EECS 483: Compiler Construction

Lecture 15:
Midterm Review

March 12
Winter Semester 2025

Midterm Logistics

- Date/Time: Tuesday, March 18 6-8pm.
- Location: 3 rooms in DOW. Which room is determined by your uniqname:
  - DOW1010 if your uniqname starts with A-JO
  - DOW1017 if your uniqname starts with JS-R
  - DOW1018 if your uniqname starts with S-Z
- Bring your own pen/pencil.
- 1 page of notes ("cheat sheet") allowed.
  - Standard letter size
  - Typed or hand-written ok

Midterm Review

Today: Go over the main topics of the first half of the course.

Please interrupt if a topic is/was unclear so we can review it!

Friday: Discussion section with further review material. Bring questions

First Half Course Major Topics

1. Well-formedness, Name resolution
   - Scope, shadowing, free and bound variables
   - Resolving names to be unique identifiers
2. Interpreters, Language Semantics
   - evaluation order
   - runtime and compile time errors
3. x86 abstract machine, instructions
   - Registers, Memory
   - Instruction pointer, Jmp/Conditional Jmp
   - Flags, cmp, test
   - Labels
4. Translation into SSA
   - encoding evaluation order using continuations
   - conditional branching, join points
   - branch with arguments
     - lambda lifting
     - minimal SSA form
5. Functions/Procedures
   - tail vs non-tail calls
     - first class vs second class
     - Calling conventions
6. Datatypes
   - semantics of dynamic typing
   - tagging schemes
   - heap allocation

Adder

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 can we generate that assembly code programmatically?

Snake v0.1: "Adder"

Today: add immutable variables to Adder, to allow saving results of intermediate computations

Snake v0.1: "Adder"

⟨*prog*⟩: `def` `main` `(` ***IDENTIFIER*** `)` `colon` ⟨expr⟩

⟨*expr*⟩:

| ***NUMBER***

| `add1` `(` ⟨expr⟩ `)`

| `sub1` `(` ⟨expr⟩ `)`

| ***IDENTIFIER***

| `let` ***IDENTIFIER*** `=` ⟨expr⟩ `in` ⟨expr⟩

Examples

```
def main(x):
  let y = sub1(x) in
  let z = add1(add1(y)) in
  add1(z)
```

Examples

```
def main(x):
  let z =
      let y = sub1(x) in
      add1(add1(y)) in
  add1(z)
```

Let is an **expression** form, just like add1 and sub1

Examples

```
def main(x):
  let z = add1(add1(let y = sub1(x) in y)) in
  add1(z)
```

Let is an **expression** form, just like add1 and sub1

Expressions vs Statements

In most languages in the C style, variable bindings belong to a separate syntactic class of **statements**.

In languages with a functional programming style, it is more common to allow most syntactic constructs.

Rust is somewhere in the middle

```
def main(x):
  let z = add1(add1(let y = sub1(x) in y)) in
  add1(z)
```

```
fn funny(x: i64) -> i64 {
    let z = {
        add1(add1({
            let y = sub1(x);
            y
        }))
    };
    add1(z)
}
```

Example?

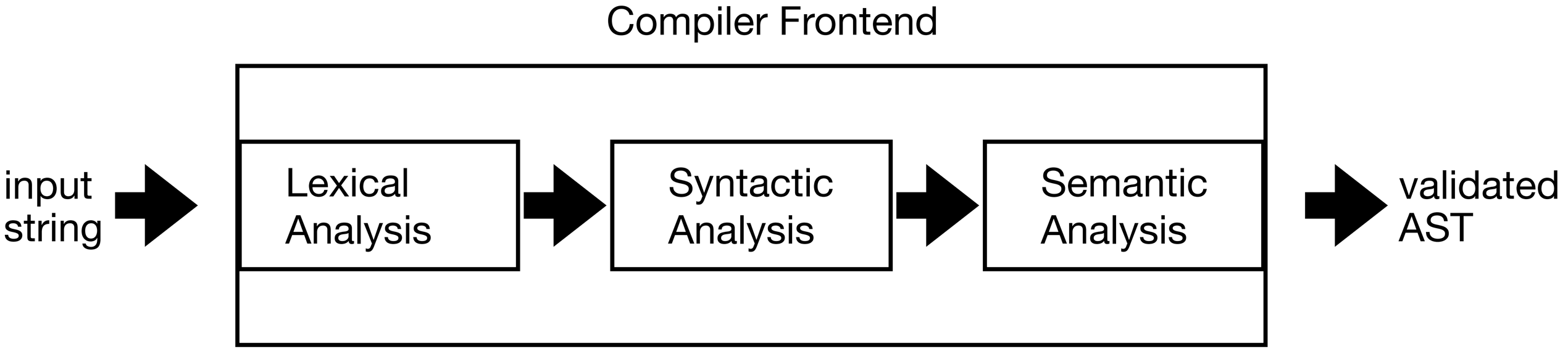
```
def main(x):
    y
```

Does this example match our grammar?

Should it be considered a valid program?

Compiler Frontends

Even after parsing, there are some conditions on the syntax that still remain to be checked. This is inherent: to be implemented efficiently, parsers use computationally restrictive languages that are not capable of performing all of the **semantic analysis** necessary to check if the input program is valid



Semantic Analysis

Examples:

- Scope checking (today)
- Type checking
- Borrow checking

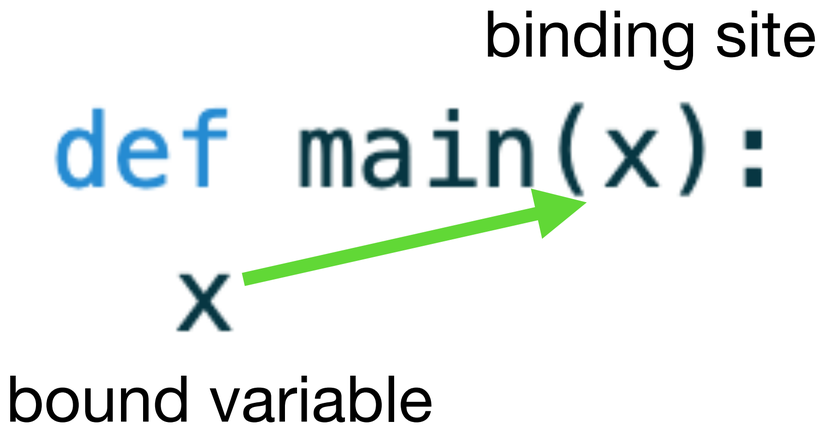
EECS 490 covers type checking in more detail.

Free and Bound Variables

```
def main(x):
    y
```

We say this program is invalid because the y is a **free variable**, meaning it has not been defined

Free and Bound Variables



The usage of x here is valid because it occurs within the scope of a binding site that binds the variable name x. We call such a usage a **bound** variable

Free and Bound variables

```
def main(x):
let y =
  let z = x in add1(y)
  in
let w = sub1(x) in
sub1(z)
```

There are 8 variables in this program. Which ones are binding sites, which ones are free variables and which ones are bound variables?

Live Code: Scope Checking

To define scope rigorously, let's define a scope checker in Rust.

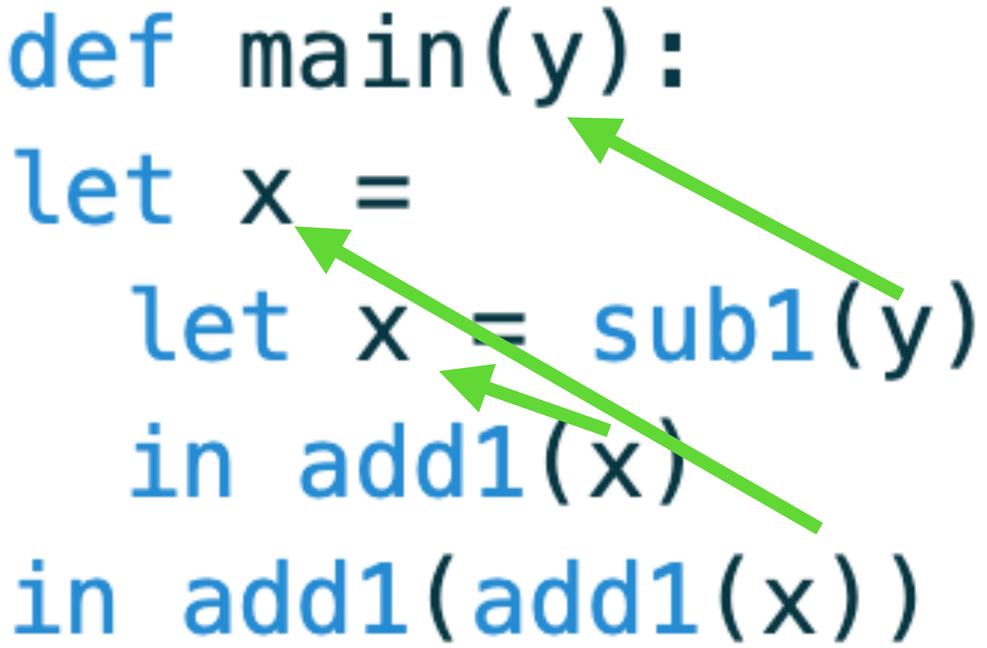
Variable Names are Tricky

We use variable names as a way to refer back to binding sites. But because names are implemented as strings, sometimes the same name is used to refer to multiple binding sites.

```
def main(y):
let x =
  let x = sub1(y)
  in add1(x)
in add1(add1(x))
```

Variable Names are Tricky

We use variable names as a way to refer back to binding sites. But because names are implemented as strings, sometimes the same name is used to refer to multiple binding sites.



Shadowing

Should this be allowed?

```
let x = 1 in
let x = 2 in
x
```

Shadowing

Should this be allowed?

We say the second binding **shadows** the first

```
let x = 1 in
let x = 2 in
x
```

If a binding is shadowed, it's impossible to refer to it in the source program!

Beta Reduction

A common rewrite we can apply to our ASTs is called beta reduction

```
let x = e1 in e2
```

rewrites to

```
e2
```

with all occurrences of x replaced by e1

Beta Reduction

```
let x = y in
let z = add1(x) in
add1(add1(z))
```

rewrites to

```
let z = add1(y) in
add1(add1(z))
```

Beta Reduction

Is there any situation where this rewrite is **not** correct? I.e., where the two different expressions have different behaviors?

```
let x = e1 in e2
```

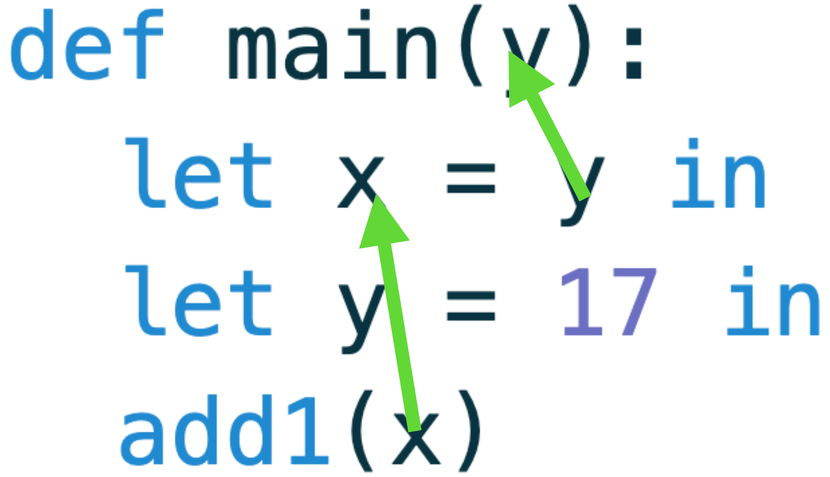
rewrites to

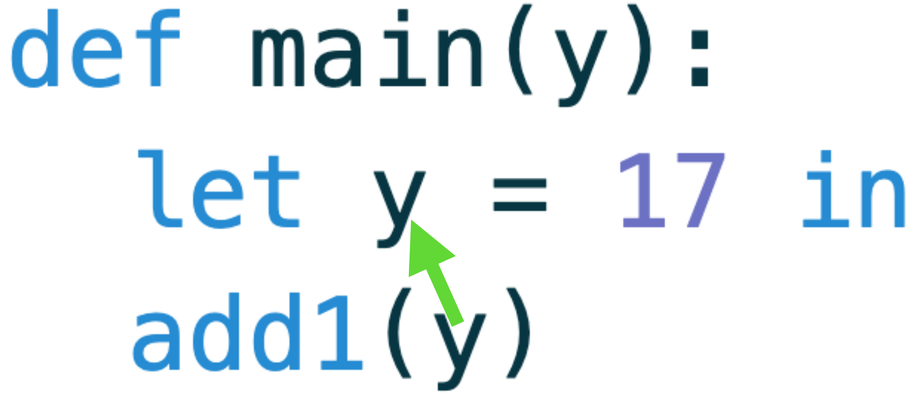
```
e2
```

with all occurrences of x replaced by e1

Beta Reduction

Is there any situation where this rewrite is **not** correct? I.e., where the two different expressions have different behaviors?





we say that the inner binding of y has **captured** the occurrence of y on the inside

Unique Variable names

Shadowing is convenient for programmers, but ultimately harmful to compilers. For this reason compilers typically implement a variable renaming phase that makes sure that all binding occurrences are globally unique

```
def main(y#0):
  let x#0 = y#0 in
  let y#1 = 17 in
  add1(x#0)
```

```
def main(y#0):
  let y#1 = 17 in
  add1(y#0)
```

Ensuring that all variables are unique ensures we can move code around without worrying about capture.

Compiling Let

In the interpreter, the value of each variable was stored in a HashMap.

In the compiled code, we correspondingly need to ensure that we have access to the value of each variable somewhere in **memory**

x86 Memory Model

16 general-purpose 64-bit registers

- rax, rcx, rdx, rbx, rdi, rsi, rsp, rbp, r8-r15

Each holds a 64-bit value, so 128 bytes of extremely fast memory.

The abstract machine also gives us access to a large amount of memory, which is addressable by byte.

- Addresses are 64-bit values, though in current hardware only the lower 48-bits are used. This gives us access to 2^48 bytes of address space, or 128 terabytes.

x86 Instructions: mov

mov dest, src

In a mov, the dest and src can be registers or memory addresses.

Use square brackets [ ] to "dereference" an address.

- mov rax, rdi copies the value stored in rdi to rax
- mov rax, [rdi] loads the memory at address rdi into rax
- mov [rax], rdi stores the value of rdi in the memory at address rax
- mov [rax], [rdi] - not allowed in x86 syntax

x86 Instructions: mov

```
mov dest, src
```

In a mov, the dest and src can be registers or memory addresses.

Addresses can be not just registers, but offsets from registers

```
mov rax, [rsp - 8 * 3]
```

x86 Memory Conventions

Registers give us access to 128 bytes, and byte-addressable memory gives us access to 128 terabytes.

But that memory needs to be **shared** by different components of the process (functions, objects, allocator, garbage collector, etc).

