EECS 483: Compiler Construction

Lecture 7:
Loops, Mutable Variables

February 4, 2025

Learning Objectives

How do we compile **mutable** variables to SSA while satisfying SSA's "static single assignment" property.

How do we compile **loops** to SSA blocks

Why do compilers go through all the trouble to compile **imperative programs** to a **functional IR** when the target is an **imparative assembly language**?

If time permits: non-tail function calls.

Previously on EECS 483...

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

1. Functional: recursive functions, tail calls

2. Imperative: loops, mutable variables

If we can only ever perform tail calls, functions are almost exactly the same as SSA parameterized blocks.

Need post-processing in the form of block sinking and parameter dropping to get **minimal** SSA form (minimal block parameters)

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: loops, mutable variables

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

Imperative Snake Language

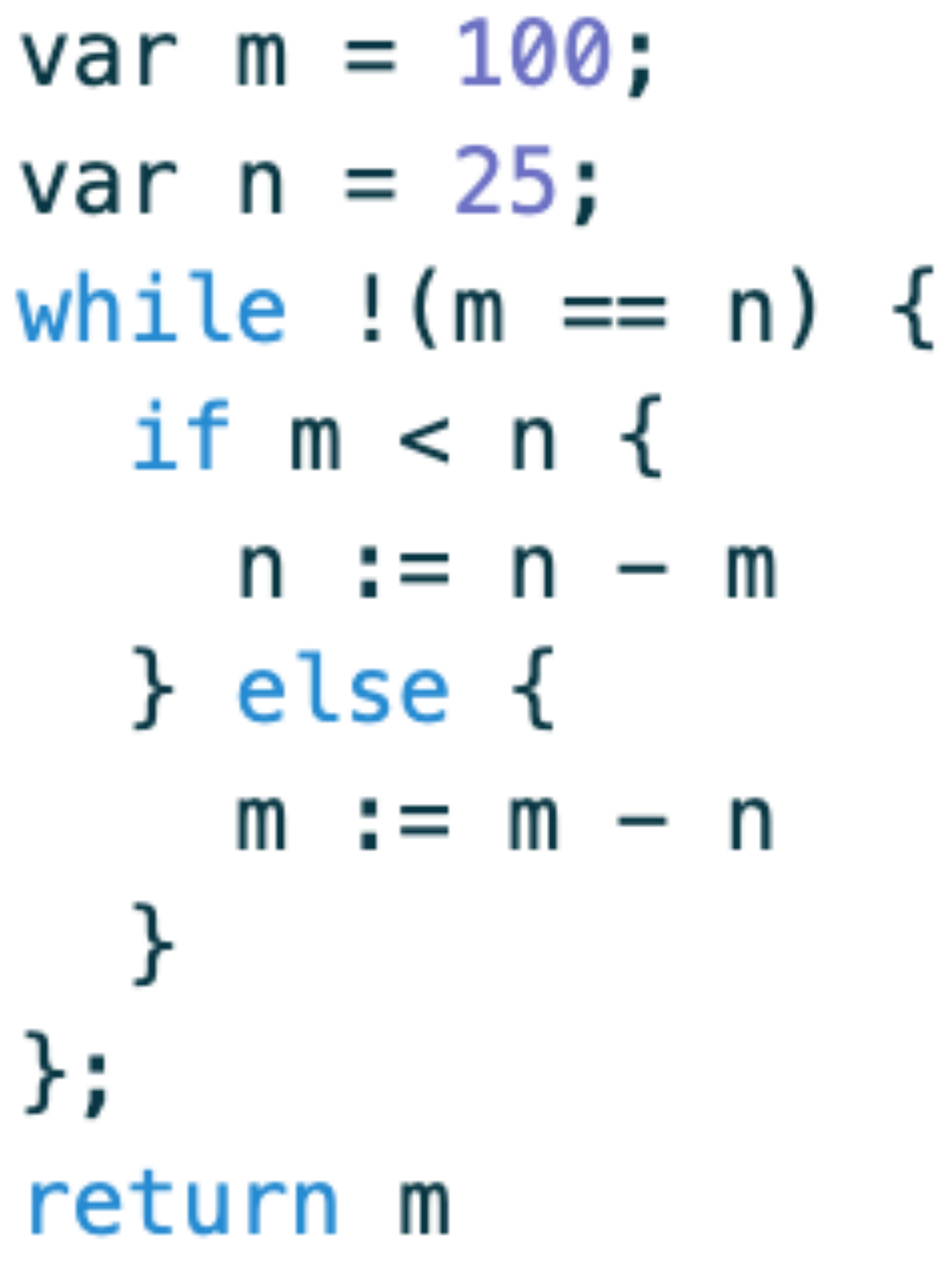
Imperative Snake Language "Imp"

