EECS 483: Compiler Construction

Lecture 6:
Tail Calls

February 3, 2025

Announcements

- Assignment 2 released today, due on Friday February 14.
- Builds on solution to Assignment 1: can use your own Assignment 1 solution or our provided reference solution as a starting point.

Extending the Snake Language

So far:

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

This week:

- 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?

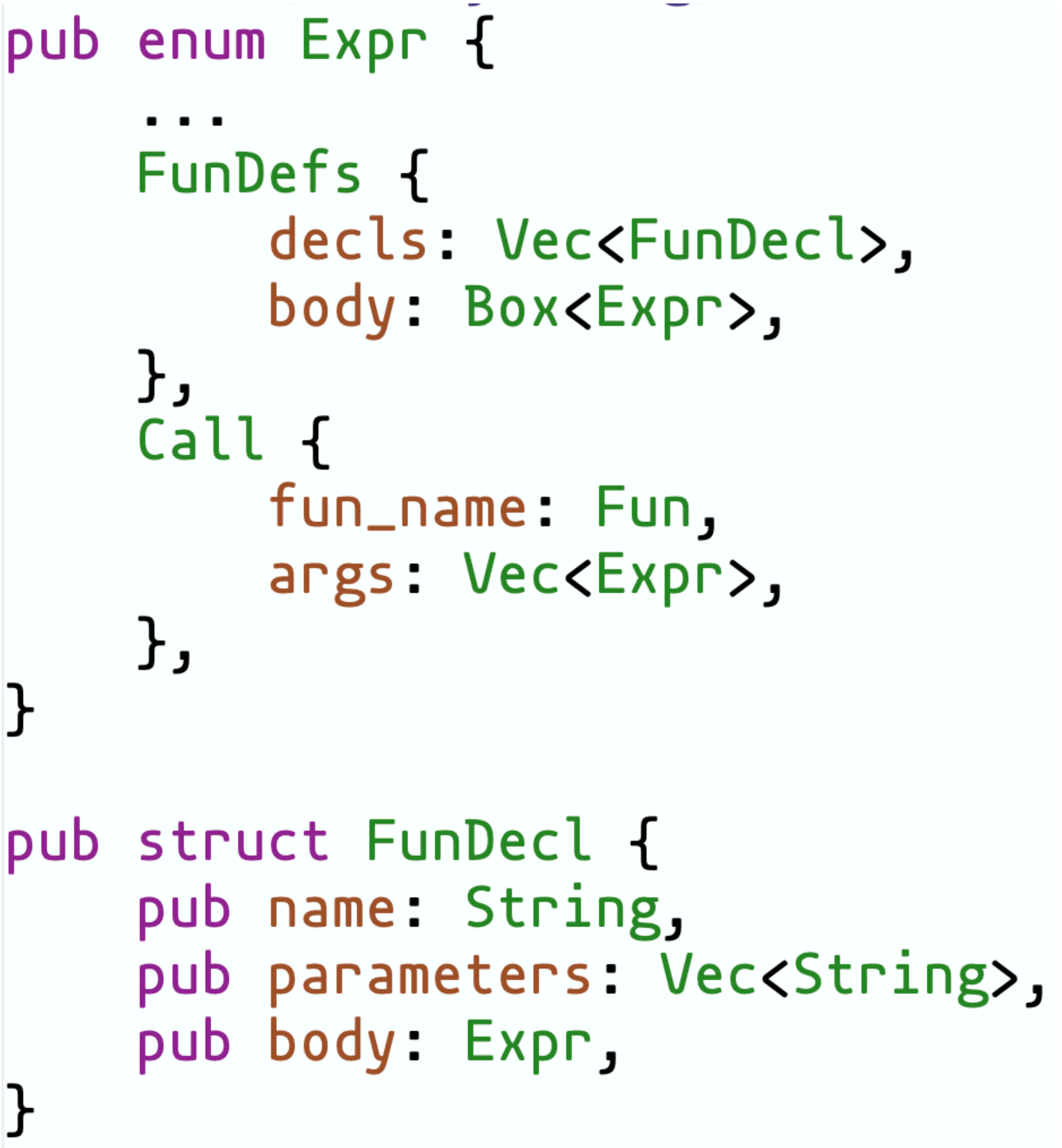
Extending the Snake Language

```
‹expr›:
        | ...
        | IDENTIFIER ( ‹exprs› )
        | IDENTIFIER ( )
        | ‹decls› in ‹expr›
‹exprs›: ‹expr› | ‹expr› , ‹exprs›
```

```
‹decls›:
        | ‹decl›
        | ‹decls› and ‹decl›
‹decl›:
        | def IDENTIFIER ( ‹ids› ) : ‹expr›
        | def IDENTIFIER ( ) : ‹expr›

‹ids›:
        | IDENTIFIER
        | IDENTIFIER , ‹ids›
```