We can't just start writing to a random portion of memory

1. That memory might be used by another component, like our caller, and we would break the invariants of that component

2. Hardware supports mechanisms for process isolation, so most of the memory space will be invalid for us to access, causing the dreaded **segmentation fault**

x86 Memory Conventions

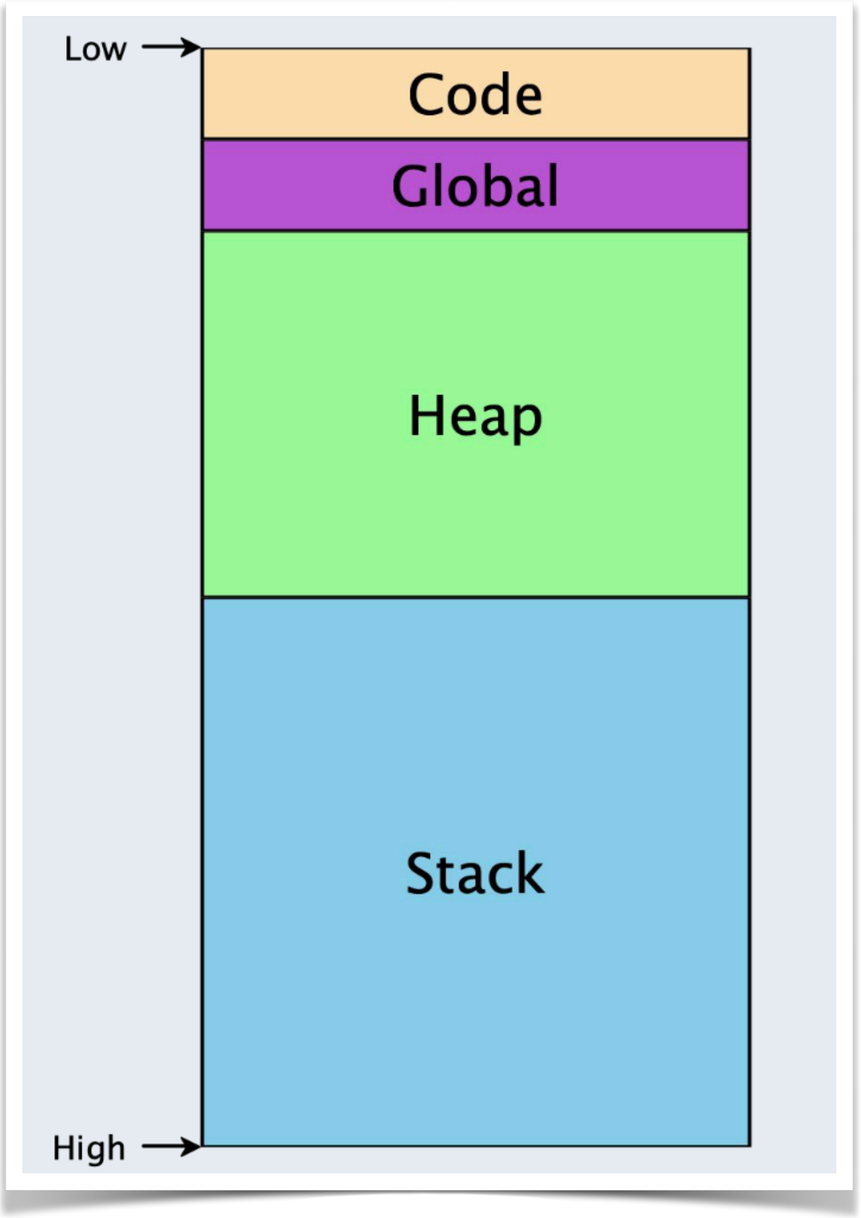
Memory in x86 processes is divided into 4 portions

1. Read-only memory containing the source code. (.text section)

2. Globals

3. Heap

4. The call Stack



x86 Memory Conventions

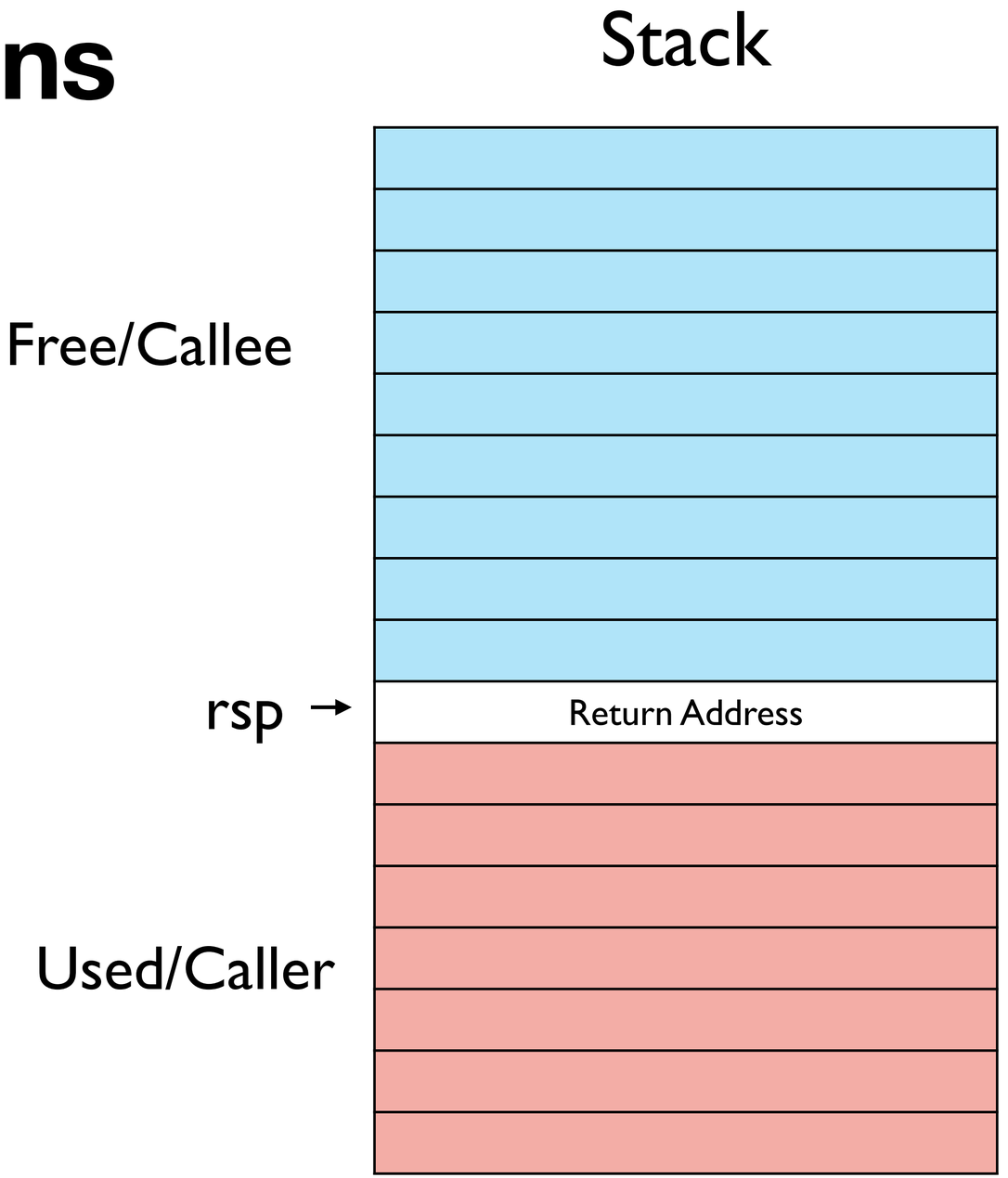
We access the stack using the "stack pointer" rsp.

The calling convention dictates that when a function is called, the stack pointer

1. Points to the return address of the caller

2. Lower memory addresses are free for the callee to use

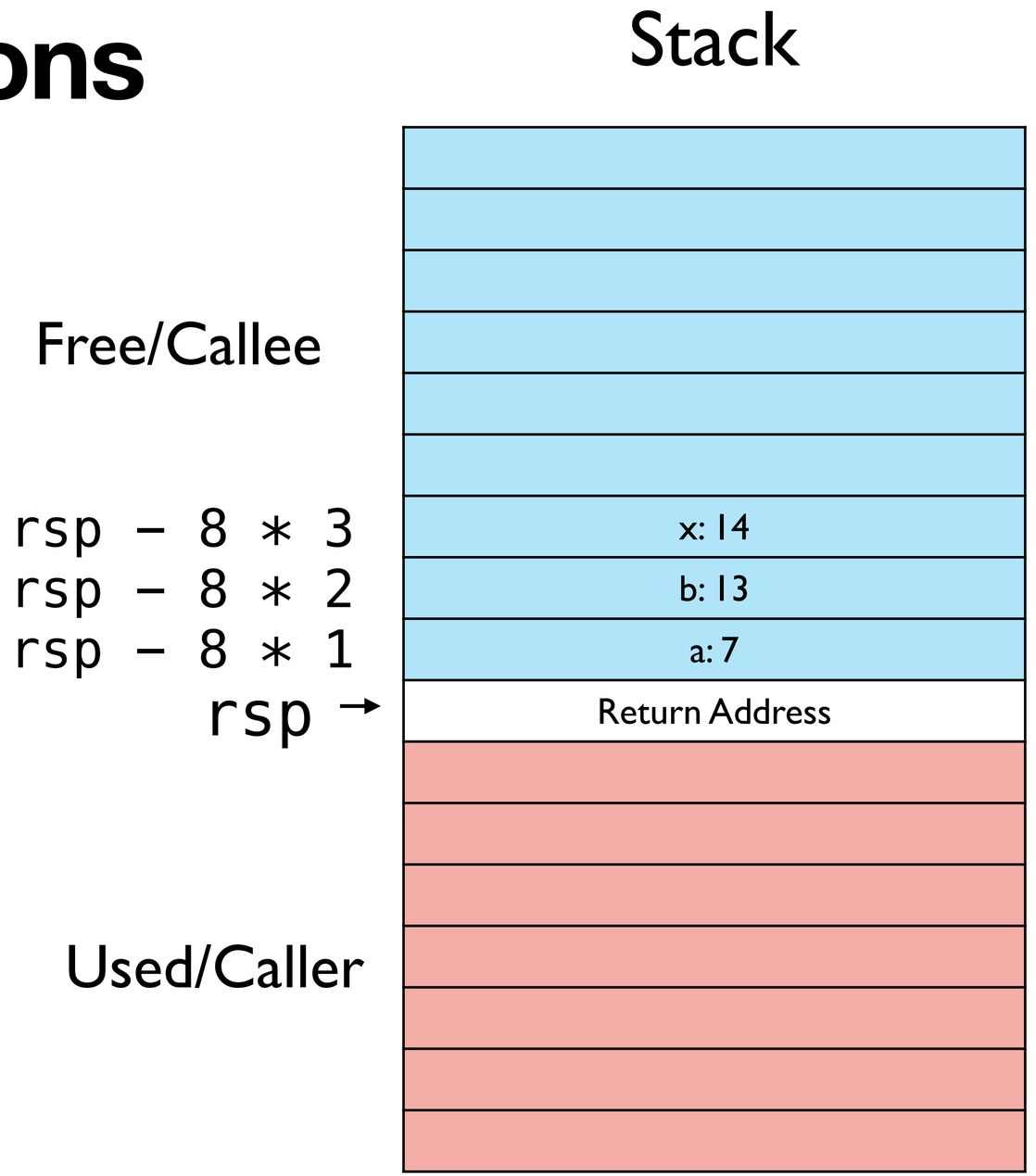
3. Higher memory addresses are owned by the caller



x86 Memory Conventions

We use the free space on the stack to store our local variables

```
let a = 7 in
let b = 13 in
let x = add1(a) in
add1(x)
```



Semantics

In an expression e1 op e2, do we evaluate e1 and then e2 or vice-versa?

Does it make a difference in Adder?

Does it make a difference in realistic extensions of Adder?

```
print(6) * print(7)
```

Compiling Binary Operations

```
(2 - 3) + (4 * 5)
```

```
mov rax, 2
sub rax, 3
?????
```

compound expressions have **implicit** intermediate results

solution: translate to a form where these intermediate results are explicit, and operations are only ever applied to **immediate** expressions (constants/variables)

```
let first = 2 - 3 in
let second = 4 * 5 in
first + second
```

Static Single Assignment v1: Basic Blocks

SSA programs aren't written by humans so they don't need a "concrete syntax"

but to make debugging easier, we will print SSA programs in the style shown below:

```
entry(x):
    y = add 2 x
    z = sub 18 3
    w = mul y z
    ret w
```

Static Single Assignment v1: Basic Blocks

Now we've reduced the compilation to two tasks:

1. "Lowering" our AST into an SSA program

2. Producing x86 assembly from an SSA program

Adder to SSA

Summary:

Translate Adder to SSA using **continuation-passing style:** expression lowering function is parameterized by a **continuation** consisting of

1. the name of the destination variable for the result.

2. a block of code to run **after** the compiled code places the result in the destination.

Need to generate **unique** names in this process to make sure that the generated variable names are all distinct and distinct from the original program variables

Boa

Snake v0.2: "Boa"

⟨*expr*⟩: ...

| `if` ⟨expr⟩ `:` ⟨expr⟩ `else:` ⟨expr⟩

Examples, Semantics

We only have one datatype of integers, no separate booleans. We'll use C's convention: 0 is false and everything else is true

Concrete Syntax	Answer
`if 5: 6 else: 7`	6
`if 0: 6 else: 7`	7
`if sub1(1): 6 else: 7`	7

Examples, Semantics

Again we have added if as an **expression** form (like Rust), so we need to handle cases like

```
(if x: 6 else: 8) + (if y: x else: 3)
```

similar to C's ternary operator `x ? 6 : 8`

For this reason, if expressions always have an else branch

Examples, Semantics

We want to ensure that our if expressions only evaluate **one** of the two branches at runtime, and not both.

How would you test that you did this correctly? What kinds of programs would behave differently if you always evaluated both branches?

```
if x:
  print(1)
else:
  print(0)
```

```
let x = 1 in
if x:
  7
else:
  infinite-loop
```

Control Flow in x86

x86 Instruction Semantics

So far, instructions execute in sequence. Why?

The instruction to execute is determined by a special register, the **instruction pointer "rip"**.

> in our abstract machine, each execution step starts by interpreting the memory at **[rip]** as a binary encoding of an assembly code instruction.
>
> Most instructions (mov, add, etc) increment rip by the size of the encoded instruction, meaning at the next step the instruction pointer will execute the instruction after it in memory
>
> What instruction have we seen so far that works differently?

x86 Instruction Semantics

So when we look at our code, we should think of it that we are looking at that code laid out in memory.

Assembly code **labels** give names to memory addresses.

```
entry:
  mov rax, rdi
  sub rax, 1
  cmp rax, 0
  je thn
els:
  mov rax, 7
  ret
thn:
  mov rax, 6
  ret
```

x86 Instructions: jmp

```
jmp loc
```

Semantics: sets the instruction pointer to loc.