- Mutable variables

- statement-expression distinction

- while loops

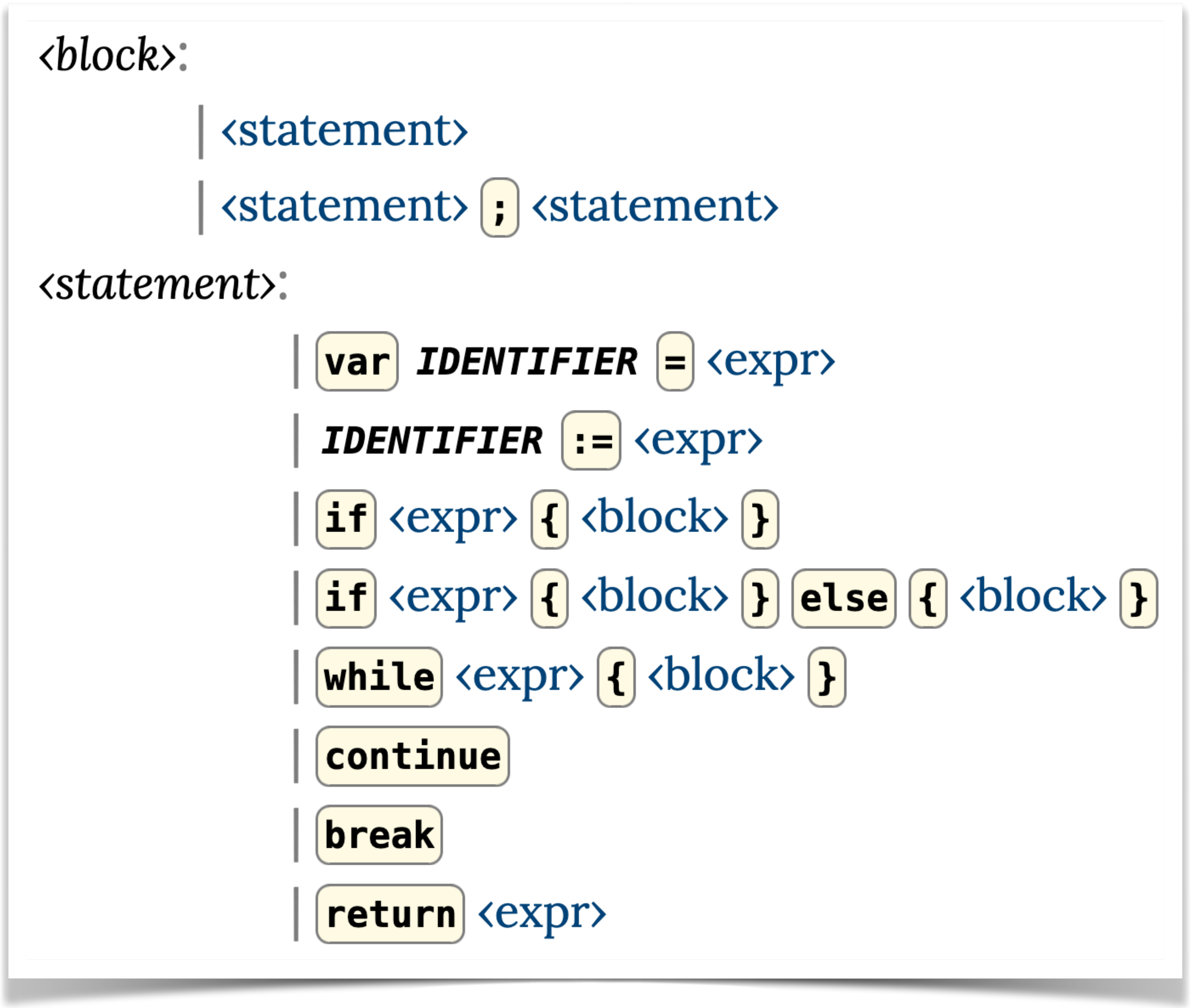
- return/break/continue



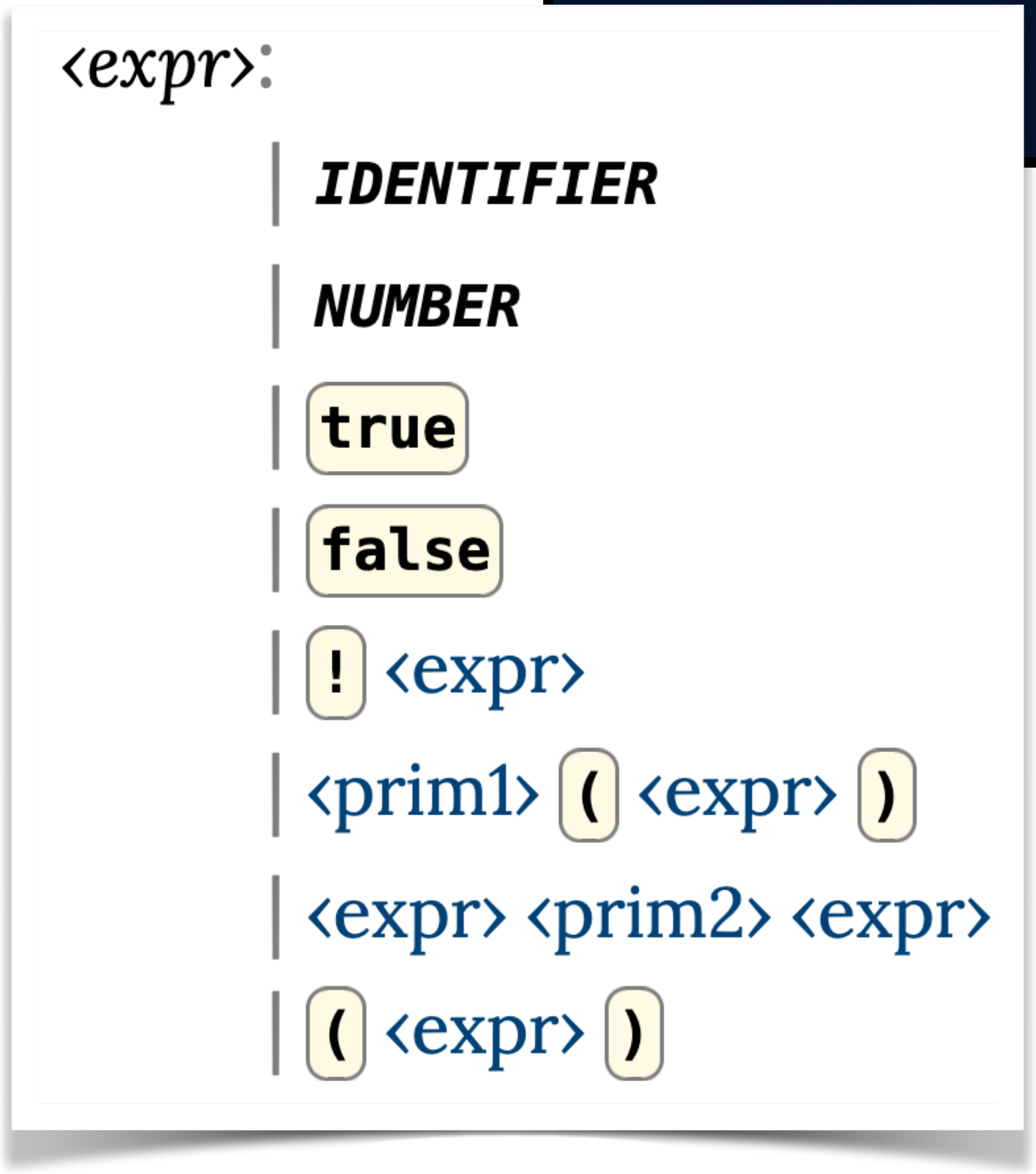
```
var m = 100;
var n = 25;
while !(m == n) {
  if m < n {
    n := n - m
  } else {
    m := m - n
  }
};
return m
```

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 Snake Language

well-formedness

Still have a notion of scope, shadowing:

1. Check variables are declared before use

2. Shadowing is allowed, semantically shadowed var is a **different** mutable variable

Translate away shadowing to unique variable names to avoid headaches, as usual

Imperative Snake Language

well-formedness

```
var x = y + z;
return x
```

undeclared var y, z

similar to existing scope checker

Imperative Snake Language

well-formedness

If implementing a procedure that returns a value, need to ensure that every code path ends in a return

```
...
if b {
    ...
    return x;
} else {
  x := 5
}
```