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

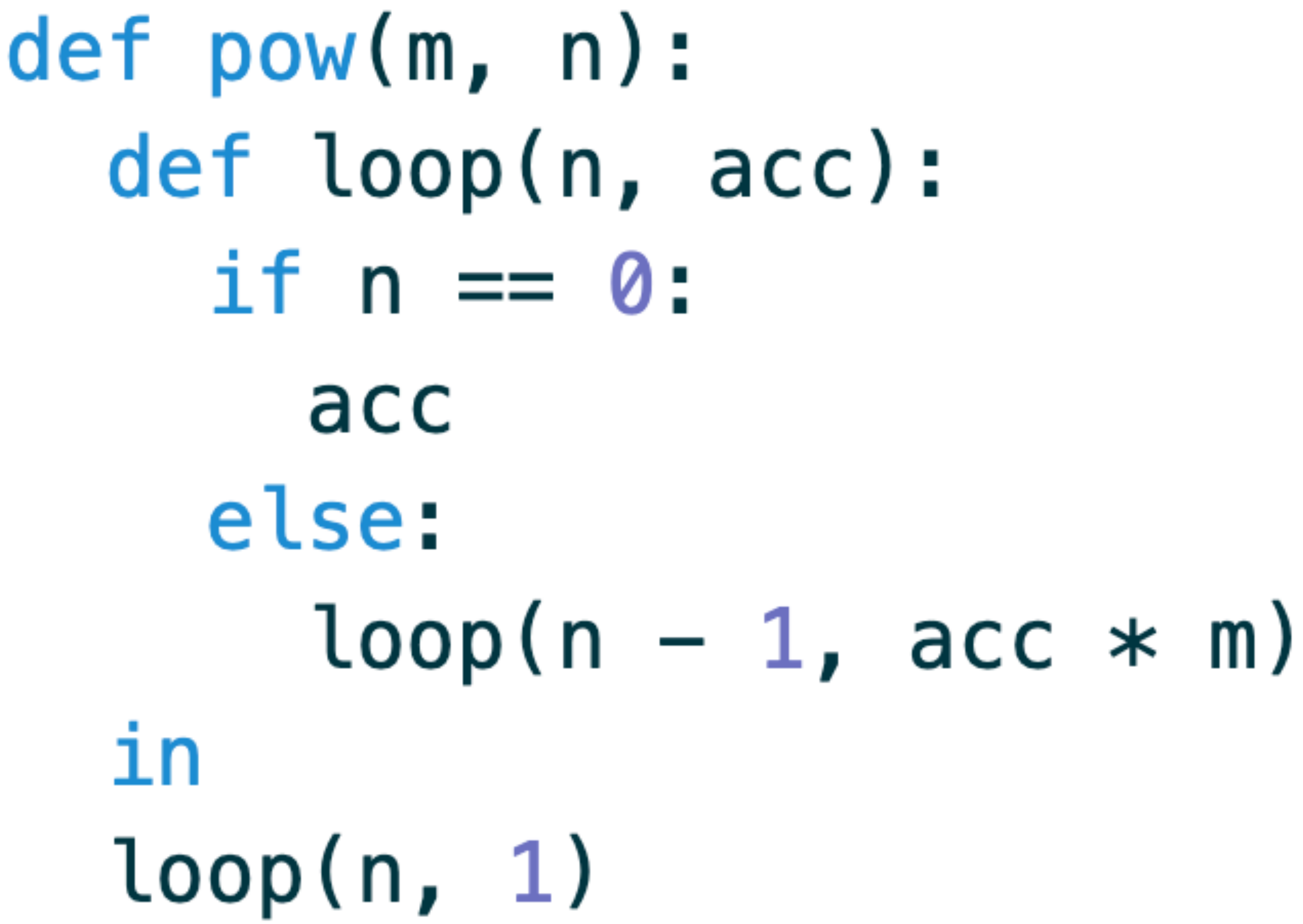
Function definitions separated by an and are **mutually recursive.** Mutually recursive functions are all in scope of each other.

```
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.

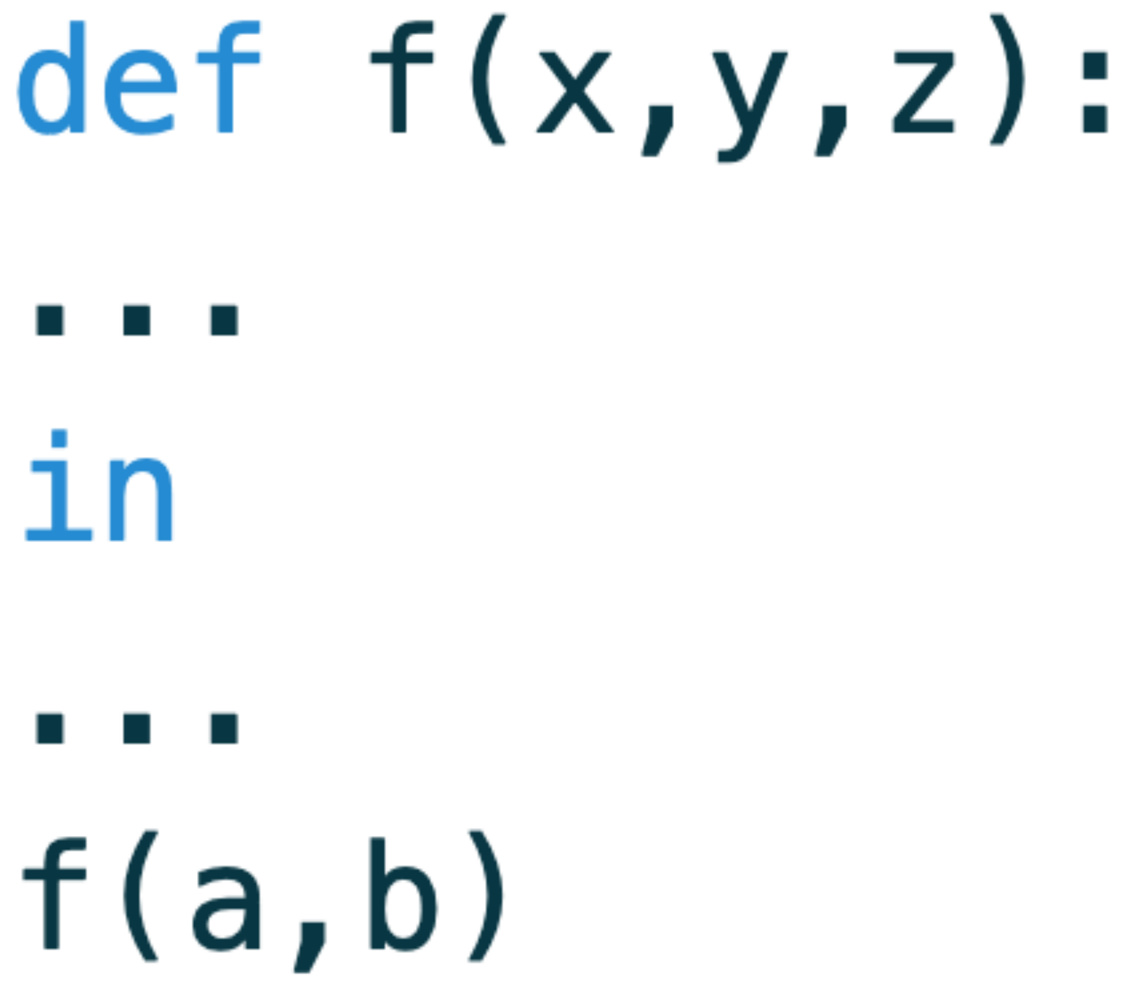
Allow shadowing of function names, like variable declarations. Similarly, resolve all function names to unique identifiers.

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

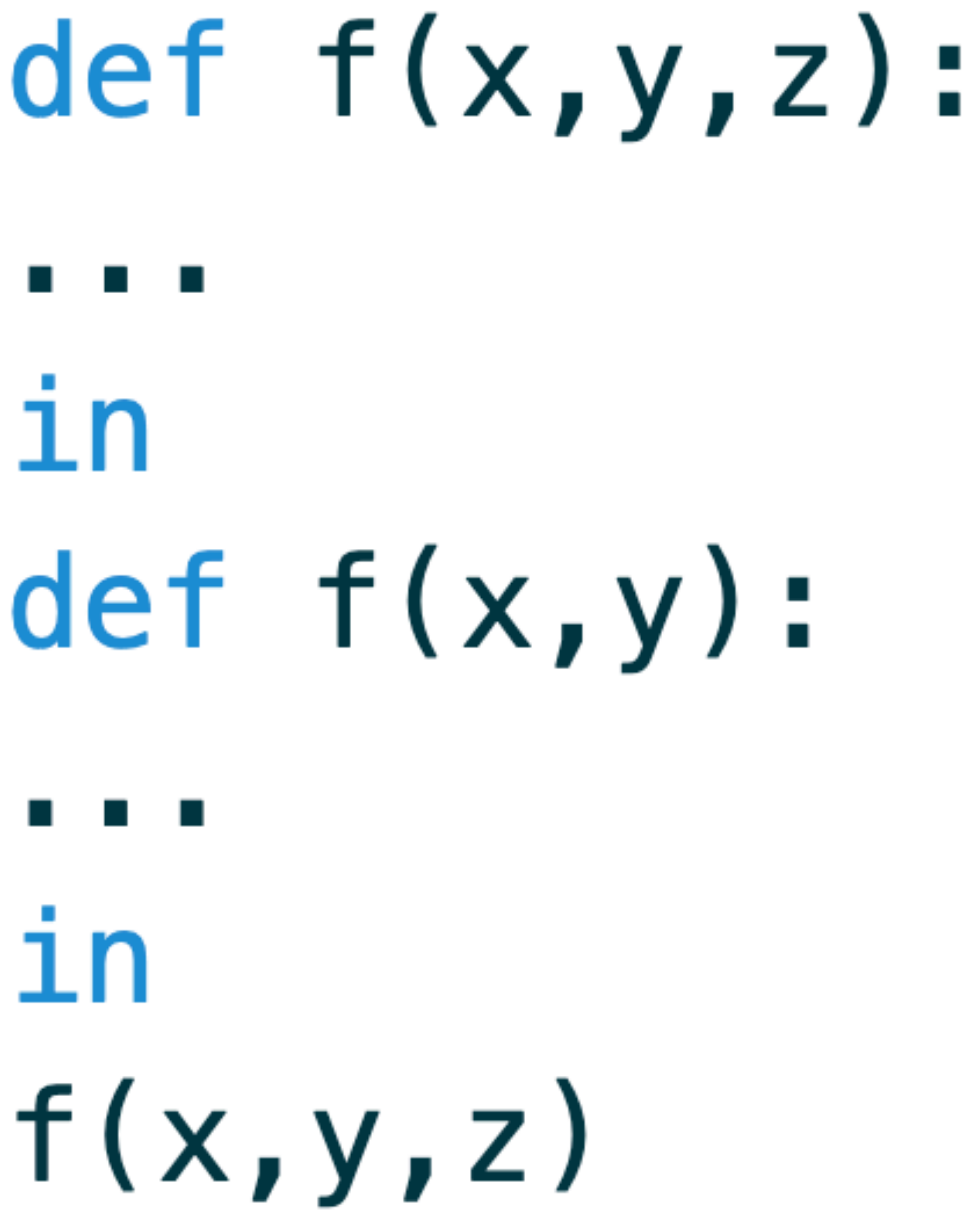