Often loc is a **label** for another instruction in the same assembly file, but it doesn't have to be, it can be a register, or a memory location, or even a constant (almost certainly will crash in that case)

x86 Instructions: *jcc*

jcc loc

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

Semantics: sets rip to loc **if the condition code is satisfied**, otherwise increment rip like a sequential instruction.

x86 RFLAGS

The x86 abstract machine includes a register **rflags**, which like **rip** is manipulated as a side-effect of many instructions.

rflags is a 64-bit register, each bit acting as a boolean flag. Most of these are irrelevant to our compiler (or unused). The most relevant to us are

- OF "overflow flag": 1 if an overflow occurs, otherwise 0

- SF "sign flag": 1 if the output is negative, otherwise 0

- ZF "zero flag": 1 if the output is zero, otherwise 0

x86 RFLAGS

The x86 abstract machine includes a register **rflags**, which like **rip** is manipulated as a side-effect of many instructions.

mov does not affect flags

add, sub, imul, other arithmetic expressions do:

```
mov rax, 15
mov rcx, 17
sub rax, rcx
```

```
OF: 0
SF: 1
ZF: 0

rax: -2
rcx: 17
```

x86 Instruction: cmp

Often we want to set **rflags**, but not actually store an arithmetic result:

```
cmp arg1, arg2
```

"compare instruction". Behaves like **sub** for the purposes of setting flags, but does **not** update arg1

```
mov rax, 15
mov rcx, 17
cmp rax, rcx
```

```
OF: 0
SF: 1
ZF: 0

rax: 15
rcx: 17
```

x86 Instruction: test

Often we want to set **rflags**, but not actually store an arithmetic result:

```
test arg1, arg2
```

"test instruction". Behaves like a bitwise **and** for the purposes of setting flags, but does **not** update arg1. Useful for checking certain bits are set

x86 Condition codes

Condition codes interpret the flags as a boolean formula. Mnemonic makes the most sense if we have just run a **sub** or **cmp** operation

- e (equal): ZF

- ne (not equal): ~ ZF

- l (less than): OF ^ SF

- le (lesser or equal): (OF ^ SF) | ZF

- g (greater than): ~ le = ~ ((OF ^ SF) | ZF)

- ge (greater or equal): ~ l = ~ (OF ^ SF)

x86 Instructions: *jcc*

```
jcc loc
```

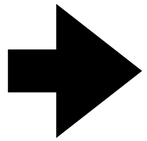
Actually a family of instructions, where **cc** is a **condition code**

Semantics: sets rip to loc **if the condition code is satisfied**, otherwise increment rip like a sequential instruction.

```
je loc

jle loc

jg loc

...
```

x86 Conditional Control Flow: Example

```
def main(x):
  if sub1(x):
    6
  else:
    7
```



```
entry:
  mov rax, rdi
  sub rax, 1
  cmp rax, 0
  jne thn
els:
  mov rax, 7
  ret
thn:
  mov rax, 6
  ret
```

SSA

Previously:

- one single block of operations ending in a return
- compiled to a block of sequential assembly labeled entry, ending in a ret

Extend as follows:

- add ability to define additional labeled blocks called **basic blocks**
- add ability to end a block by **branching** rather than returning

SSA Concrete Syntax

```
entry(x):
  thn:
    ret 6
  els:
    ret 7
  sub1_arg = x
  cond = sub sub1_arg 1
  cbr cond thn els
```

Compiling Basic Blocks to x86

For each basic block, we will emit a block of assembly code with a label corresponding to the name of the block.

Need to ensure that the sub-blocks are emitted after the instructions for the current block.

Conditional branches can be encoded using a mix of x86 conditional jumps and unconditional jumps

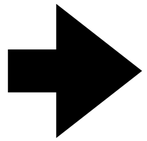
Compiling Basic Blocks to x86

```
entry(x):
  thn:
    ret 6
  els:
    ret 7
  sub1_arg = x
  cond = sub sub1_arg 1
  cbr cond thn els
```

```
entry:
        mov [rsp + -8], rdi
        mov rax, [rsp + -8]
        mov [rsp + -16], rax
        mov rax, [rsp + -16]
        mov r10, 1
        sub rax, r10
        mov [rsp + -24], rax
        mov rax, [rsp + -24]
        cmp rax, 0
        jne thn#0
        jmp els#1
thn#0:
        mov rax, 6
        ret
els#1:
        mov rax, 7
        ret
```

Compiling Conditionals to (Sub-)blocks

```
def main(x):
  if sub1(x):
    6
  else:
    7
```



```
entry(x):
  thn:
    ret 6
  els:
    ret 7
  sub1_arg = x
  cond = sub sub1_arg 1
  cbr cond thn els
```

Conditionals and Continuations

```
enum Exp {
  ...
  If { cond: Box<Exp>, thn: Box<Exp>, els: Box<Exp> }
}
```

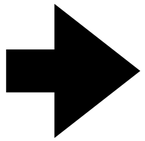
Strategy:

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

Compiling Conditionals to (Sub-)blocks

```
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'
```

Conditionals and Continuations

This works if the result of the if expression is to be returned, but what if it's more complex:

```
let x = (if y: 5 else: 6) in
x * 3
```

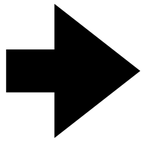
We need to also account for the **continuation** of the if expression!

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
```



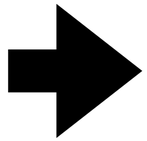
```
thn%uid:
  ... thn code
els%uid':
  ... els code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Copying Continuations

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

+

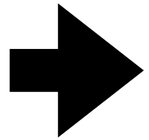
```
... continuation code
```



```
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

```
let x = (if y: 5 else: 6) in
x * 3
```



```
thn%0:
  x%2 = 5
  res%3 = x%2 * 3
  ret res%3
els%1:
  x%4 = 6
  res%3 = x%4 * 3
  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?

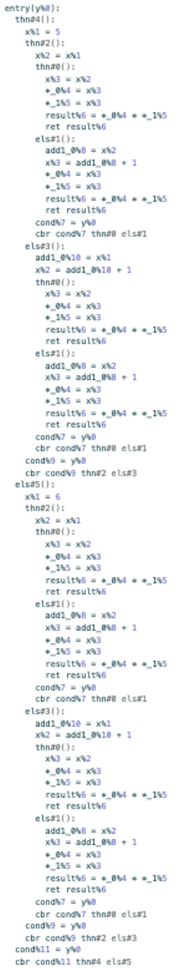
Exponential Blowup in Copying Continuations

```
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?



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.

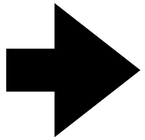
Join Points

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

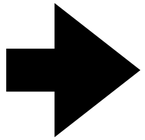
How can we extend our IR to express join points?

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
```



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

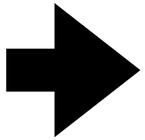
x * x

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

```
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
```

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()
```

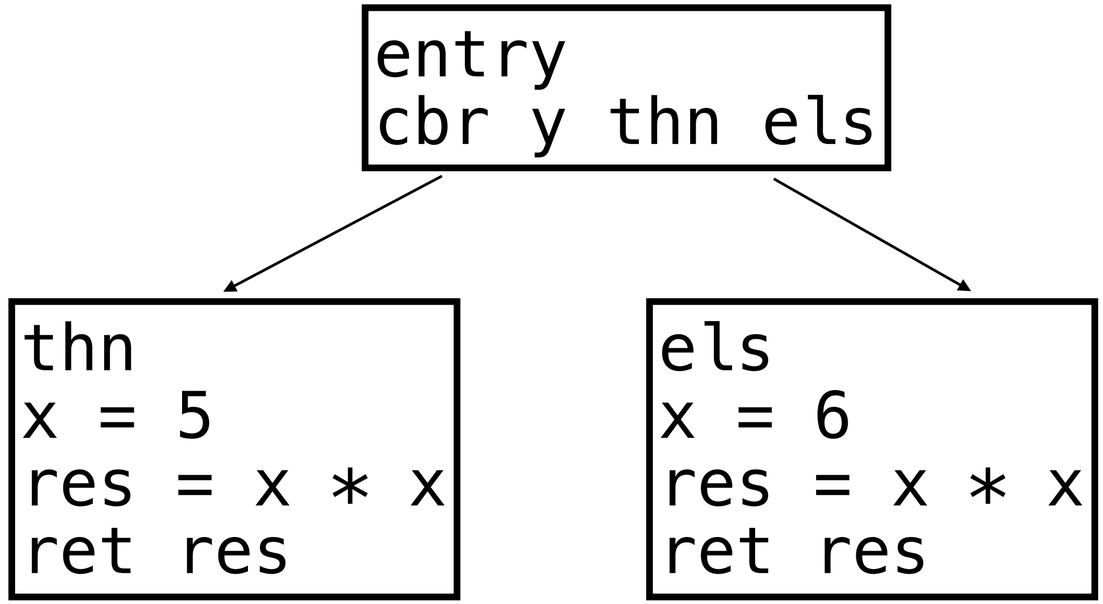
Represent the **continuation** directly in the syntax: a block can have **parameters** just like a continuation has an input variable.

Directly allow us to turn continuations into blocks

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
```



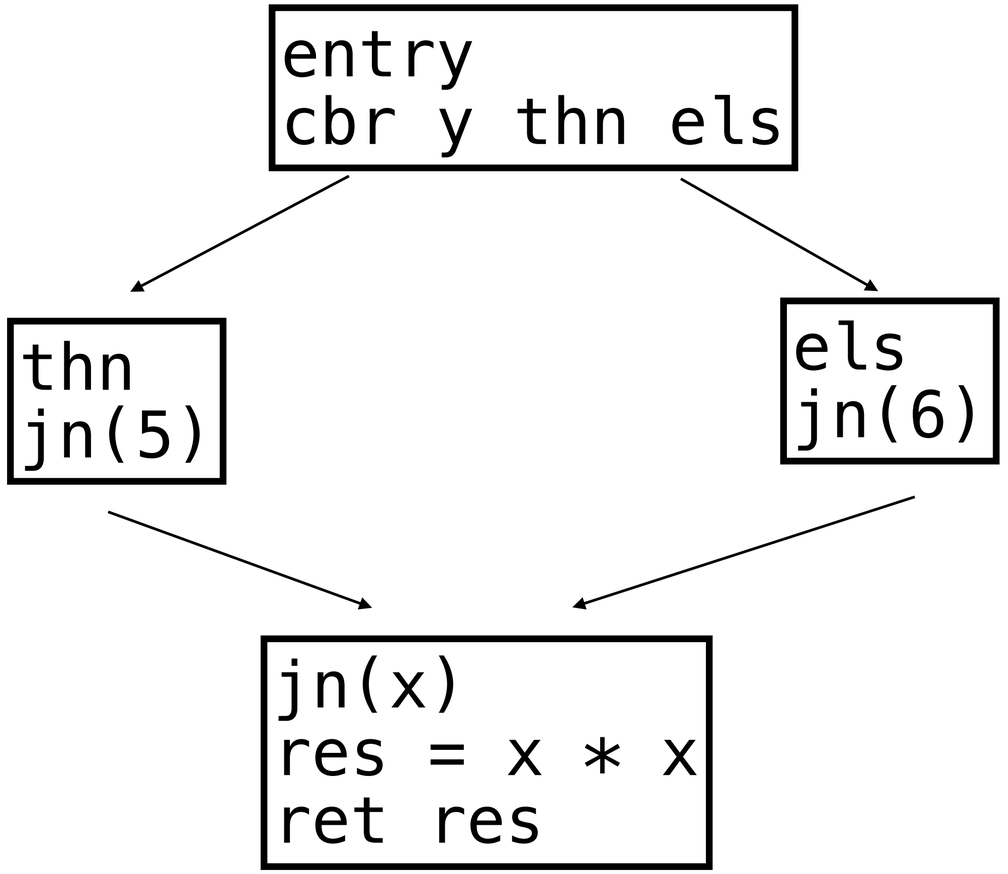
Nodes of CFG: basic blocks
edges are branches

Control Flow Graph

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

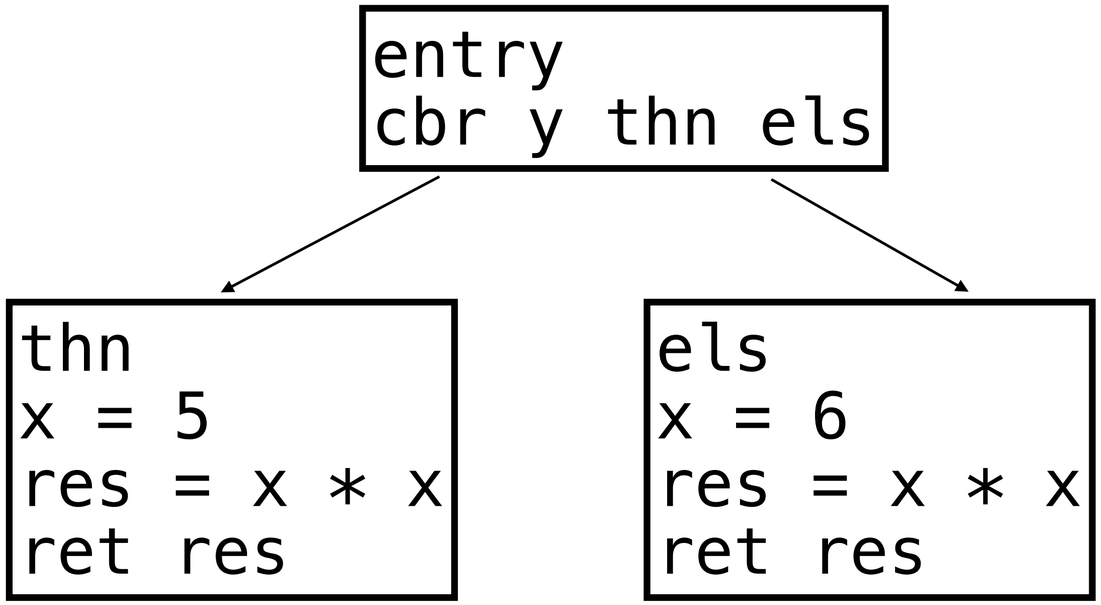
Join point: multiple predecessors

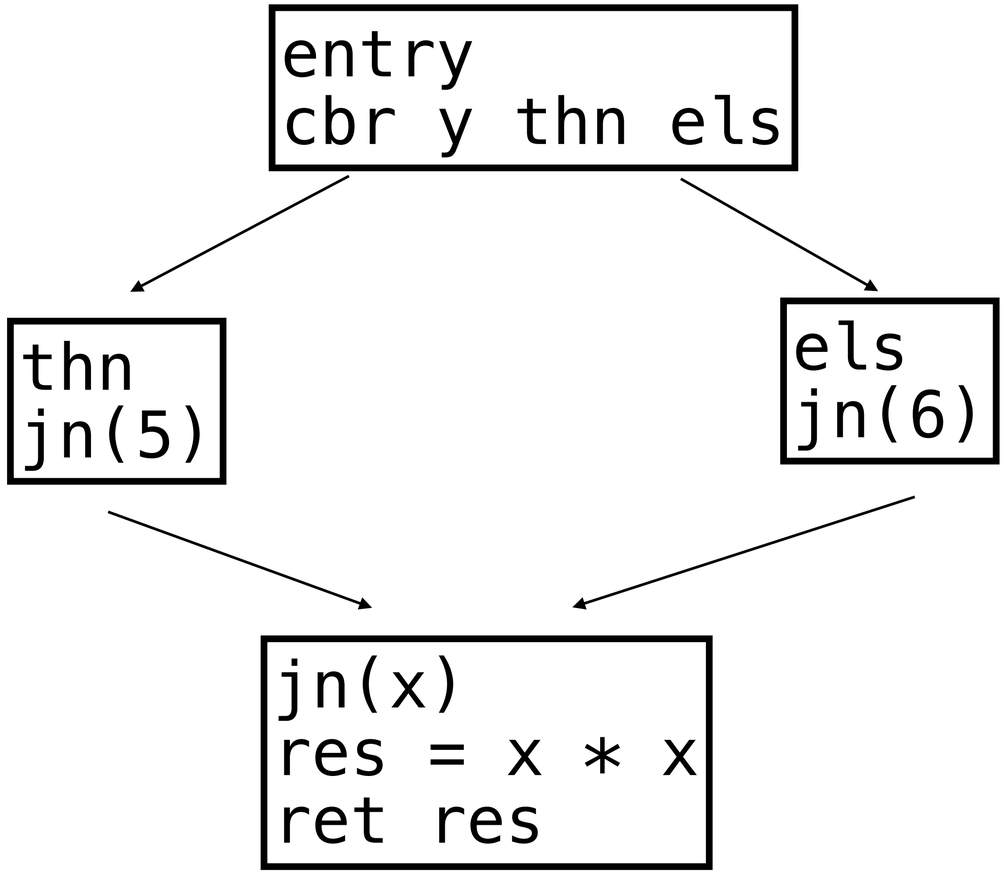
```
entry(y%0):
  jn#2(x%1):
    result%4 = x%1 * x%1
    ret result%4
  thn#0():
    br jn#2(5)
  els#1():
    br jn#2(6)
  cond%5 = y%0
  cbr cond%5 thn#0() els#1()
```



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.