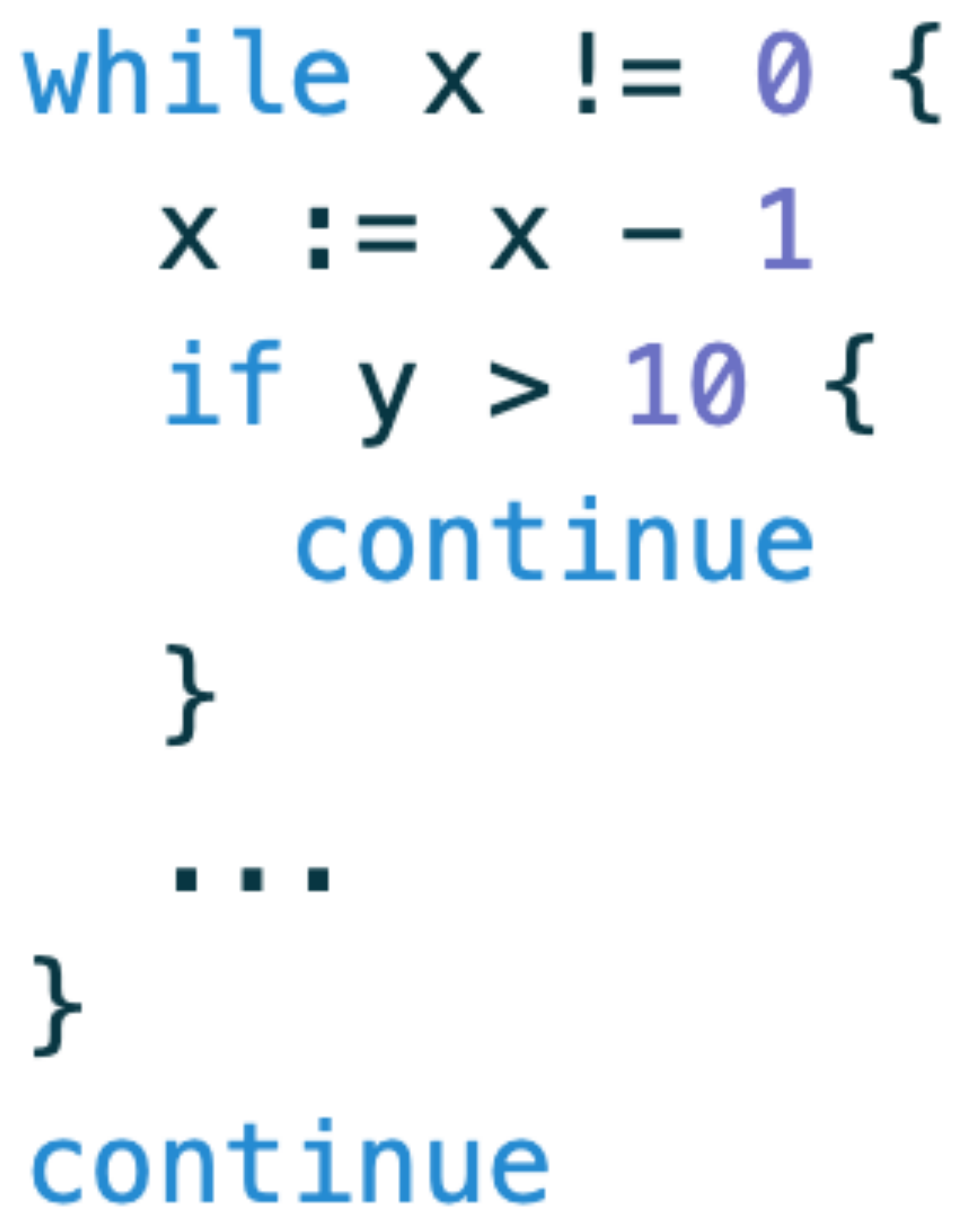
Imperative Snake Language

well-formedness

Naked break/continue:

Verify that break/continue operations only occur inside of an enclosing while loop

alternative: use named break/continue



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

Imperative Snake Language

semantics

Each variable acts like a 64-bit "register"

When evaluating, need to keep track of the current state of all the variables

Imperative Snake Language

semantics

```
var x = 10;
...
if x != y {
   var x = 14;
   ...
   x := x + 1
}
return x;
```

shadowed variables should not be overwritten. Making variable names unique makes this easier to get right