```
def f(x,y,z):
...
in
...
f(a,b)
```

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)
```

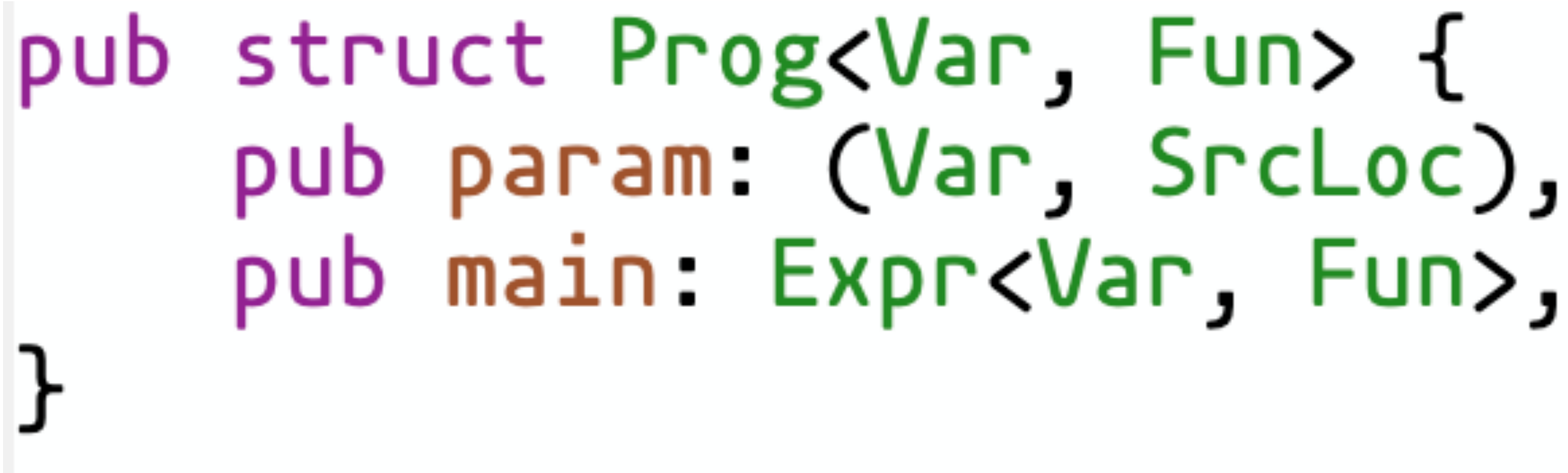
```
def factorial(x):
  if x == 0:
    1
  else:
    x * factorial(x - 1)
in
factorial(6)
```