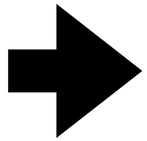
We can adapt our scope checker from the Snake language AST to the SSA programs. Gives us a "linting" pass that can help us find bugs if we accidentally made ill-formed SSA programs. If we implemented our compiler correctly, this should always succeed, but can be helpful for debugging.

Compiling Conditionals by Copying Continuations

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

+

```
... continuation code
```



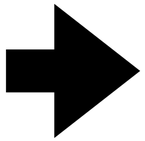
```
thn%uid:
  ... thn code
  ... continuation code
els%uid':
  ... els code
  ... continuation code
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

Compiling Conditionals by Generating Joins

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

+

```
... continuation code
```

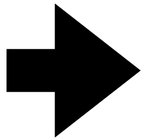


```
jn%uid''(x): ; continuation parameter
  ... continuation code
thn%uid:
  ... thn code
  br jn%uid''(thn_res)
els%uid':
  ... els code
  br jn%uid''(els_res)
... cond code
cond_result%uid'' = ...
cbr cond_result%uid'' thn%uid els%uid'
```

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)
```

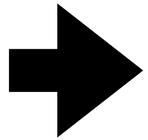


```
mov rax, imm1
mov [rsp - offset(x1)], rax
mov rax, imm2
mov [rsp - offset(x2)], rax
mov rax, imm3
mov [rsp - offset(x3)], rax
jmp l
```

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.

Should Conditional Branches be allowed to have arguments?

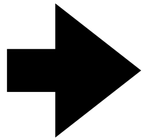
```
cbr x l1 l2
```



```
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)
```

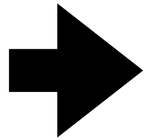


```
mov rax, [rsp - offset(x)]
cmp rax, 0
mov rax, [rsp - offset(y1)]
mov [rsp - offset(v1)], rax
mov rax, [rsp - offset(y1)]
mov [rsp - offset(v1)], rax
jne l1
mov rax, [rsp - offset(z)]
mov [rsp - offset(w)], rax
jmp l2
```

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

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

⟨*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

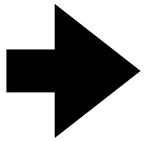
Compile each function definition directly to a corresponding block.

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

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

Imperative Snake Language

Imperative Snake Language "Imp"

- Mutable variables

- statement-expression distinction

- while loops

- return/break/continue

```
var m = 100;
var n = 25;
while !(m == n) {
  if m < n {
    n := n - m
  } else {
    m := m - n
  }
};
return m
```

Imperative Snake Language

concrete syntax

⟨*block*⟩:

| ⟨statement⟩
| ⟨statement⟩ `;` ⟨statement⟩

⟨*statement*⟩:

| `var` ***IDENTIFIER*** `=` ⟨expr⟩
| ***IDENTIFIER*** `:=` ⟨expr⟩
| `if` ⟨expr⟩ `{` ⟨block⟩ `}`
| `if` ⟨expr⟩ `{` ⟨block⟩ `}` `else` `{` ⟨block⟩ `}`
| `while` ⟨expr⟩ `{` ⟨block⟩ `}`
| `continue`
| `break`
| `return` ⟨expr⟩

⟨*expr*⟩:

| ***IDENTIFIER***
| ***NUMBER***
| `true`
| `false`
| `!` ⟨expr⟩
| ⟨prim1⟩ `(` ⟨expr⟩ `)`
| ⟨expr⟩ ⟨prim2⟩ ⟨expr⟩
| `(` ⟨expr⟩ `)`

Imperative Snake Language

abstract syntax

```
pub enum Block {
    End(Box<Statement>),
    Sequence(Box<Statement>, Box<Block>),
}
```

```
pub enum Statement {
    VarDecl(String, Expression),
    VarUpdate(String, Expression),
    If(Expression, Block, Block),
    IfElse(Expression, Block, Block),
    While(Expression, Block),
    Continue,
    Break,
    Return(Expression),
}
```

```
pub enum Expression {
    Var(String),
    Num(i64),
    Bool(bool),
    Prim(Prim, Vec<Expression>),
}
```

Imperative to SSA

Step 1: Expressions, variable declarations

Step 2: variable updates

Step 3: Join Points

Step 4: Loops

Step 5: Break, Continue, Return

Imperative to SSA

Step 1: Expressions, variable declarations

Expressions are defined just as in Adder: generate temporaries and use continuations to turn tree of operations into straightline code

Variable declarations are implemented just as with Let: a var declaration in Imp becomes a variable assignment in SSA

```
var x = 10;
var p = (x * x) + 5 * x + 7;
...
```

```
x = 10
tmp0 = x * x
tmp1 = 5 * x
tmp2 = tmp0 + tmp2
p = tmp2 + 7
...
```

Imperative to SSA

Step 2: Variable Updates

```
var x = 10;
x := (x * 2) + 1;
x := x + x
...
```

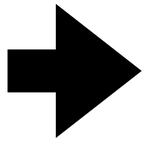
how to compile to SSA?

idea: the updated x acts like it's shadowing the original. Treat it as an assignment to a new variable

Imperative to SSA

Step 2: Variable Updates

```
var x = 10;
x := (x * 2) + 1;
x := x + x
...
```



```
x0 = 10
tmp0 = x0 * 2
x1 = tmp0 + 1
x2 = x1 + x1
...
```

Keep track in an environment of the current "version" of each variable in scope

Imperative to SSA

Step 2: Variable Updates

Simple idea: replace mutable updates with assignments to a new variable

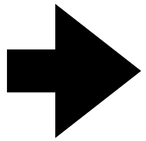
in straightline code, mutable variables are just shadowing!

Imperative to SSA

Step 2: If

Imperative to SSA

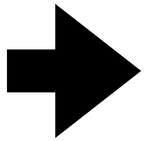
```
...
var x = 10;
if y {
  x = x + 1
} else {
  x = x * 2
  x = x - 1
}
return x
```



```
x0 = 10
thn():
  x1 = x0 + 1
  br ??
els():
  x2 = x0 * 2
  x3 = x2 - 1
  br ??
cbr y thn() els()
```

Imperative to SSA

```
...
var x = 10;
if y {
  x = x + 1
} else {
  x = x * 2
  x = x - 1
}
return x
```



Join points!

```
x0 = 10
jn(x4):
  ret x4
thn():
  x1 = x0 + 1
  br jn(x1)
els():
  x2 = x0 * 2
  x3 = x2 - 1
  br jn(x3)
cbr y thn() els()
```

Imperative to SSA

Step 2: If

Generate join points for if statements.

In an imperative program, join points are parameterized not just by a single variable, but by as many as can be updated in the two branches.

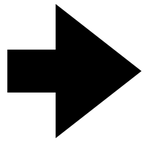
Need to calculate which variables ot include in the join point:

Simplest algorithm is called **crude** ϕ-node insertion: add **every** variable that is in scope to the join point.

Rely on a later SSA-minimization pass to remove unnecessary parameters

Unnecessary Parameters

```
...
var x = 10;
var z = 7;
if y {
  x = x + 1
} else {
  y = x * 2
  x = x - 1
}
var w = z * x
return w + y
```



```
...
x0 = 10
z0 = 7
jn(x4, y1, z1):
  w = z1 * x4
  tmp = w + y1
  ret tmp
thn():
  x1 = x0 + 1
  br jn(x1, y0, z0)
els():
  y2 = x0 * 2
  x2 = x1 - 1
  br jn(x2, y2, z0)
cbr y0 thn() els()
```

Imperative to SSA

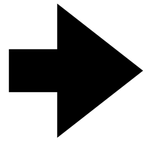
Step 4: while loops

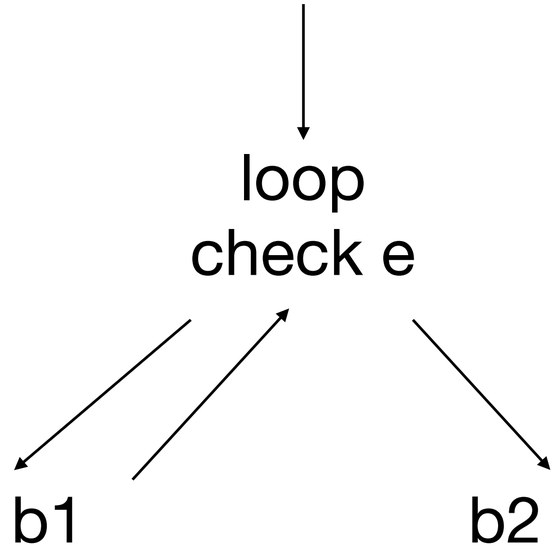
encode semantics using SSA blocks

which blocks in a loop are join points?

Imperative to SSA

```
while e {
   b1
}
b2
```

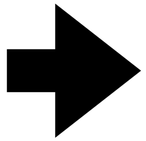




Notice: loop has 2 predecessors, so it is a join point, add block parameters

Imperative to SSA

```
while e {
  b1
}
b2
```



```
loop(...): ;; loop is a join point, include all in-scope vars
  done():
    ... ;; compiled code for b2
  body():
    ... ;; compiled code for b1
    br loop(...)
  ...
  c = ... ;; compiled code for e
  cbr c body() done()
br loop(...)
```

Imperative to SSA

Step 5: return, break, continue

Return is easy: just compile the expression and produce the ret terminator

Break, continue: depend on the **context**

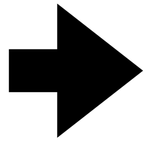
when we enter a while loop, we make blocks for the entry point and exit point

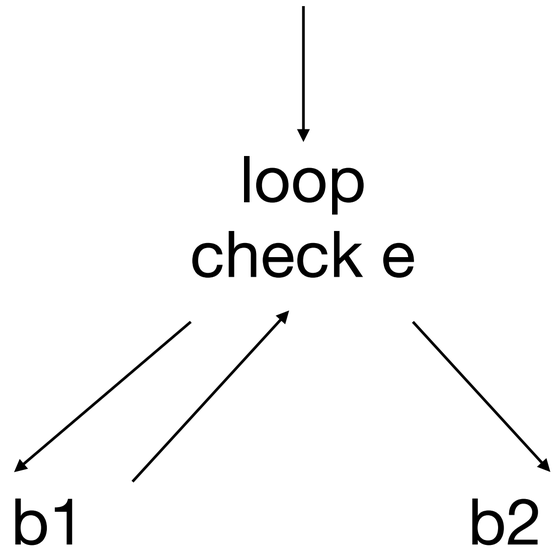
continue: branch to entry of loop

break: branch to exit of loop

Imperative to SSA

```
while e {
   b1
}
b2
```

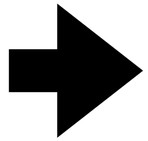


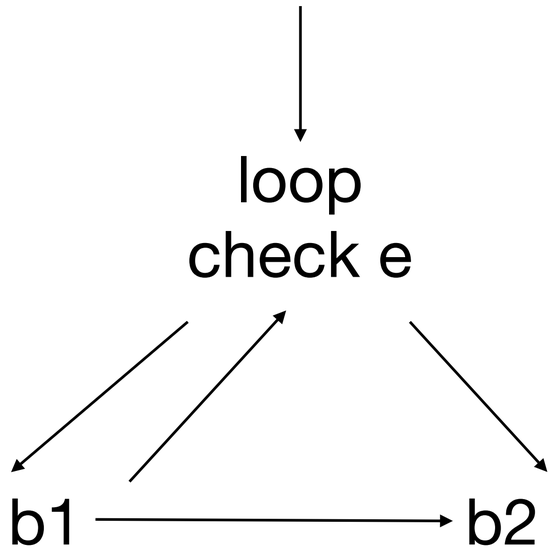


Notice: loop has 2 predecessors, so it is a join point, add block parameters

Imperative to SSA

```
while x != 0 {
  x := x - 1
  if y > 10 {
    break
  }
  ...
}
b2
```



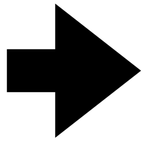


If we can break, then b1 can branch directly to b2

if break is used, b2 is **also** a join point

Imperative to SSA

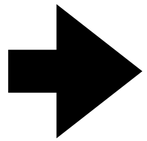
```
while e {
  b1
}
b2
```



```
loop(...): ;; loop is a join point, include all in-scope vars
  done(...): ;; done is a join point as well because of break
    ... ;; compiled code for b2
  body():
    ... ;; compiled code for b1
    br loop(...)
  ...
  c = ... ;; compiled code for e
  cbr c body() done(...)
br loop(...)
```

Imperative to SSA

```
var m = 100
var n = 25
while ! (m == n) {
  if m < n {
    n := n - m
  } else {
    m := m - n
  }
}
return m
```



```
m0 = 100
n0 = 25
loop(m2, n2):
  done(m1,n2):
    return m1
  body(m3, n3):
    lt():
      n4 = n3 - m3
      br loop(m3, n4)
    gt():
      m4 = m3 - n3
      br loop(m4, n3)
    b = m3 < n3
    cbr b lt() gt()
  c = m2 == n2
  d = not c
  cbr d body(m2, n2) done(m2, n2)
loop(m0, n0)
```

Cobra

State of the Snake Language

Adder: Straightline Code

Boa: Conditionals + Tail-called functions = arbitrary "local" control flow.

Good for implementing the "bodies" of functions, but missing key features:

1. Interaction with the Operating System (stdin/stdout, file I/O, random number generation)

2. Reusable sub-procedures (functions with non-tail calls)

Add these in **Cobra**

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

Want the ability to interact with the operating system

So far: add new primitives one at a time

More flexible: allow importing "extern" functions from our Rust stub.rs file

SSA Changes

Add extern declarations to top-level SSA program

Add call as a new operation in SSA (not a terminator!)

```
x = call f(x1,...)
```

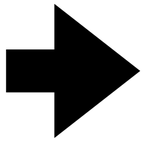
Change to lowering:

if a call to an **internal** function is a tail call, compile it as before as a branch with arguments

if a call to an **external** function is a tail call, compile it as a call and then return the result

SSA Changes

```
extern print(x)


def main(x):
  let y = x + 10
  print(y)
```



```
extern print(x)


main(x):
  y = x + 10
  res = call print(y)
  ret res
```

Code Generation

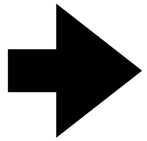
Each extern declaration becomes an x86 extern declaration

Function calls are compiled using the System V AMD64 Calling convention

Linking will fail unless the extern functions are implemented in stub.rs

SSA Changes

```
extern print(x)


main(x):
  y = x + 10
  res = call print(y)
  ret res
```



```
section .text
global entry
extern print

entry:
  ...
```

Non-tail Procedure Calls

When a function is called it needs to know **where** to return to, i.e., what its **continuation** is.

