code? Probably not!

Minimal SSA

An SSA program is **minimal** if it uses as few block arguments (phi nodes) as possible.

Useful for optimization: branching to a block with arguments is compiled to a **mov,** potentially causing memory access. Want to reduce these as much as possible.

Minimal SSA

The following SSA is **not** minimal

$$
\begin{aligned}
&\textbf{function } f_1() = \textbf{let } v = 1,\ z = 8,\ y = 4 \\
&\qquad\qquad\qquad\qquad \textbf{in } f_2(v, z, y) \textbf{ end} \\
&\textbf{and } f_2(v, z, y) = \textbf{let } x = 5 + y,\ y = x \times z,\ x = x - 1 \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in if } x = 0 \textbf{ then } f_3(y, v) \textbf{ else } f_2(v, z, y) \textbf{ end} \\
&\textbf{and } f_3(y, v) = \textbf{let } w = y + v \textbf{ in } w \textbf{ end} \\
&\textbf{in } f_1() \textbf{ end}
\end{aligned}
$$

SSA Minimization

Minimizing SSA form consists of two phases:

1. Block Sinking: pushing block definitions lower in the SSA AST, so that more variables are in scope of its definition

2. Parameter dropping: removing unnecessary block parameters

Block Sinking

Push function definitions inside of others if they are **dominated**. I.e., given f and g, if g is only ever called inside f or g, then f **dominates** g, and so g's definition could be sunk inside of the definition of f.

$$
\begin{aligned}
&\textbf{function } f_1() = \textbf{let } v = 1,\ z = 8,\ y = 4 \\
&\qquad\qquad\qquad\qquad \textbf{in } f_2(v, z, y) \textbf{ end} \\
&\textbf{and } f_2(v, z, y) = \textbf{let } x = 5 + y,\ y = x \times z,\ x = x - 1 \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in if } x = 0 \textbf{ then } f_3(y, v) \textbf{ else } f_2(v, z, y) \textbf{ end} \\
&\textbf{and } f_3(y, v) = \textbf{let } w = y + v \textbf{ in } w \textbf{ end} \\
&\textbf{in } f_1() \textbf{ end}
\end{aligned}
$$

which of f1, f2, f3 dominates which?

Block Sinking

f1 dominates f2 dominates f3. Sink blocks accordingly:

```
function f1() =
    let v = 1, z = 8, y = 4
    in function f2(v, z, y) =
            let x = 5 + y, y = x × z, x = x − 1
            in if x = 0
                then function f3(y, v) = let w = y + v in w end
                        in f3(y, v) end
                else f2(v, z, y)
            end
        in f2(v, z, y) end
    end
in f1() end
```

Parameter Dropping

If a parameter **x** is always instantiated with **y** or itself, then we can remove **x** and replace all occurrences with **y** as long as it is in the scope of **y**.

Parameter Dropping

Which parameters can be dropped?

```
function f1() =
    let v = 1, z = 8, y = 4
    in function f2(v, z, y) =
            let x = 5 + y, y = x × z, x = x − 1
            in if x = 0
                then function f3(y, v) = let w = y + v in w end
                        in f3(y, v) end
                else f2(v, z, y)
            end
        in f2(v, z, y) end
    end
in f1() end
```

Parameter Dropping

Which parameters can be dropped?

```
function f1() =
    let v = 1, z = 8, y = 4
    in function f2(v, z, y) =
            let x = 5 + y, y = x × z, x = x − 1
             in if x = 0
                then function f3() = let w = y + v in w end
                     in f3() end
                else f2(v, z, y)
            end
        in f2(v, z, y) end
    end
in f1() end
```

Parameter Dropping

```
function f1() =
    let v = 1, z = 8, y = 4
    in function f2(y) =
            let x = 5 + y, y = x × z, x = x − 1
            in if x = 0
                then function f3() = let w = y + v in w end
                        in f3() end
                else f2(y)
            end
        in f2(y) end
    end
in f1() end
```

Minimal: only block arg is y and this does take on multiple values