Tail Position

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

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

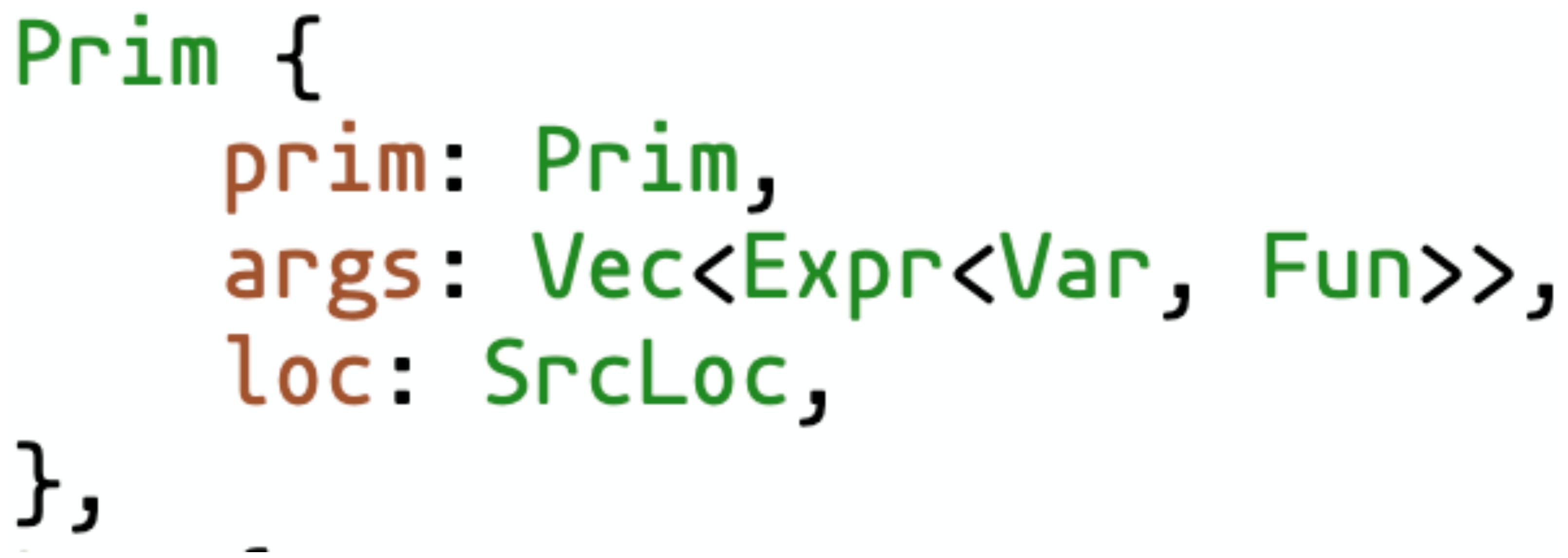
Tail Position



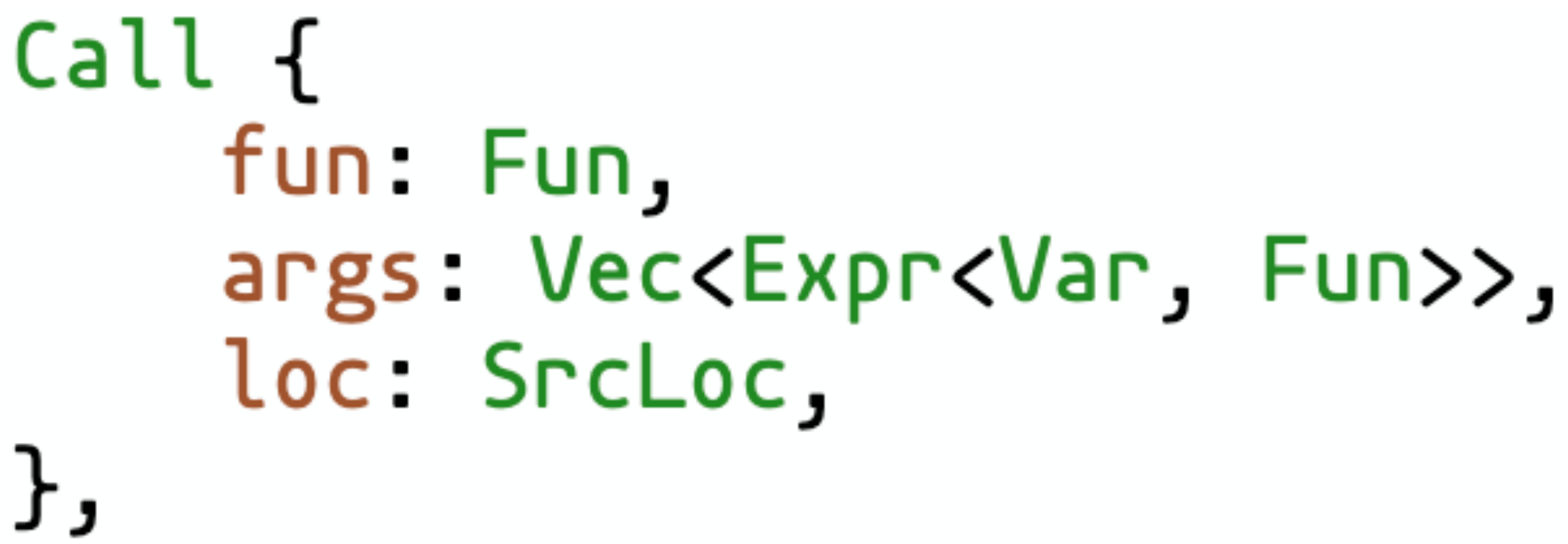
```
pub struct Prog<Var, Fun> {
    pub param: (Var, SrcLoc),
    pub main: Expr<Var, Fun>,
}
```

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

Tail Position



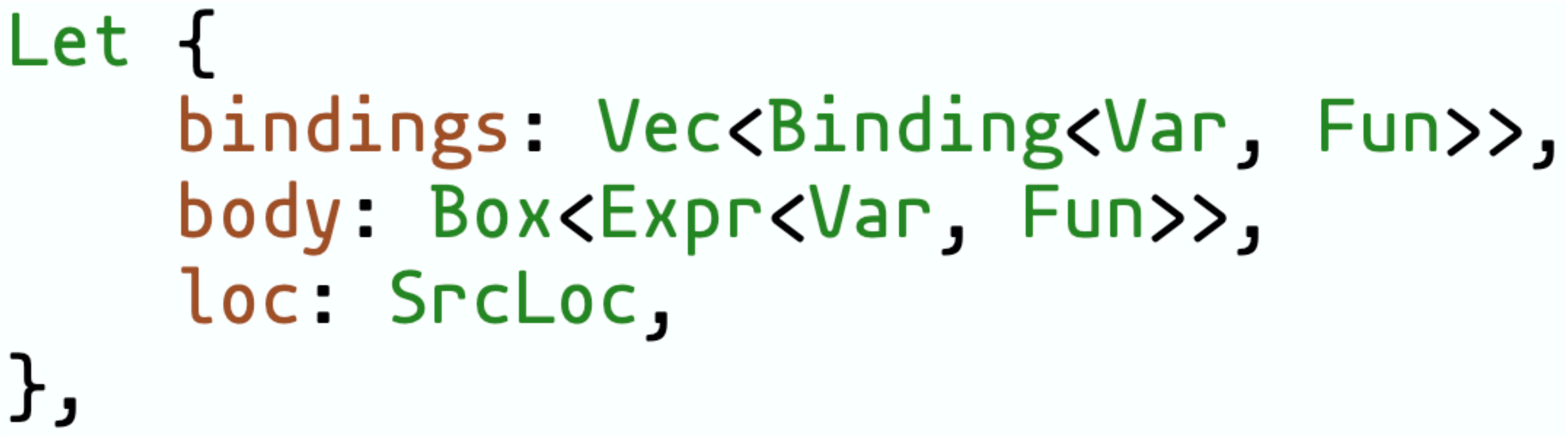
```
Prim {
    prim: Prim,
    args: Vec<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```