To implement this, procedure calls pass a **return address**. Think of this as an "extra argument" that is implicit in the original program.

When a function is called, it needs to know which registers and which regions of memory it is free to use

Use **rsp** as a pointer to denote which part of the stack is free space

Designate which registers are **volatile** or **non-volatile**

When implementing calls within a programming language we can decide these conventions for ourselves. When implementing calls that work with external code need to pick a standard **calling convention**

x86 Abstract Machine: The Stack

So far we have used rsp as a base pointer into our stack frame

But several instructions treat it as a "stack pointer"

x86 Instructions: push

```
push arg
```

Semantics:

```
sub rsp, 8
mov [rsp], arg
```

If rsp is the pointer to the "top" of the stack, this pushes a new value on top.

x86 Instructions: pop

```
pop reg
```

Semantics:

```
mov reg, [rsp]
add rsp, 8
```

If rsp is the pointer to the "top" of the stack, this pops the current value off of it

x86 Instructions: call

call loc

Semantics is a combination of **jmp** and **push**

sets rip to loc (like jmp loc)

and pushes the address of the next instruction onto the stack (like jmp next)

Example:

x86 Instructions: ret

```
ret
```

Semantics is a combination of **pop** and **jmp**

- pops the return address off of the stack and jumps to it
- like
  - pop r
  - jmp r
- except without using up a register r

Calling Convention

Calling Convention

When implementing a call into Rust, we need to use a common **calling convention**

We use the System V AMD 64 ABI

- Standard "C" calling convention for 64-bit x86 code on Linux/Intel Macs
- Has some idiosyncracies from supporting C code and SSE instructions

Calling Convention

A calling convention is a protocol that a caller and callee follow in order to implement a procedure call and return

Caller and callee need to agree on:

1. State of memory/registers when the callee begins executing

2. State of memory/registers once the callee has returned

So a calling convention is really a combination of a "calling" convention and a "returning" convention

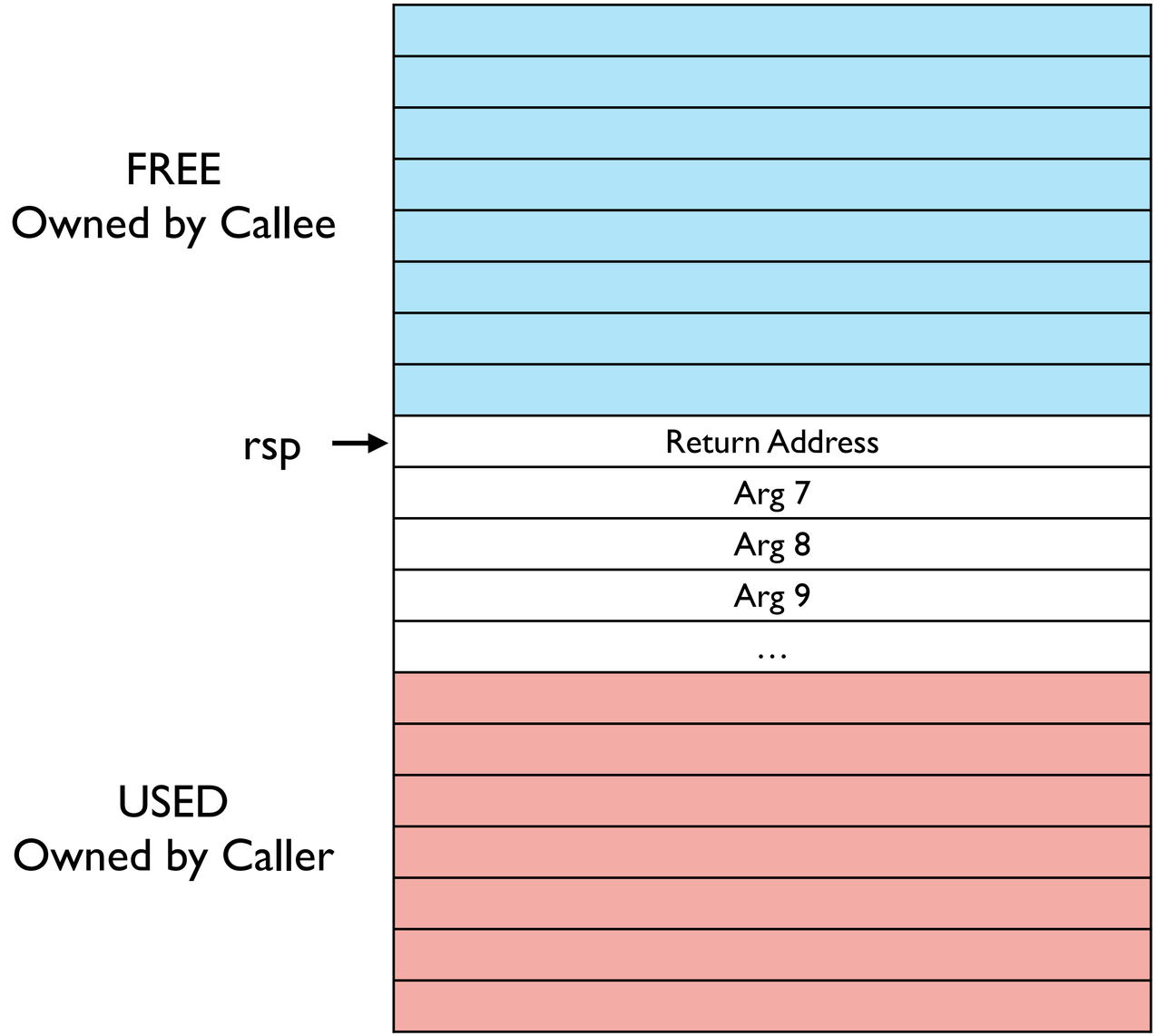
System V AMD 64

Calling protocol: When a called function starts executing the machine state is as follows:

1. Arguments 1-6 are stored in rdi, rsi, rdx, rcx, r8, r9
2. Arguments 7-N are stored in [rsp + 1 * 8], [rsp + 2 * 8],...[rsp + (N - 6) * 8]
3. rsp points to the return address.
4. Stack Alignment: rsp % 16 == 8

System V AMD 64

rdi	Arg 1
rsi	Arg 2
rdx	Arg 3
rcx	Arg 4
r8	Arg 5
r9	Arg 6
rsp	0xXX...X8



System V AMD 64

Returning protocol: When a called function returns to its caller

1. Return value is stored in rax
2. Registers rbx, rbp, r12-r15 are in their original state when the function was called **(non-volatile aka callee-save)**
3. Stack memory at higher addresses than rsp is in the original state when the function was called
4. Original value of rsp holds the return address, pop this address and jump to it

Volatile/Non-volatile Registers

A register is **volatile** if its value may be changed by a function call

This means the **callee** can set the register as they see fit, no promises on what its value will be when the callee returns

Also known as **caller-save** because if a local variable is stored in a volatile register it must be "saved" somewhere non-volatile if its value is needed after the call

A register is **non-volatile** it it must be preserved by a function call

This means the **callee** must ensure that when the function returns, the register has the same value as it did when the function began execution

Also known as **callee-save** because if the callee wants to use a non-volatile register, the original value must be saved somewhere and restored before returning

Volatile/Non-volatile Registers

Current Strategy:

We use registers as scratch registers, but always store local variable values on the stack

Should a scratch register be **volatile** or **non-volatile**?

volatile: we are free to change it without worrying about the caller

don't need to worry about saving it when we make a call because we don't store long-lived values in it

examples: rax, r10, r11, any argument register if we move its value to the stack

Revisit this once we implement **register allocation**

Stack Alignment

When a function is called, rsp % 16 == 8

Needed for certain SSE instructions which require data alignment

To ensure correct alignment: rsp % 16 == 0 **before** executing the **call** instruction (since call pushes an 8-byte address)

Caller cleanup

In the SysV AMD 64 calling convention, the **caller** is responsible for "cleanup" of the arguments.

That is, when a function returns, the arguments that are passed on the stack are still there, even though they are not necessarily needed

Why?

> Used to implement C-style variadic function. In C, a variadic function doesn't know how many arguments have been passed, so impossible for it to perform caller cleanup.

Downside:

> Impossible to perform tail call to a function that takes more stack-allocated arguments than the caller using SysV AMD 64 calling convention.

Stack Frame Management

When making a function call we need to ensure that the newly allocated stack frame is "above" all of our local variables

Stack Frame Management

The calling convention dictates an **interface** between the caller and the callee.

It does not dictate **internal** details of a function implementation, e.g., where in registers, memory, local variables are stored

2 common strategies for managing stack-allocated variables

1. (Modern) Use rsp as the base pointer of the stack frame

2. (C style) Use rbp as the base pointer of the stack frame and rsp as the pointer to the **top** of the stack frame

State of the Snake Language

Adder: Straightline Code

Boa: Conditionals + Tail-called functions = arbitrary "local" control flow.

Good for implementing the "bodies" of functions, but missing key features:

1. Interaction with the Operating System (stdin/stdout, file I/O, random number generation)

2. Reusable sub-procedures (functions with non-tail calls)

Add these in **Cobra**

State of the Snake Language

Adder: Straightline Code

Boa: Conditionals + Tail-called functions = arbitrary "local" control flow.

Good for implementing the "bodies" of functions, but missing key features:

1. Interaction with the Operating System (stdin/stdout, file I/O, random number generation)

2. Reusable sub-procedures (functions with non-tail calls)

Add these in **Cobra**

Running Examples

Non-tail calls

```
def main(x):
  def max(m,n):
    if m >= n: m else: n
  in
  max(10, max(x, x * -1))
```

non-tail and tail call of the same function

Running Examples

Non-tail calls

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

captured variables in non-tail called function

Design Goals

Non-tail calls

We want to support the ability to **call** or **tail call** our internally defined functions.

We want **tail calls** to be implemented the same way as in Boa: this ensures tail-recursive functions are still compiled efficiently.

We want **calls** to be implemented using the System V AMD64 calling convention. This allows us to compile calls to Rust or Cobra functions the same way, simplifying code generation.

Change to SSA

Previously we had one code block that would be called with the SysV calling convention: **main**

Generalize this to have many top level function blocks in SSA.
The body of a function block should immediately branch with arguments to an ordinary SSA block, which is compiled as before.

In code generation: compile these as moving the arguments from the SysV AMD64-designated locations to the stack.

Change to SSA: Abstract Syntax

An SSA program has three parts:
1. Extern declarations
2. Function blocks
3. Top-level Basic blocks

Side condition: one of the function blocks has the unmangled name "entry", corresponding to the main function in the source program.

All of these are globally scoped: functions can branch to any of the top-level blocks and vice-versa the blocks can call any of the functions.

```
pub struct Program {
    pub externs: Vec<Extern>,
    pub funs: Vec<FunBlock>,
    pub blocks: Vec<BasicBlock>,
}
```

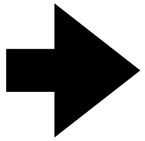
Change to SSA: Abstract Syntax

Function blocks always have the same structure: immediately branch to one of the top-level blocks

```
pub struct FunBlock {
    pub name: Label,
    pub params: Vec<VarName>,
    pub body: Branch,
}
```

SSA Generation Example

```
def main(x):
  def max(m,n):
    if m >= n: m else: n
  in
  max(10, max(x, x * -1))
```



```
block max_tail(m, n):
  ... as in Boa
block main_tail(x):
  tmp1 = x * -1
  tmp2 = call max_fun(x, tmp1)
  br max_tail(10, tmp2)
fun max_fun(m, n):
  br max_tail(m, n)
fun entry(x):
  br main_tail(x)
```

give the blocks and funs different names as we need to assign both of them labels in code generation

Functions vs Basic Blocks in SSA

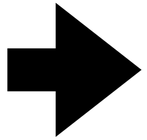
In SSA, we make a distinction between **functions** and **parameterized blocks**
Both have a label and arguments, but the way they are used and compiled is different

Functions can only ever be the target of a **call**, using the System V AMD64 ABI
Blocks can only ever be the target of a **branch**, where arguments are placed at stack offsets

Blocks can be nested as sub-blocks within other blocks, can refer to outer scope
Functions are only ever **top-level**, only variables in scope inside are arguments

Code Generation for Function Blocks

```
fun max_fun(m, n):
  br max_tail(m, n)
```



```
max_fun:
  mov [rsp - 8], rdi
  mov [rsp - 16], rsi
  jmp max_tail
```

Functions are just a thin wrapper around their blocks, mov arguments from where the calling convention dictates to where the block expects them to be (on the stack)

Variable Capture

```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```

x is "captured" by the function **pow** in that it is used in the function because it is in scope when the function **pow** is defined.

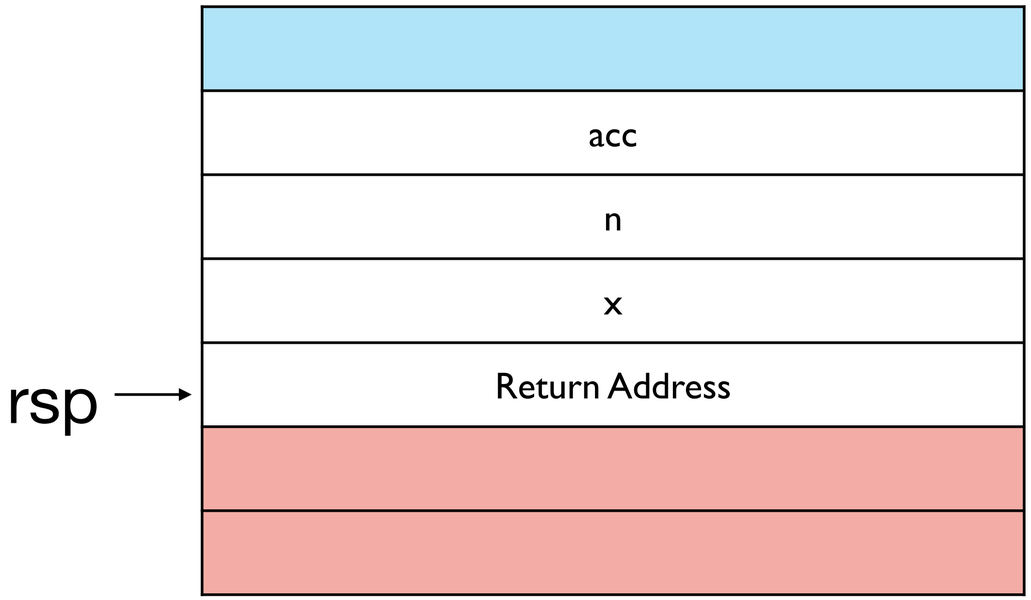
Why does this work?

Variable Capture

```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```

Variable Capture

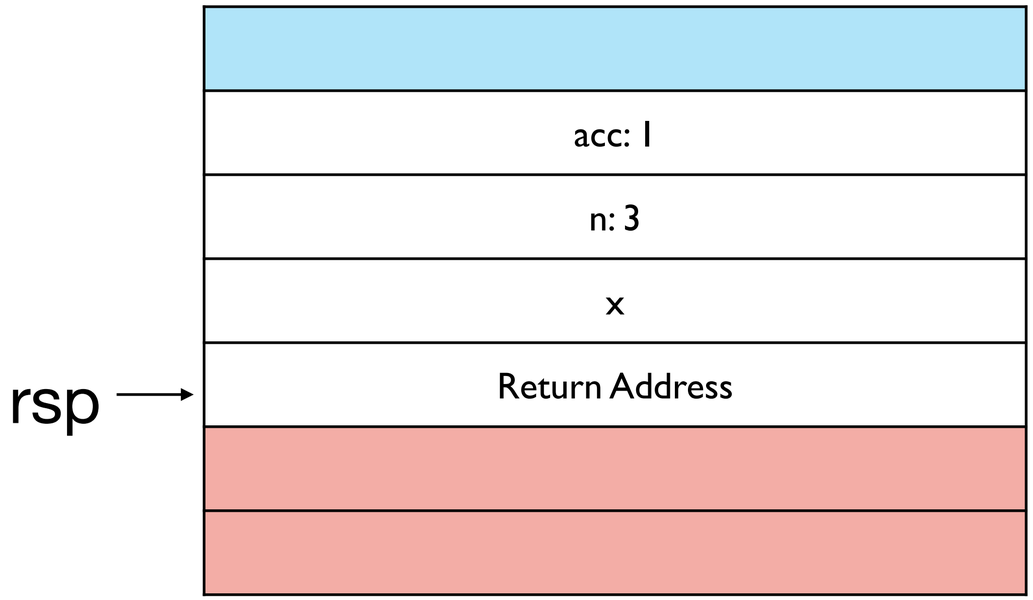
```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```



Variable Capture

```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```

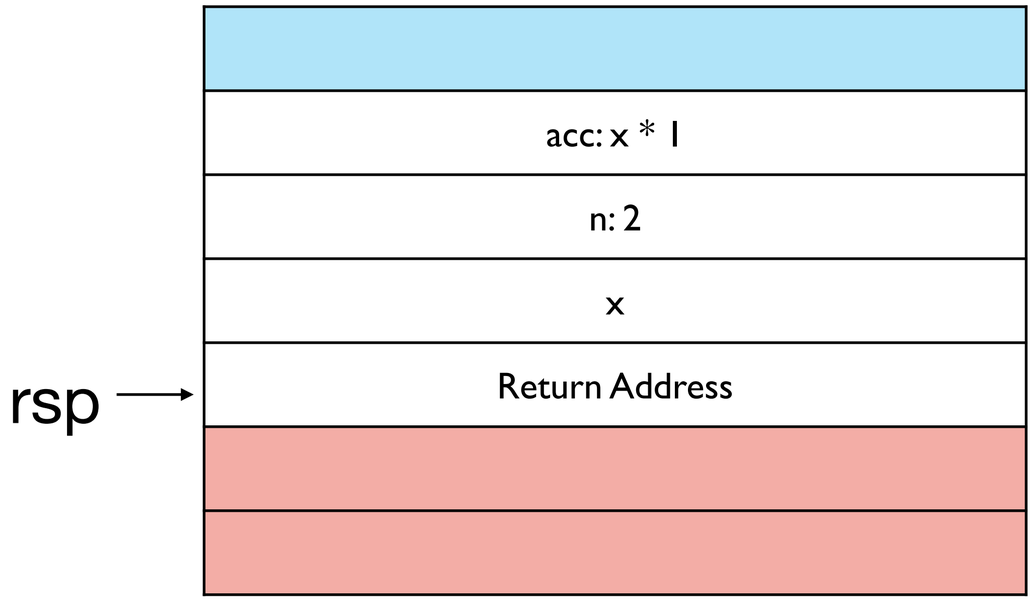
first iteration



Variable Capture

```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```

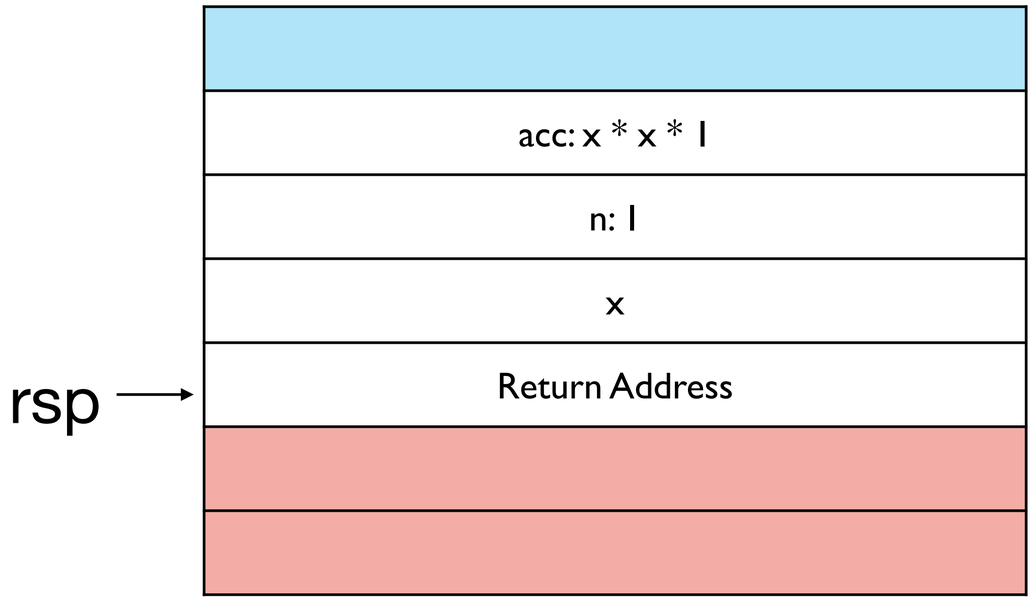
second iteration



Variable Capture

```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```

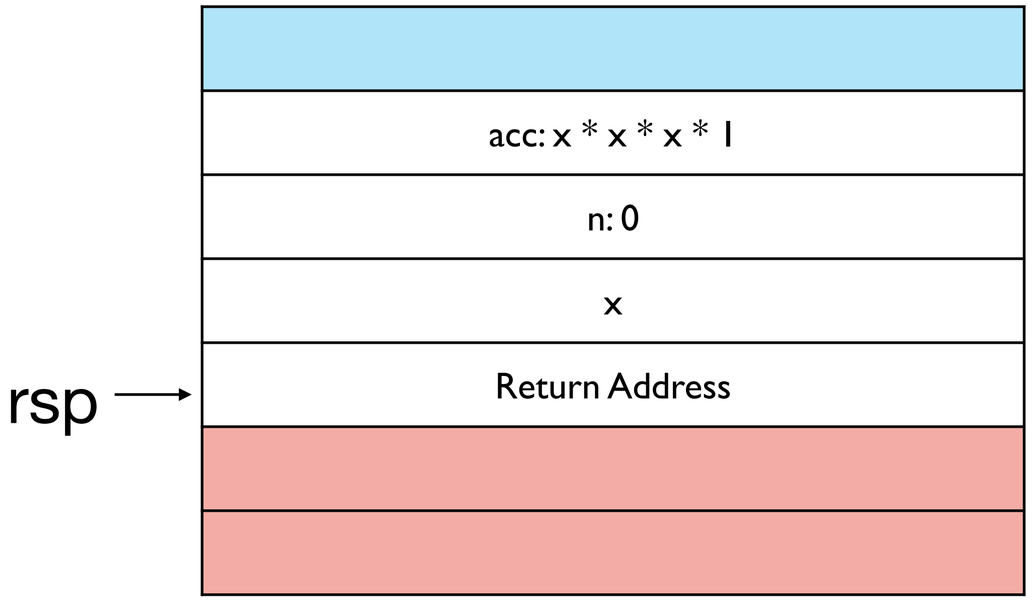
third iteration



Variable Capture

```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```

final iteration



Variable Capture

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

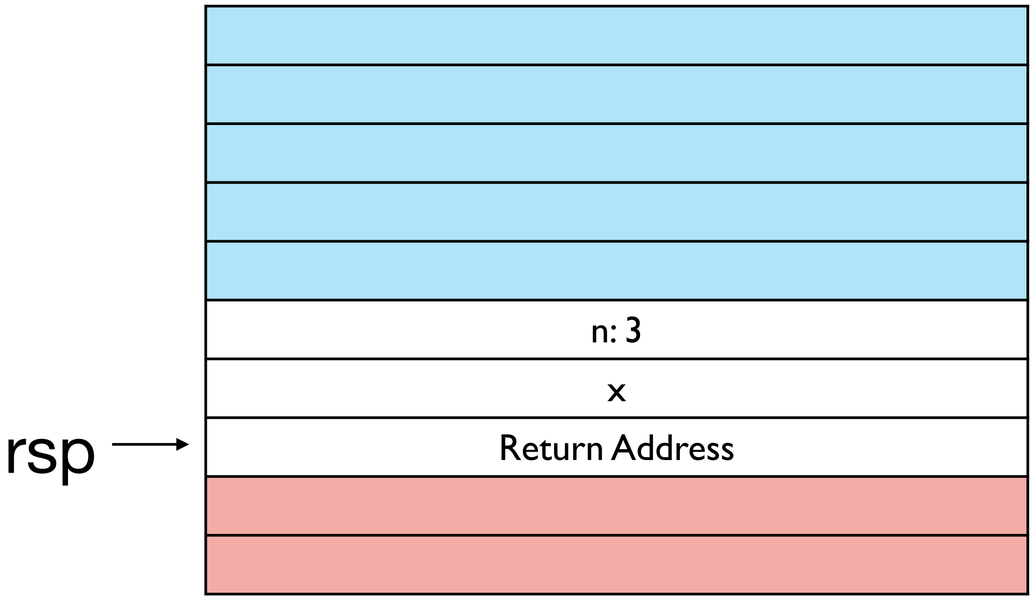
x is "captured" by the function **pow** in that it is used in the function because it is in scope when the function **pow** is defined.

But now **pow** is not tail-recursive. What happens?

Variable Capture

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

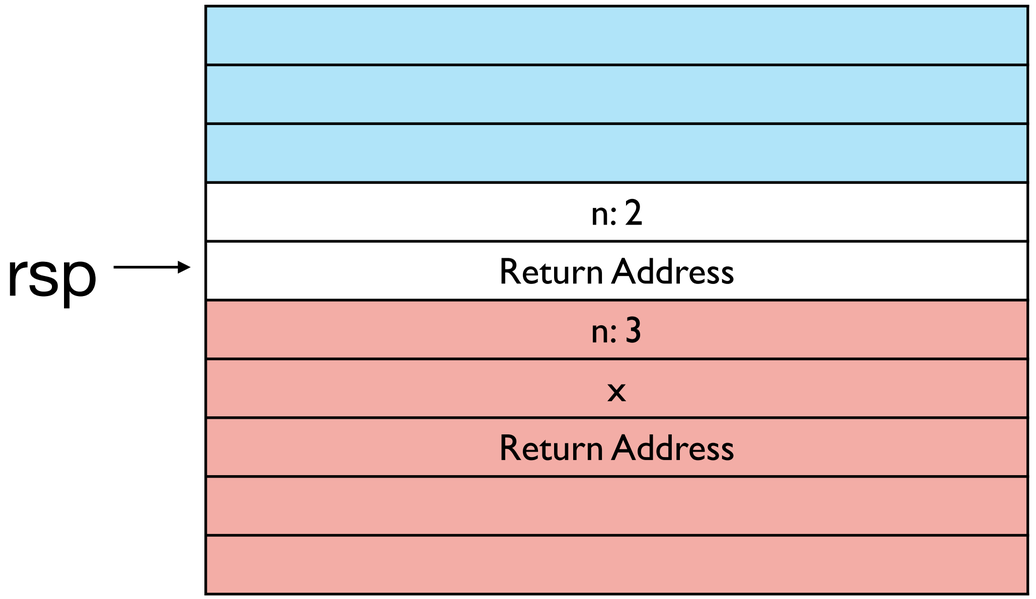
initial tail call



Variable Capture

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

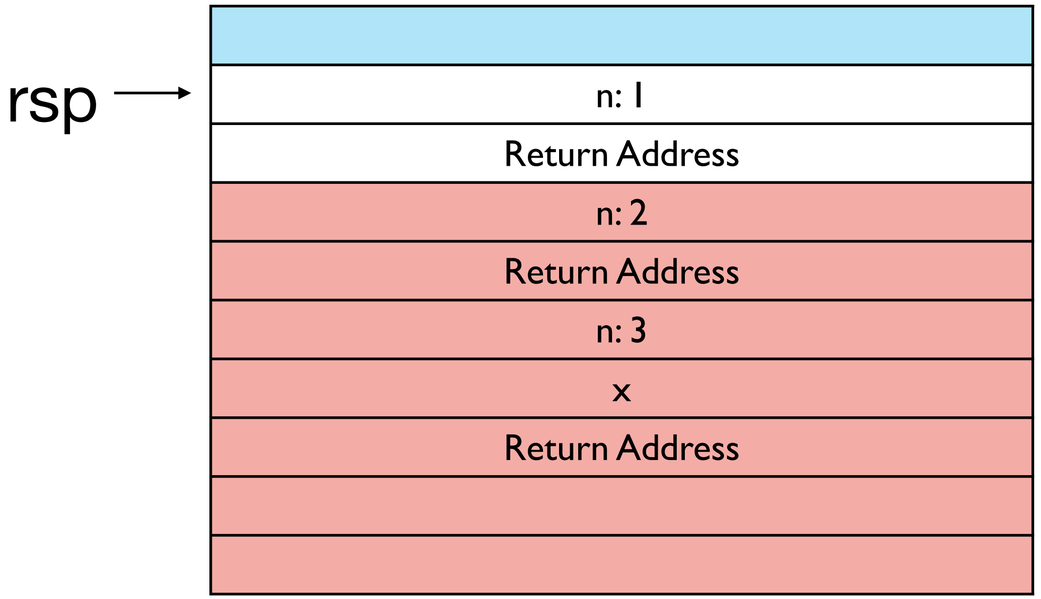
first non-tail recursive call



Variable Capture

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

second non-tail recursive call



Variable Capture

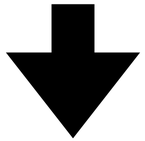
Variable capture in a tail-called function is not a problem, as the captured variables are still available in our local stack frame.

Variable capture in a called function is an issue: the distance from the current stack pointer to the location of the captured variable is not statically determined

Can solve this by **copying** the value into each stack frame.
Implement as a code transformation: add all the captured variables as **extra arguments**. This process is called **lambda lifting**.

Lambda Lifting (As AST to AST transform)

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```



```
def pow(x, n):
  if n == 0: 1 else x * pow(x, n - 1)
def main(x):
 pow(x, 3)
```

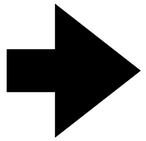
Lambda Lifting

Instead of an AST to AST transform, incorporate this in our AST to SSA transformation.

In the lowering to SSA, any function that is called must be "lifted" to the top-level, where all captured variables are added as extra arguments, and all **calls** or **branches** pass the additional arguments.

Lambda Lifting (As AST to SSA transform)

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```