```
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

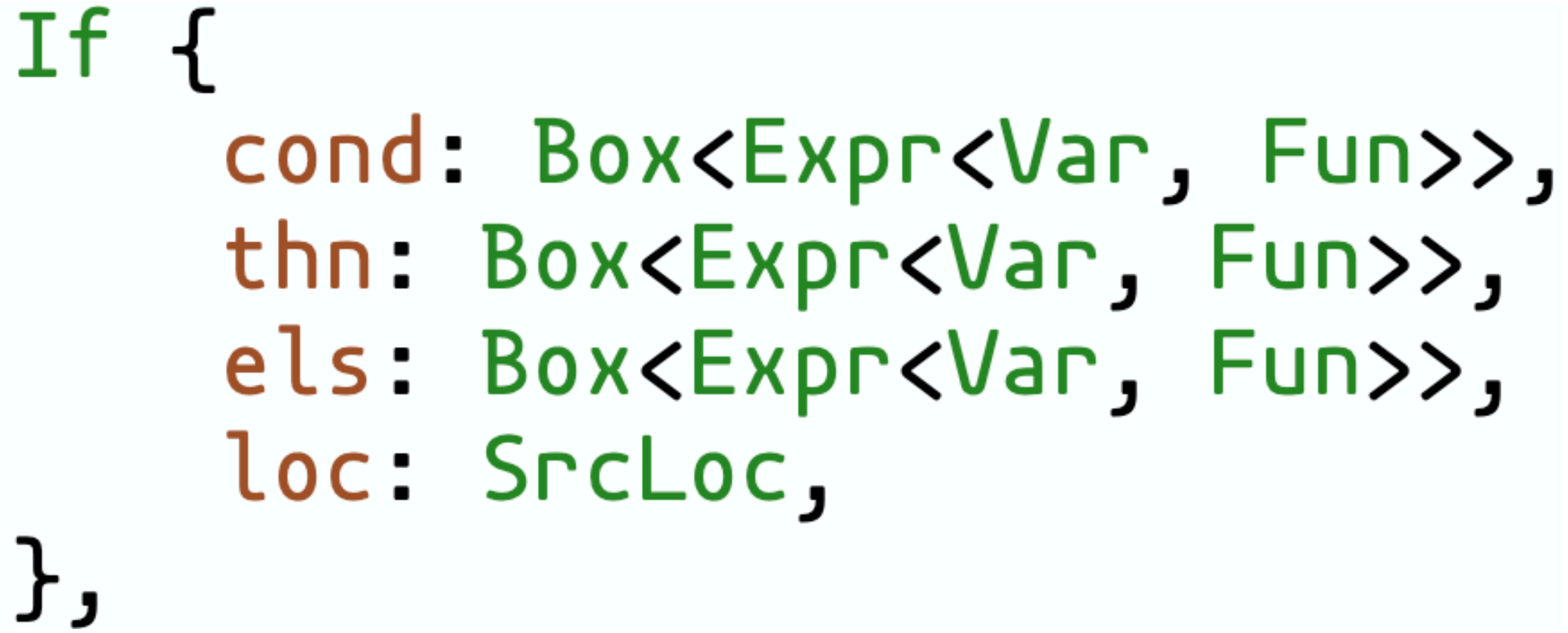


```
Let {
    bindings: Vec<Binding<Var, Fun>>,
    body: Box<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

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

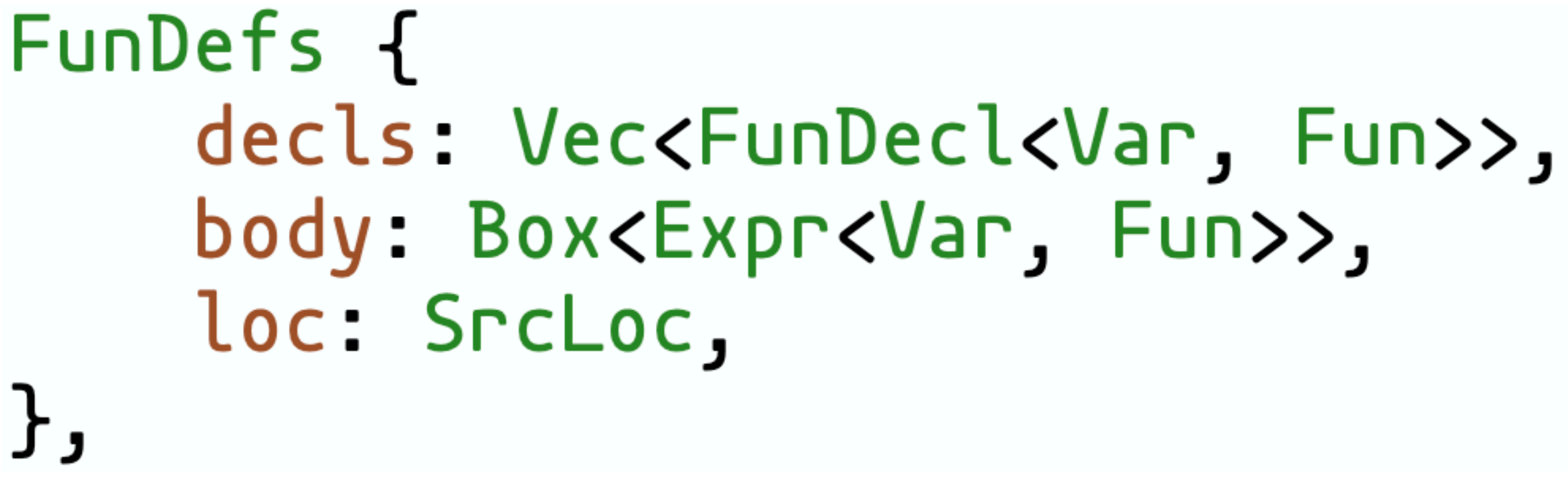


```
If {
    cond: Box<Expr<Var, Fun>>,
    thn: Box<Expr<Var, Fun>>,
    els: Box<Expr<Var, Fun>>,
    loc: SrcLoc,
},
```

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

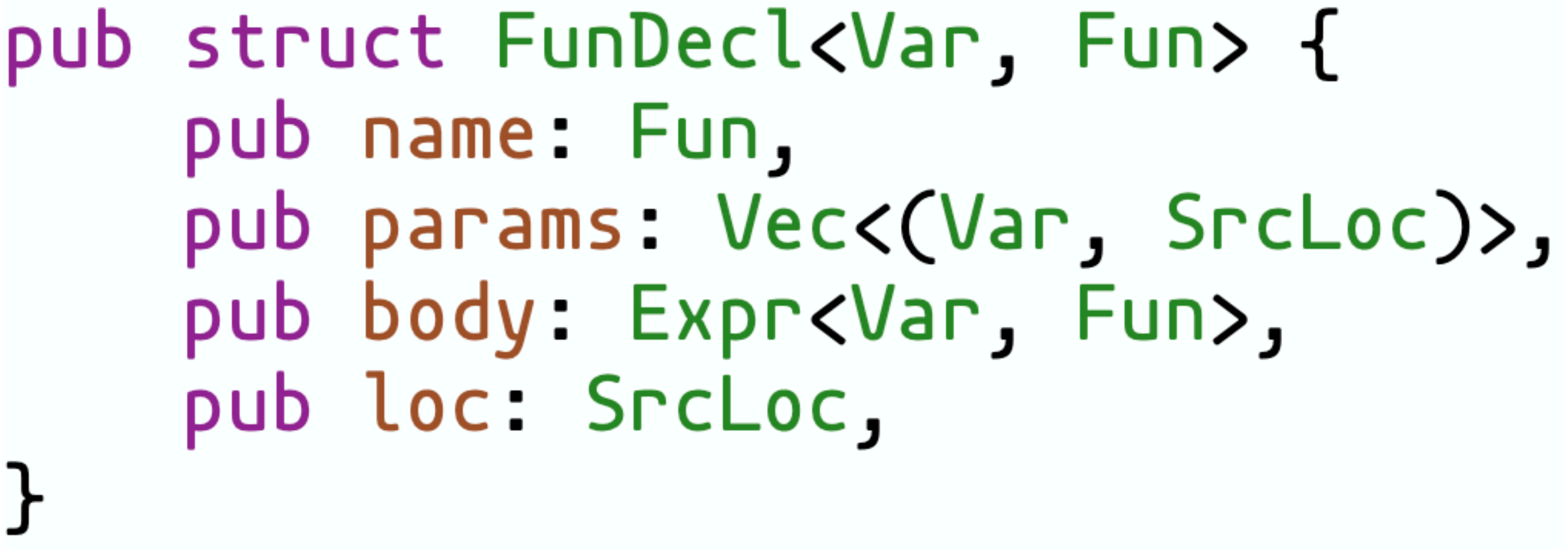
Tail Position



```
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.

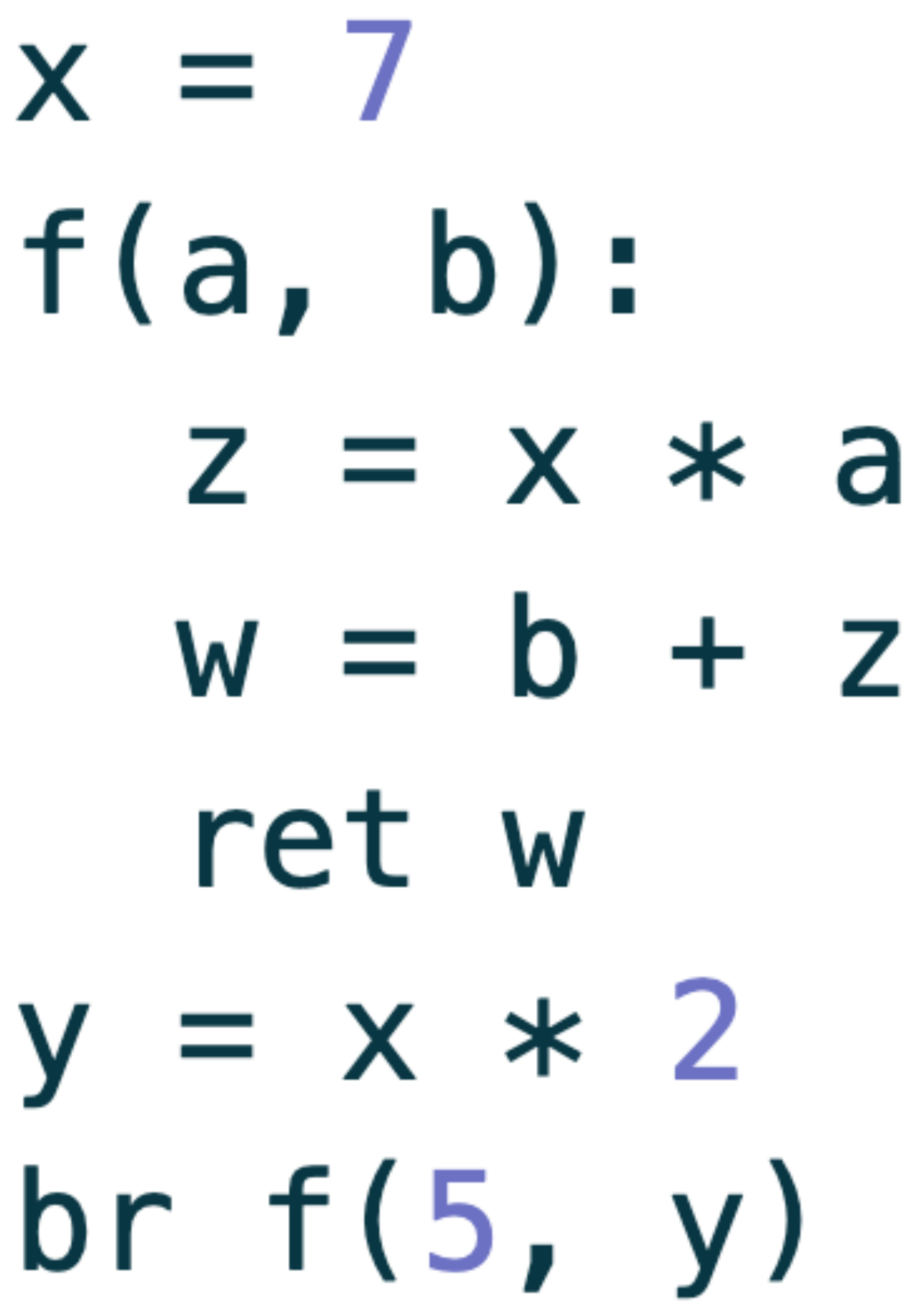
Compiling Branch with Arguments

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

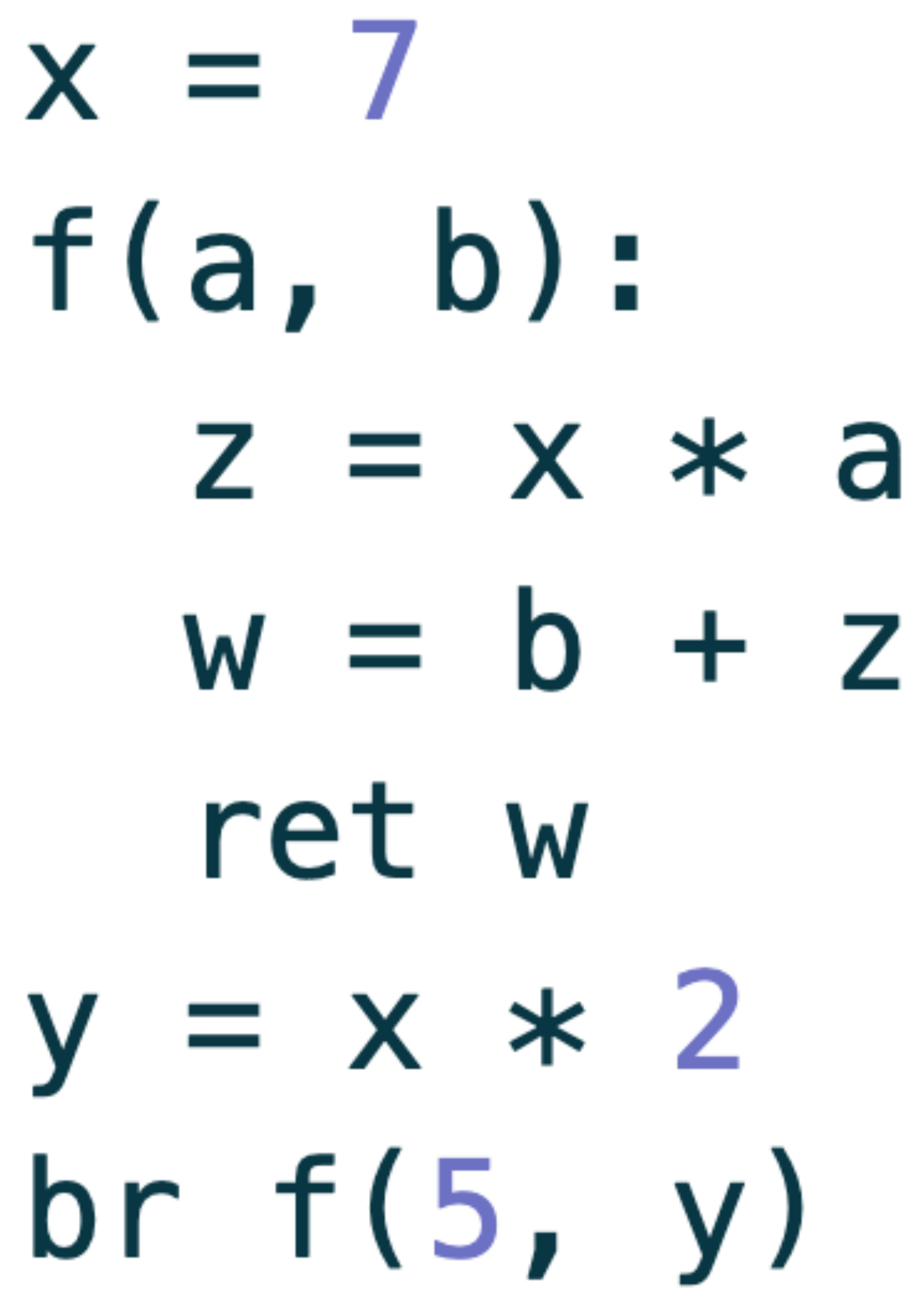


```
x = 7
f(a, b):
  z = x * a
  w = b + z
  ret w
y = x * 2
br f(5, y)
```