```
block pow_tail(x, n):
  b = n == 0
  thn():
    ret 1
  els():
    n2 = n - 1
    r = call pow_call(x, n)
    tmp = x * r
    ret tmp
  cbr b thn() els()
block main_tail(x):
  br pow_tail(x, 3)
fun pow_call(x, n):
  br pow_tail(x, n)
fun main(x):
  br main_tail(x)
```

Lambda Lifting: Details

To implement lambda lifting, we need to address two questions.
1. Which functions need to be lifted?
2. Given a function to be lifted, which arguments need to be added?

For both of these we need to consider
1. Correctness
   Must ensure every function that must be lifted is lifted and that every argument that must be added is added, but we can **over-approximate** by lifting more than necessary and adding more arguments than necessary
2. Efficiency
   Lifting too many functions or adding too many arguments can impact runtime and space usage in our generated programs.

Correctness is **always** a must. Efficiency is best-effort.

Lambda Lifting: Who to Lift

What definitions need to be lifted?
- any function that is (non-tail) called needs to be lifted
- the tail-callable version of that function also needs to be lifted

Any other functions?

Yes: any function that is tail called by a lifted function must also be lifted, even if it is never non-tail called

Lambda Lifting: Who to Lift

Which of e,f,g,h,k need to be lifted in this example?

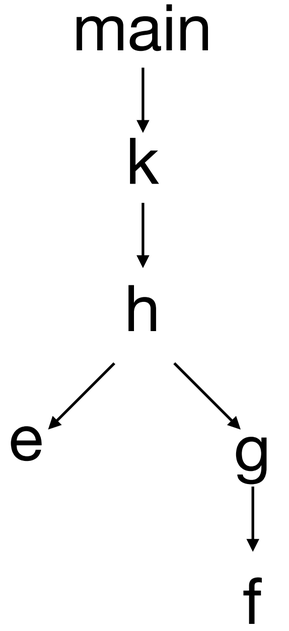
Answer:
h must be lifted because it is non-tail called
g, e must be lifted because they are tail called by a lifted function
f must be lifted because it is tail called by a lifted function
k does not need to be lifted

```
def main(a):
  def e(): a * 2 in
  def f(): a in
  def g(): f() in
  def h(b): if b: g() else: e() in
  def k(): h(a) + 1 in
  k()
```

Lambda Lifting: Who to Lift

```
def main(a):
  def e(): a * 2 in
  def f(): a in
  def g(): f() in
  def h(b): if b: g() else: e() in
  def k(): h(a) + 1 in
  k()
```

Call Graph



1. Anything (besides main) that is called needs to be lifted

2. Anything reachable in the call graph from a lifted function needs to be lifted

Lambda Lifting: Who to Lift

1. Anything (besides main) that is called needs to be lifted

2. Anything reachable in the call graph from a lifted function needs to be lifted

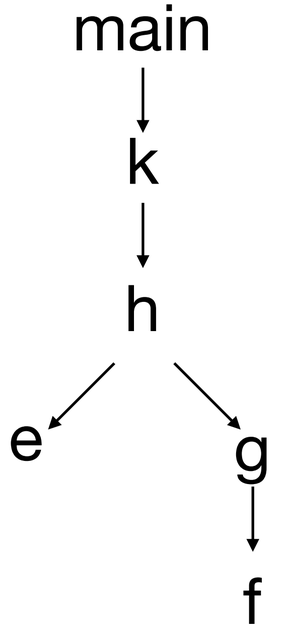
Implement by worklist algorithm:

1. Build a call graph

2. Initialize worklist with the functions that are non-tail called

3. While the worklist is nonempty: pop a function off, add the function to the set of functions that need to be lifted, add sucessors that are not already identified as lifted to the worklist

Call Graph



Lambda Lifting: What Args to Add

When we lift a function, we need to add extra arguments. But **which** arguments need to be added?

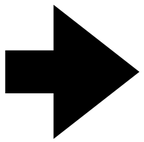
Answer: all the non-local variables that **must** be in the function's stack frame.

Which variables **must** be in the stack frame?
  Easy over-approximation: all the variables that are in scope at the function's definition site.

  Tempting but incorrect: just the variables that actually occur in the body of the lifted function?

Lambda Lifting: What Args to Add

```
def main(a):
  def e(): a * 2 in
  def f(): a in
  def g(): f() in
  def h(b): if b: g() else: e() in
  def k(): h(a) + 1 in
  k()
```



```
block e_tail(a):
  r = a * 2
  ret r
block f_tail(a):
  ret a
block g_tail(a):
  br f_tail(a)
block h_tail(a,b):
  cbr b g_tail(a) e_tail(a)
fun h_fun(a,b):
  br h_tail(a,b)
fun entry(a):
  k():
    tmp1 = call h_fun(a,a)
    tmp2 = tmp1 + 1
    ret tmp2
  br k()
```

Notice: need to add **a** to **g** even though it doesn't occur syntactically in the body of **g**

Lambda Lifting: What Args to Add

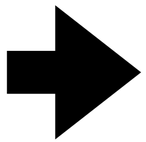
Adding all variables that are in scope is correct, but inefficient. Why?

Lambda Lifting: What Args to Add

The easiest way to correctly implement lambda lifting is to add **all** variables that are in scope as extra arguments.

But this can be inefficient:

```
def main(x):
  let a1 = ... in
  ...
  let a100 = ... in
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```



```
fun pow_tail(a1,...a100,x,n):
  ...
fun entry(x):
  ...
```

every unnecessary variable is extra space in our stack frames!

Lambda Lifting: What Args to Add

When lambda lifting, we need to add captured variables to lifted functions.

Capturing all in-scope variables is correct, but can be wasteful.

Two options:

- Perform a **liveness analysis** before lambda lifting, to determine exactly which variables are used.

- Wastefully add all variables, and perform liveness analysis afterwards, at the SSA level, and perform **parameter dropping**.

We'll adopt the second approach. We'll discuss how to implement **liveness analysis** later when we cover optimizations.

Diamondback

Static vs Dynamic Typing

How would we implement a language where integers and booleans were considered disjoint?

1. Static Typing (C/C++, Java, Rust, OCaml)

   Identify the runtime types of all variables in the program

   Reject type-based misuse of values in the frontend of the compiler.

2. Dynamic Typing (JavaScript, Python, Ruby, Scheme)

   Use **type tags** to identify the type of data at runtime

   Reject type-based misuse of values at runtime, right before the operation is performed

Static Typing vs Dynamic Typing

Example 1:

```
true + 5
```

Static typing: **compile time error**: true used where integer expected

Dynamic typing: **runtime error**: addition operation expects inputs to be integers

Static Typing vs Dynamic Typing

Example 2:

```
def main(x):
  x + 5
```

Static typing: need to declare a type for **x**, in this case **int**

Dynamic typing: succeed at runtime if **x** is an int, otherwise fail

Static Typing vs Dynamic Typing

Example 2:

```
def main(a):
  def complex_function(): ... in
  let x = if complex_function(): 1 else: true
  x + 5
```

Static typing: reject this program, even if **complex_function** always returns true

Dynamic typing: succeed at runtime if **complex_function** returns true, otherwise fail

Semantics of Dynamic Typing

Live code interpreter

Semantics of Dynamic Typing

- A Snake value is not just an int anymore. It is **either** an int or a boolean, and we need to be able to tell the difference at runtime in order to determine when we should error and how to implement isInt, isBool.
- Many operations can now produce runtime errors if type tags are incorrect, need to specify
  - what the appropriate error messages are
  - evaluation order between expressions executing and type tags
    - true + (let _ = print(3) in 3)
      - does this print 3 before it errrors?

Representing Dynamically Typed Values

In Adder/Boa/Cobra, all runtime values were integers.

In Diamondback, a runtime value must have both a type tag and a value that matches the type tag

How should we represent tags and values in our compiled program?

Representing Dynamically Typed Values

Approach 3: compromise

A snake value is a 64-bit value.

- Use the least significant bits of the value as a **tag**.
- Represent simple data like integers, booleans within the 64-bits
- Represent large datatypes like arrays, closures, structs as pointers to the heap

Upside: use stack allocation more often

Downside: can't fit 64 bits and a tag...

Roughly the approach used in high-performance Javascript engines (v8) as well as some garbage-collected typed languages (OCaml)

Representing Dynamically Typed Values

To implement our compiler, we need to specify

1. How each of our Snake values are represented at runtime
2. How to implement the primitive operations on these representations

Integers

Implement a snake integer as a 63-bit signed integer followed by a 0 bit to indicate that the value is an integer

Number	Representation
1	`0b00000000_0000.....0000_00000010`
6	`0b00000000_0000.....0000_00001100`
-1	`0b11111111_1111.....1111_11111110`

I.e., represent a 63-bit integer **n** as the 64-bit integer 2 * n

Booleans

The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b01` to distinuish from integers and other datatypes

Use the remaining 62 bits to encode true and false as before as 1 and 0

Number	Representation
`true`	`0b00000000_0000.....0000_00000101`
`false`	`0b00000000_0000.....0000_00000001`

2^62 - 2 bit patterns are therefore "junk" in this format

Boxed Data

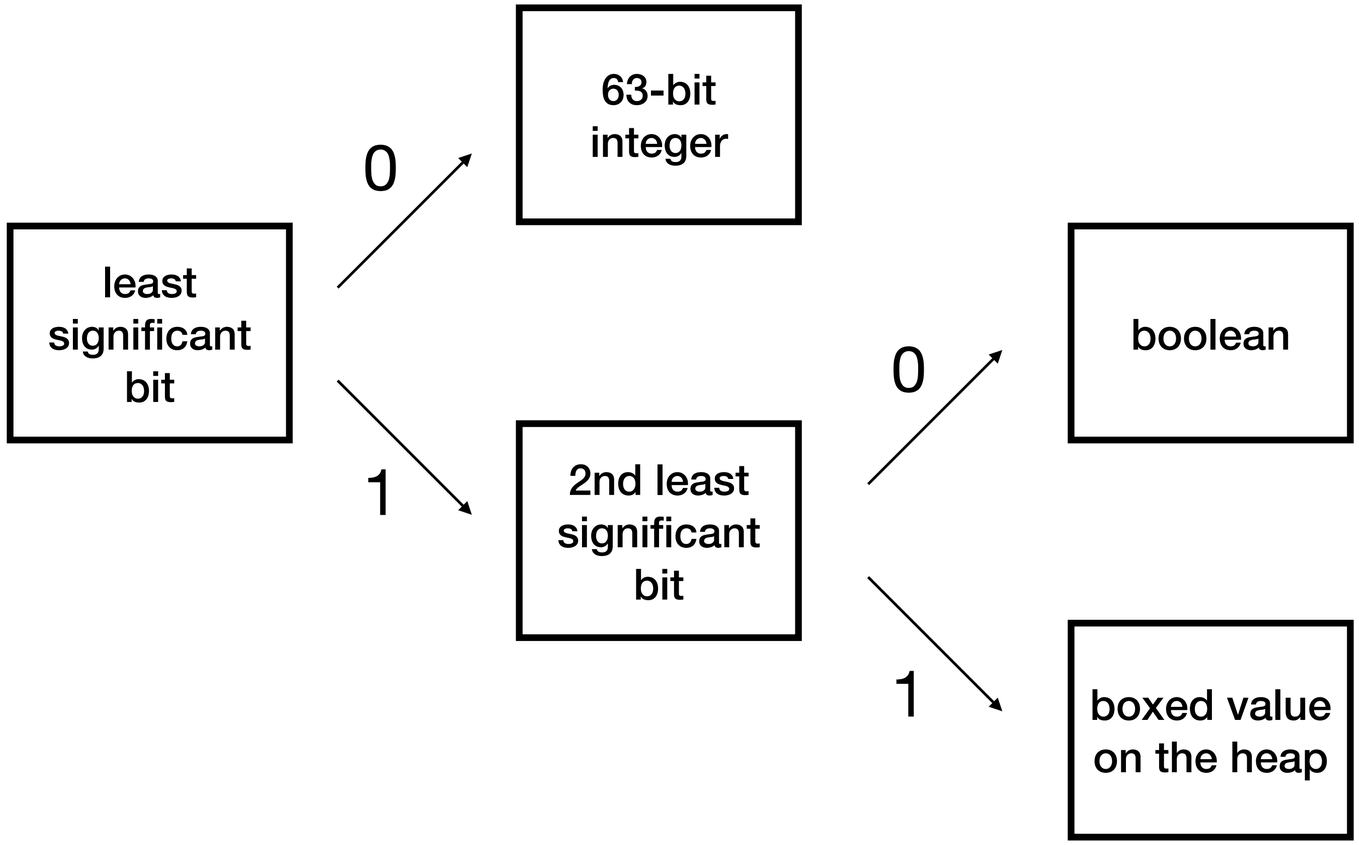
The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b11` to distinguish from booleans.

Use remaining 62-bits to encode a pointer to the data on the heap

Why is this ok? Discuss more thoroughly on Wednesday

Representing Dynamically Typed Values



Compiling Dynamic Typing

We know what the source semantics is and what kind of assembly code we want to generate.

In implementing the compiler, we now we have a design choice: in what phase of the compiler do we actually "implement" dynamic typing?

1. Implement everything in x86 code generation
2. Implement everything in lowering to SSA
3. Implement in multiple passes

Compiling Dynamic Typing

Approach 3: implement dynamic typing in multiple passes

In lowering to SSA, make some aspects of dynamic typing explicit but leave the tag checking as primitive operations.

Implement the tag checking in the x86 code generation.

Compiling Dynamic Typing

Approach 3: implement **some** dynamic typing in SSA lowering

Diamondback

```
x * y
```

SSA

```
assertInt(x)
assertInt(y)
half = x >> 1
r = half * y
ret r
```

Insert type tag assertions in SSA, implement bit-twiddling manually

Compiling Dynamic Typing

Approach 3: implement **some** dynamic typing in SSA lowering

SSA

```
assertInt(x)
```

x86

```
mov rax, [rsp - offset(x)]
test rax, 1
jnz assert_int_fail
...
assert_int_fail:
  sub rsp, 8
  call snake_assert_int_error
```

Compiling Dynamic Typing

Optimization opportunity

Diamondback

```
def fact(x):
  if x == 0:
    1
  else:
    x * fact(x - 1)
in
fact(7)
```

SSA

```
...
tmp1 = x - 1
tmp2 = call fact(tmp1)
assertInt(x)
assertInt(tmp2)
r = x * tmp2
ret r
```

will these **assertInt** ever fail?

Compiling Dynamic Typing

Optimization opportunity

Diamondback

```
def fact(x):
  if x == 0:
    1
  else:
    x * fact(x - 1)
in
fact(7)
```

SSA

```
...
tmp1 = x - 1
tmp2 = call fact(tmp1)
~~assertInt(x)~~
~~assertInt(tmp2)~~
r = x * tmp2
ret r
```

with a simple **static analysis** determine that x, tmp2 always have the correct tag for an Int. Remove unnecessary assertions

Compiling Dynamic Typing

Compare to approach 2:

Diamondback

```
x * y
```

how would we remove the checking from the code on the right?