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

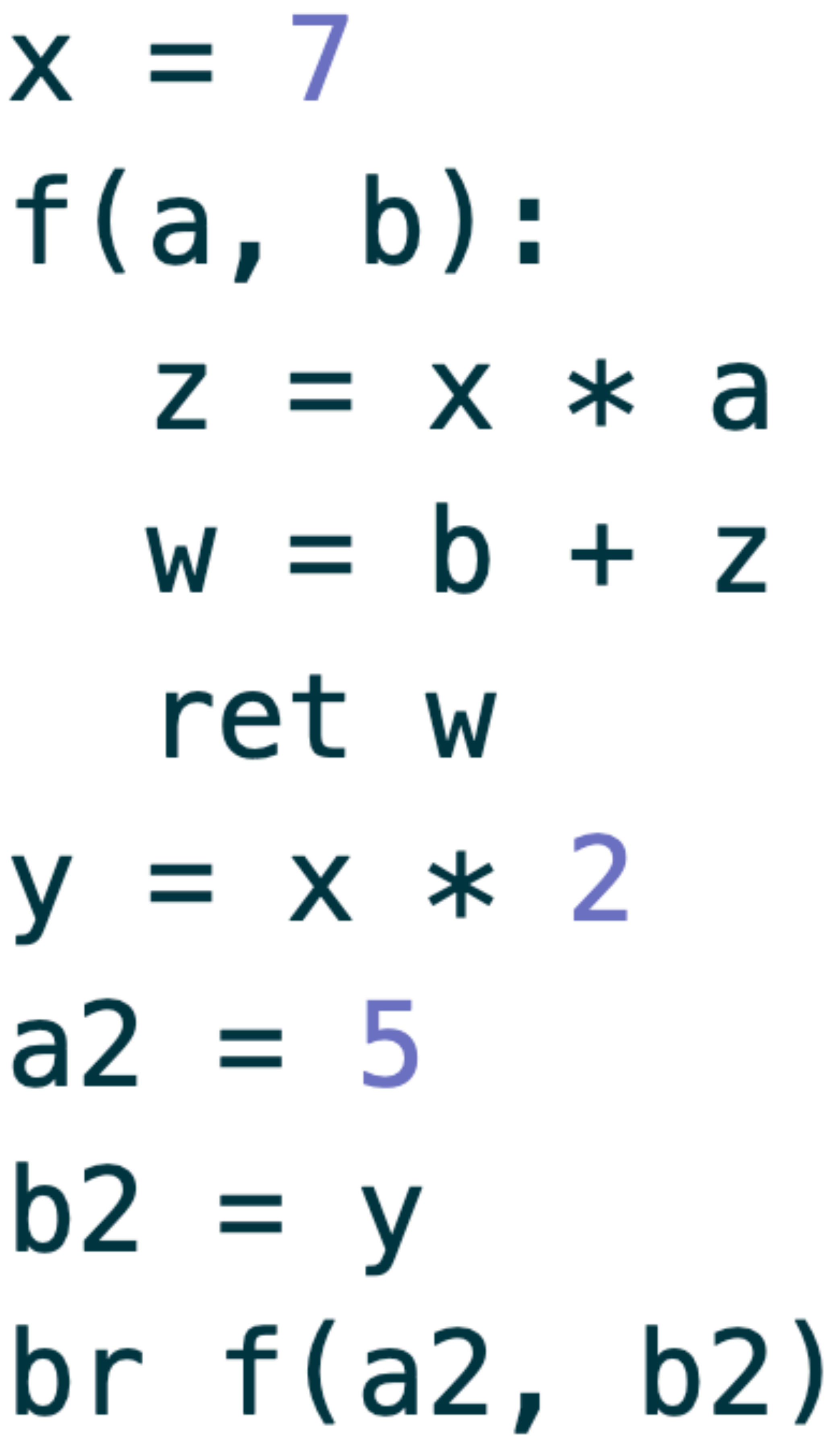
To ensure we don't overwrite memory we are about to use, we can introduce extra temporaries for the arguments.

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!

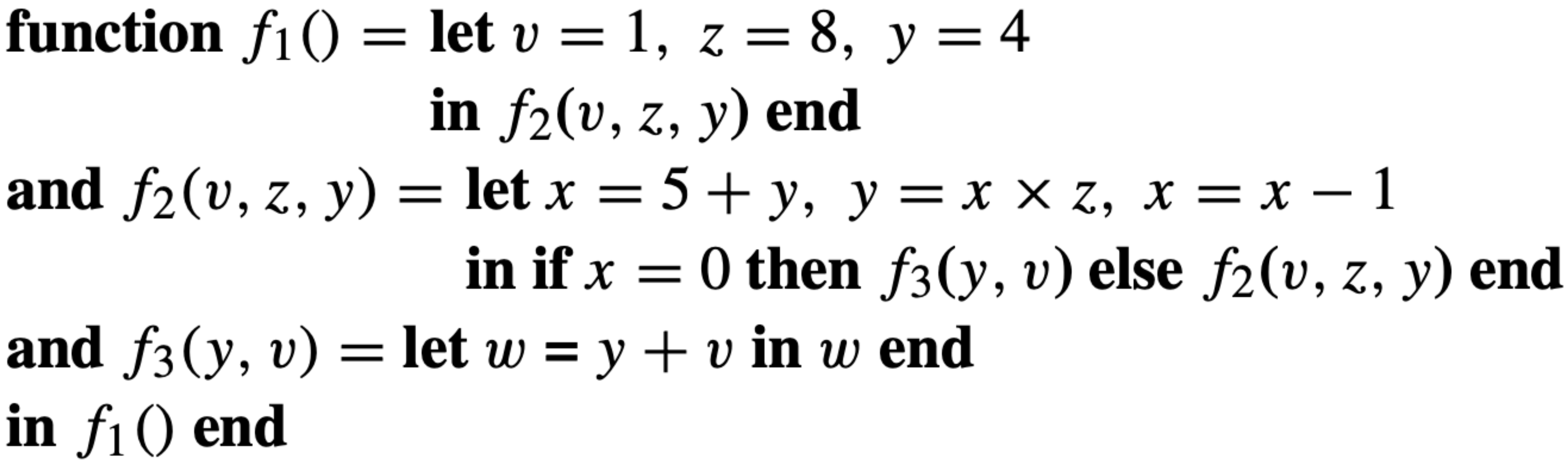
Minimal SSA

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



```
function $f_1() =$ let $v = 1,\ z = 8,\ y = 4$
                   in $f_2(v, z, y)$ end
and $f_2(v, z, y) =$ let $x = 5 + y,\ y = x \times z,\ x = x - 1$
                     in if $x = 0$ then $f_3(y, v)$ else $f_2(v, z, y)$ end
and $f_3(y, v) =$ let $w = y + v$ in $w$ end
in $f_1()$ end
```

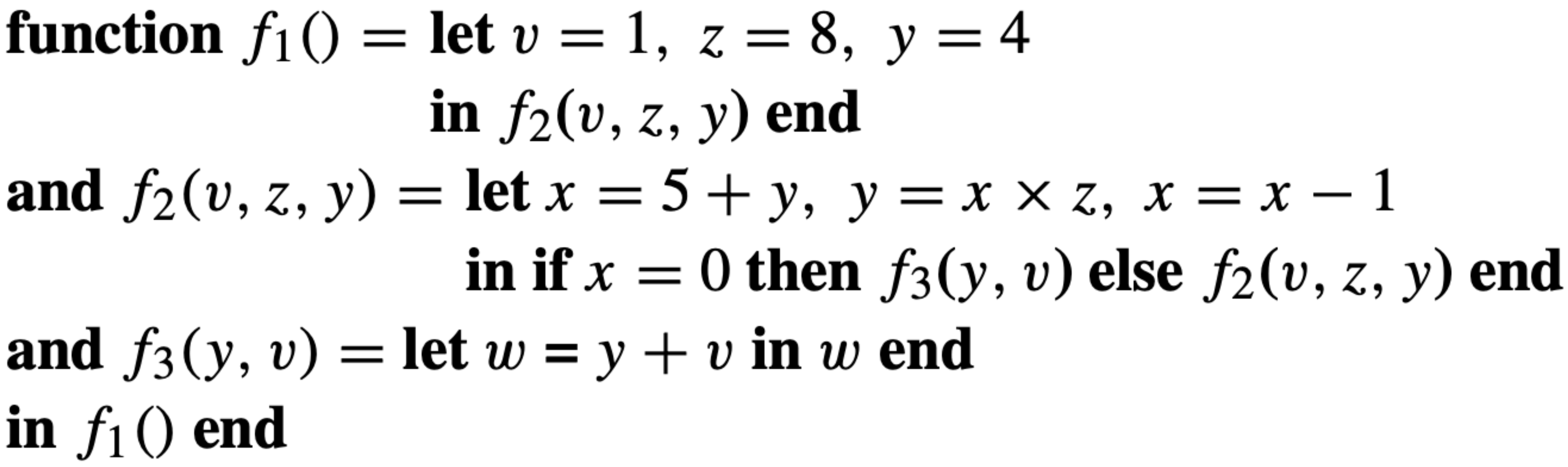
SSA Minimization

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.

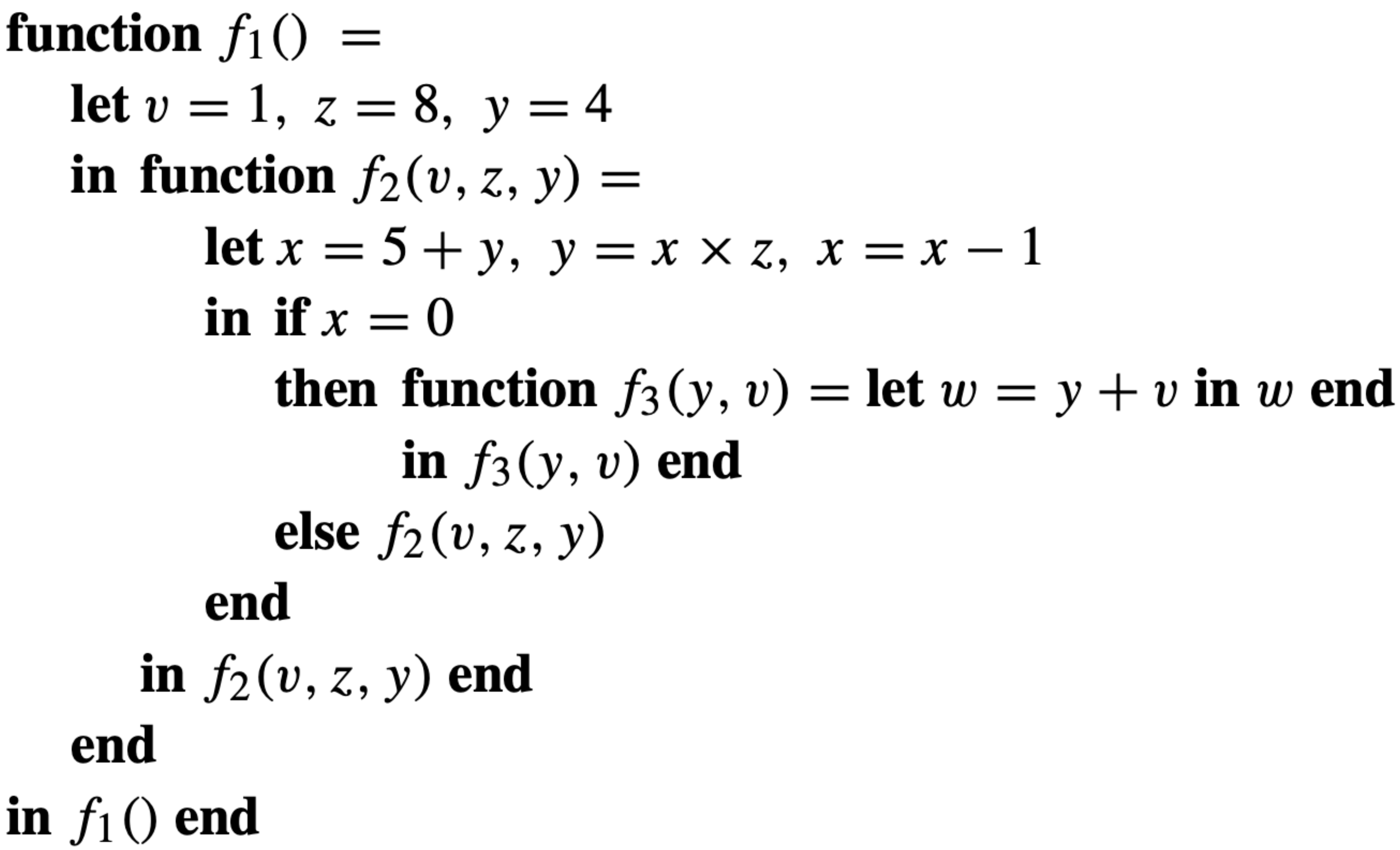


```
function f1() = let v = 1,  z = 8,  y = 4
                in  f2(v, z, y) end
and f2(v, z, y) = let x = 5 + y,  y = x × z,  x = x − 1
                  in if x = 0 then  f3(y, v) else  f2(v, z, y) end
and f3(y, v) = let w = y + v in w end
in f1() end
```

which of f1, f2, f3 dominates which?

Block Sinking

f1 dominates f2 dominates f3. Sink blocks accordingly:



```
function $f_1()$ =
    let $v = 1,\ z = 8,\ y = 4$
    in function $f_2(v, z, y)$ =
            let $x = 5 + y,\ y = x \times z,\ x = x - 1$
            in if $x = 0$
                then function $f_3(y, v)$ = let $w = y + v$ in $w$ end
                        in $f_3(y, v)$ end
                else $f_2(v, z, y)$
            end
        in $f_2(v, z, y)$ end
    end
in $f_1()$ 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 $f_1()$ =
   let $v = 1,\ z = 8,\ y = 4$
   in function $f_2(v, z, y)$ =
         let $x = 5 + y,\ y = x \times z,\ x = x - 1$
         in if $x = 0$
            then function $f_3(y, v)$ = let $w = y + v$ in $w$ end
                  in $f_3(y, v)$ end
            else $f_2(v, z, y)$
         end
      in $f_2(v, z, y)$ end
   end
in $f_1()$ end
```

Parameter Dropping

Which parameters can be dropped?

```
function $f_1()$ =
    let $v = 1,\ z = 8,\ y = 4$
    in function $f_2(v, z, y) =$
            let $x = 5 + y,\ y = x \times z,\ x = x - 1$
            in if $x = 0$
                then function $f_3() = $ let $w = y + v$ in $w$ end
                        in $f_3()$ end
                else $f_2(v, z, y)$
            end
        in $f_2(v, z, y)$ end
    end
in $f_1()$ end
```

Parameter Dropping

```
function $f_1()$ =
    let $v = 1,\ z = 8,\ y = 4$
    in function $f_2(y)$ =
            let $x = 5 + y,\ y = x \times z,\ x = x - 1$
            in if $x = 0$
                then function $f_3()$ = let $w = y + v$ in $w$ end
                        in $f_3()$ end
                else $f_2(y)$
            end
        in $f_2(y)$ end
    end
in $f_1()$ end
```

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