SSA

```
check_y():
  y_bit = y & 1
  c = y_bit == 0
  cbr mult_xy() err()
mult_xy():
  tmp = x * y
  z = tmp >> 1
  ret z
x_bit = x & 1
b = x_bit == 0
cbr check_y() err()
...
```

State of the Snake Language

Adder: Straightline Code (arithmetic circuits)

Boa: local control flow (finite automata)

Cobra: procedures, extern (pushdown automata)

Snake v4: **Diamondback**

1. Add new datatypes, use dynamic typing to distinguish them at runtime
2. **Include heap-allocated variable-sized arrays, allowing for unrestricted memory usage**

Computational power: Turing complete

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  [x , x + 1, x + 2]
```

allocate an array with a statically known size

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  newArray(x)
```

allocate an array with dynamically determined size (elements initialized to 0)

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
   a[0]
```

array indexing

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
   let _ = a[1] := a[1] + 1 in
   a[1]
```

arrays can be mutably updated

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
  length(a)
```

should be able to access the length of any array value

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
   a[3]
```

Out of bounds access/update should be runtime errors

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
   isArray(a)
```

support tag checking as with ints, bools

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let list = [0, 1, false] in
  let _ = list[2] := list in
   ...
```

mutable updates allow for cyclic data

Concrete Syntax

⟨*expr*⟩: ...

| ⟨array⟩
| ⟨expr⟩ `[` ⟨expr⟩ `]`
| ⟨expr⟩ `[` ⟨expr⟩ `]` `:=` ⟨expr⟩
| `newArray` `(` ⟨expr⟩ `)`
| `isBool` `(` ⟨expr⟩ `)`
| `isInt` `(` ⟨expr⟩ `)`
| `isArray` `(` ⟨expr⟩ `)`
| `length` `(` ⟨expr⟩ `)`

⟨*exprs*⟩:

| ⟨expr⟩
| ⟨expr⟩ `,` ⟨exprs⟩

⟨*array*⟩:

| `[` `]`
| `[` ⟨exprs⟩ `]`

Abstract Syntax

```
enum Prim {
  ...
  // Unary
  IsArray,
  IsBool,
  IsInt,
  NewArray,
  Length,

  MakeArray, // 0 or more arguments
  ArrayGet, // first arg is array, second is index
  ArraySet, // first arg is array, second is index, third is new value
}
```

Extending the Snake Language

Diamondback: Arrays

Semantics:

1. Each time we allocate an array should be a new memory location, so that updates don't overwrite previous allocations

2. What value does `e1[e2] := e3` produce?
   options: a constant, the value of e1 or e3, the old value of `e1[e2]`

3. Is equality of arrays by value or by reference?

   [0, 1, 2] == [0 , 1, 2]

Allocating Arrays

Where should the contents of our arrays be stored?

- Stack?

- Heap?

Stack Allocation

Dynamically sized data can only feasibly be stack allocated if it is **local** to the function, i.e., only used in call stacks that contain the current function's stack frame.

If the dynamically sized data is **returned** from the function that allocates it, we instead allocate it in a separate memory region, the **heap**, and return a pointer to it.

Heap Allocation

The heap contains data whose lifetime is not tied to a local stack frame.

This makes the usage of the data more flexible, but complicates the question of when the data is **deallocated**.

For today, let's assume we do not deallocate memory.
  A strategy used in some specialized applications (missiles)

Today's simple heap model: the heap is a large region of memory, disjoint from the stack, some of it is used, and we have a pointer to the next available portion of memory.

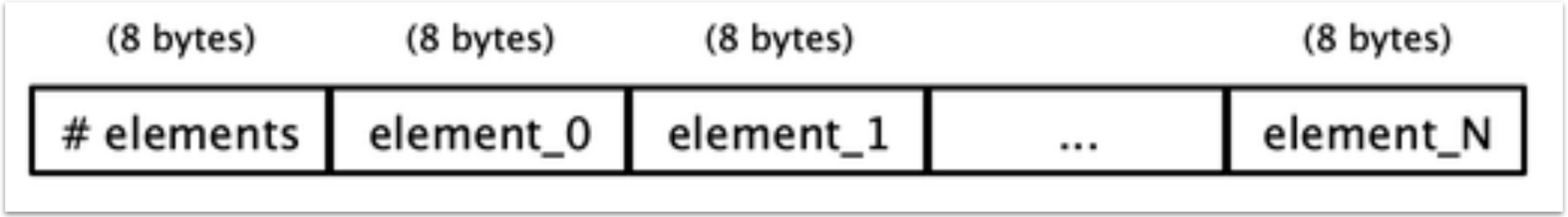
Implementing Arrays

When we implement arrays, we have two different representations to define:
1. How they are stored as "objects" in the heap
2. How they are represented as Snake values

Arrays as Objects

What data does an array need to store?

1. Need to layout the values sequentially so we can implement get/set
2. Need to store the **length** of the array to implement length as well as bounds checking for get/set.



Arrays as Values

The Snake value we store on the stack for an array is a **tagged** pointer to the array stored on the heap.

We overwrite the 2 least significant bits of the pointer with the tag 0b11.

This is safe, as long as those 2 least significant bits of the pointer contain no information, i.e., if they are always 0.

2 least significant bits of a pointer are 0 means the address is a multiple of 4, meaning the address is at a 4-byte alignment.

All arrays on our heap take up size that is a multiple of 8 bytes, so as long as the base of the heap is 4-byte aligned, we maintain this invariant.

Functions as Values

So far in our Snake language, functions are **second class,** meaning that unlike integers/booleans/arrays:

- ordinary program variables cannot be functions

- functions can't be passed as arguments to other functions

- functions can't be returned as values from other functions

This restriction simplifies the job of the compiler, but is uncommon in modern programming languages.

Functions as Values

Modern programming languages allow us to use functions as first-class data

- Low level languages like C/C++ have **function pointers**, which can be passed and returned like any other pointer type

- Higher-level languages both statically (C++, Rust, Java, Go, OCaml, Haskell) and dynamically typed (Python, Ruby, JavaScript, Racket) allow for a more flexible type called **closures**, sometimes called **lambdas**

Used as a convenient interface for implementing iterators, callbacks, concurrency,...

Functions as Values

```
def applyToFive(f):
  f(5)
in

def incr(x):
  x + 1
in

applyToFive(incr)
```

Functions as Values

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
def incr(x): x + 1 in
def sqr(x): x * x in
let a = [0, 1, 2] in
map(incr, map(sqr, a))
```

Functions as Values

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
let a = [0, 1, 2, 3] in
def sqr_a(i): a[i] := a[i] * a[i] in
let _ map(sqr_a, [1,3]) = in
a
```

need to support variable capture

Lambda Notation

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
let a = [0, 1, 2] in
map(lambda x: x + 1 end,
    map(lambda x: x * x end,
    a))
```

Lambda Notation

Lambda notation is a syntax for defining function values directly rather than using `def`

```
lambda x1, x2,...: e end
```

Convenient for defining small functions to pass to map/filter/fold, etc.

Implementing First-Class Functions

How can we implement first class functions:

1. What is the runtime representation of a function value?

2. What data do we need to correctly implement **dynamic** type checking for functions

3. How can we ensure that we handle **variable capture** ?

Function Pointers

The compiled instructions implementing our functions are stored in executable memory.

The simplest way to implement functions as first class values is for the **runtime representation** of a function to simply be the **address of its code** in memory.

Function Pointers

The compiled instructions implementing our functions are stored in executable memory.

The simplest way to implement functions as first class values is for the **runtime representation** of a function to simply be the **address of its code** in memory.

This representation works in C, a **statically typed** language where functions are only defined at the **top level**, i.e., cannot capture any variables.

Our language is **dynamically typed** and has **local function definitions.**

Dynamic Typing for Function Values

- Need to tag function values so that we can distinguish them from other data.

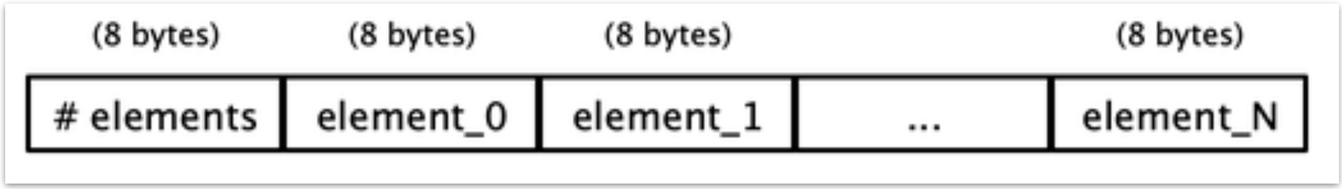
- Need to store the **arity**, i.e., number of parameters, with the function. Similar to storing array length.

Dynamically typed function pointer then needs to take up more than 8 bytes to store the function pointer and the arity, so we can store this as **boxed** data stored on the heap.

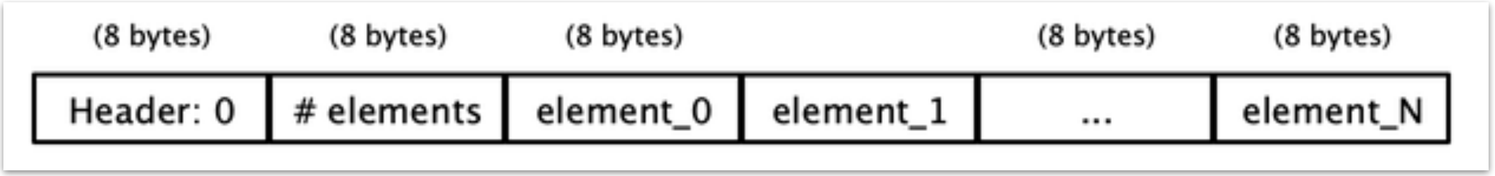
Heap Object Metadata

We have already used 2 bits in our Snake values for tagging datatypes. If values are **boxed** we can easily allocate many more by using a **header word** in our heap representation.

E.g., for arrays:



Update to include an initial 8 byte header that identifies the type of the object on the heap. For arrays: tag 0



Heap Object Metadata

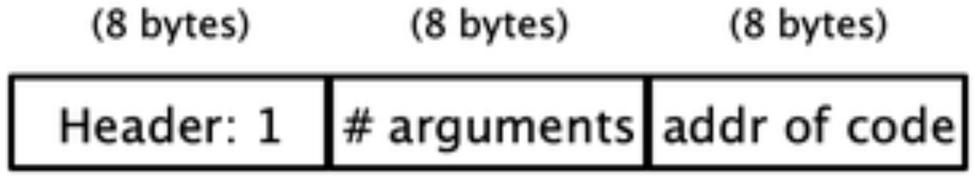
We have already used 2 bits in our Snake values for tagging datatypes. If values are **boxed** we can easily allocate many more by using a **header word** in our heap representation.

For function pointers, we need to store

- A different header tag

- the number of arguments

- the function pointer itself



Function Pointers

The compiled instructions implementing our functions are stored in executable memory.

The simplest way to implement functions as first class values is for the **runtime representation** of a function to simply be the **address of its code** in memory.

This representation works in C, a **statically typed** language where functions are only defined at the **top level**, i.e., cannot capture any variables.

Our language is **dynamically typed** and has **local function definitions.**

Variable Capture

```
let a = [0, 1, 2, 3] in
let _ map(lambda i: a[i] := a[i] * a[i] end, [1,3]) = in
a
```

the lambda function here **captures** the array variable **a**

Variable Capture

For second-class functions, we used lambda lifting to implement variable capture.

- Key property of second-class functions: at a call site, we can statically determine the function we are being called with. So we can lookup what extra arguments the function requires

- This property fails for first-class functions

Functions Pointers can't Capture

Just like second class functions, first-class functions can **capture** variables in scope at their definition site.

Unlike second class functions, the caller of a first-class function

1. Doesn't know how many variables the function captured

2. May not even have access to the variables the function captured

Our current strategy of supplying captured variables as extra arguments at the **call site** is doomed to fail for first-class functions that can capture.

Alternative: need to supply the captured variables at the **definition site.**

Local Functions Attempt 2: Closures

The most common strategy to implement local functions is to use **closures**.

- A function value at runtime is a heap-allocated object grouping a function pointer with an array containing the values of its captured variables

- Constructing a function value involves storing the captured variables on the heap

- To call a function, unpack its code pointer and pass a pointer to its captured variables.

Closure Conversion

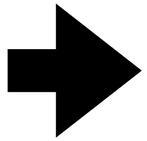
Similar to our lambda lifting pass, we can translate programs that implicitly use closures and capture variables to one that explicitly constructs/deconstructs them. This pass is called **closure conversion.**

Can be done at the AST level or the SSA level, just like lambda lifting

Closure Conversion

An easy way to implement closure conversion is to compile to a version of the language with arrays + function pointers.

```
def adder(x):
  lambda y: x + y end
in
let add1 = adder(1),
    add2 = adder(2),
in add1(add2(0))
```



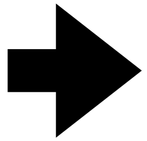
```
def lambda_fun(captures,y):
  captures[0] + y
in
def adder(x): [lambda_fun, [x]] in

let add1 = adder(1),
    add2 = adder(2),
in add1[0](add1[1],
     add2[0](add2[1], 0))
```

By storing the captured variables as part of the function value, we can supply them at the definition site, and use them at the call site

Closure Conversion

An easy way to implement closure conversion is to compile to a version of the language with arrays + function pointers.

```
def main(b):
  let f =
    if b:
      let x = 7 in
      lambda y:
        x + y end
    else: lambda y: y end
  in
  f(5) + 1
```



```
def lambda_1(captures, y):
  captures[0] + y
in
def lambda_2(captures, y): y in
def main(b):
  let f =
    if b: let x = 7 in [lambda_1, [x]]
    else: [lambda_2, []]
  in
  f[0](f[1], 5) + 1
```

Important to represent all lambda functions as taking an array of arguments so that they have a uniform interface: the caller doesn't know how large the captured environment is

Closure Conversion

Translating closures to arrays + function pointers gives correct semantics, but doesn't support dynamic typing features. E.g., all closures values would satisfy **isArray**.

Instead make a new type of heap object for closures

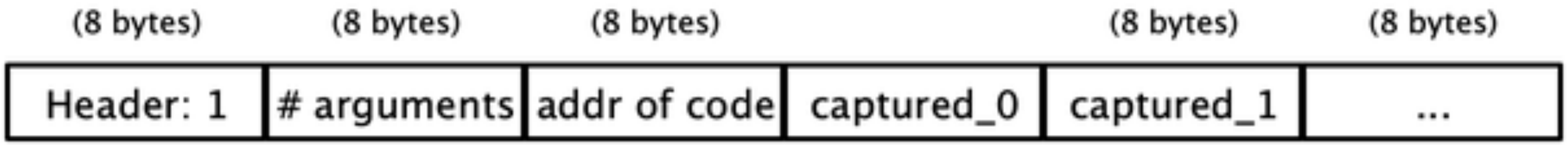
Functions as Closures

A **closure** is a datatype for first-class functions consisting of both

1. The function pointer

2. An array of **captured arguments**

We store the captured arguments at the function's **definition site**, rather than passing them at the **call site**



Compiling Functions

In our source programming languages, functions are a simple, elegant abstraction.

But they do not have a single elegant implementation.

Modern compilers work hard to combine multiple implementation strategies behind this single source interface:

1. Local tail calls can be compiled as efficiently as loops

2. Most calls are to statically determined functions, don't require allocating a closure

3. Construct a closure only when necessary: when the function is actually used in a first class manner.

Essential in performance-sensitive languages like Rust that use closures for core functionality (iterators)!