Imperative Snake Language

semantics

while loop:

  check the condition expression

    true: execute the block and repeat

    false: execute the next statement

break:

  in a while loop, goto the next statement after the loop

continue:

  in a loop, goto the beginning of the loop

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

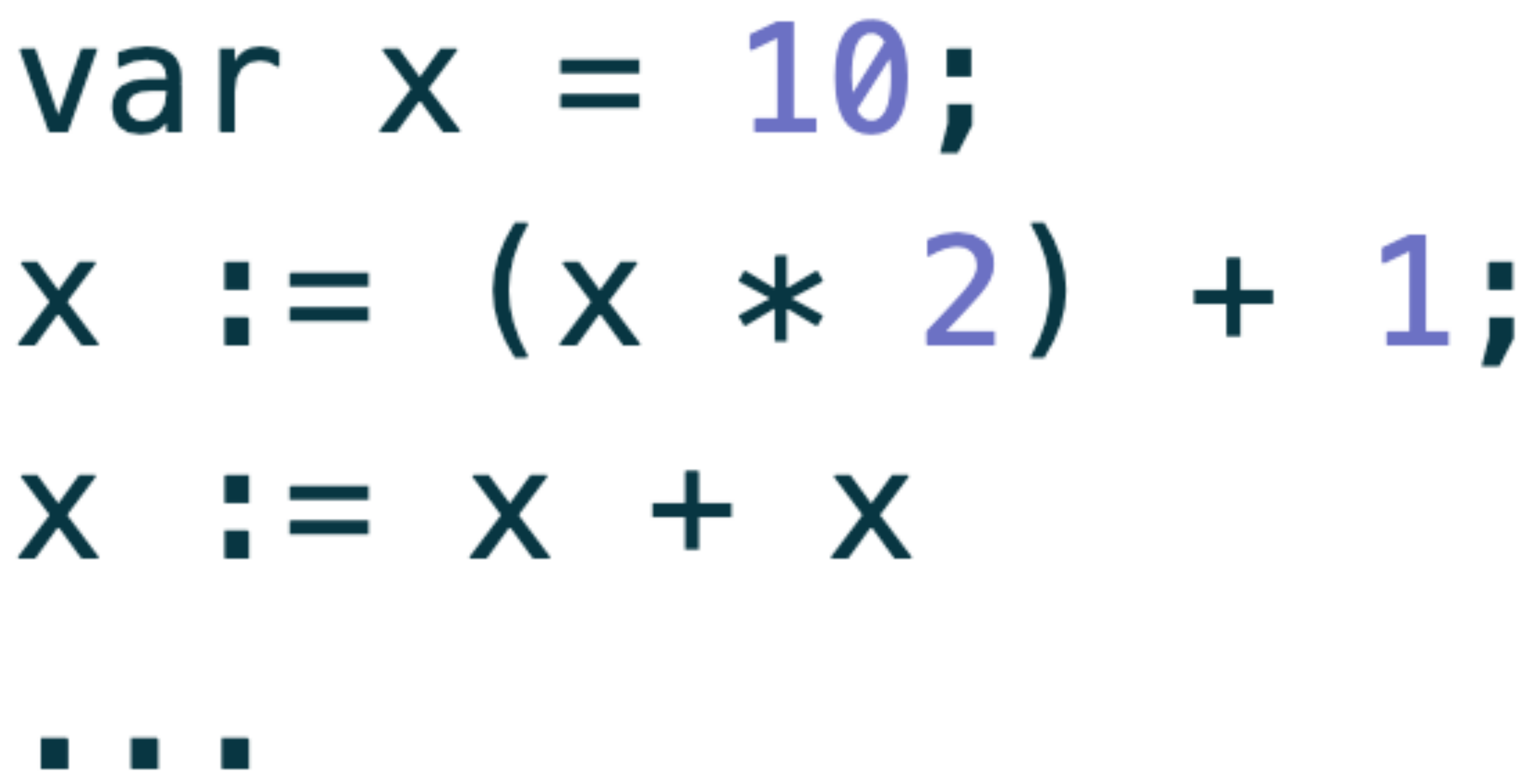
Variable declarations are implemented just as with Let: a var declaration in Imp becomes a variable assignment in SSA

```
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 3: Conditionals

Imperative to SSA

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

$\Rightarrow$

```
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 3: Conditionals

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 to 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
```

$$\begin{tikzcd} & {} \arrow[d] & \\ & \begin{matrix}\text{loop}\\ \text{check e}\end{matrix} \arrow[dl, shift right] \arrow[dr] & \\ \text{b1} \arrow[ur, shift right] & & \text{b2} \end{tikzcd}$$

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(...)
```

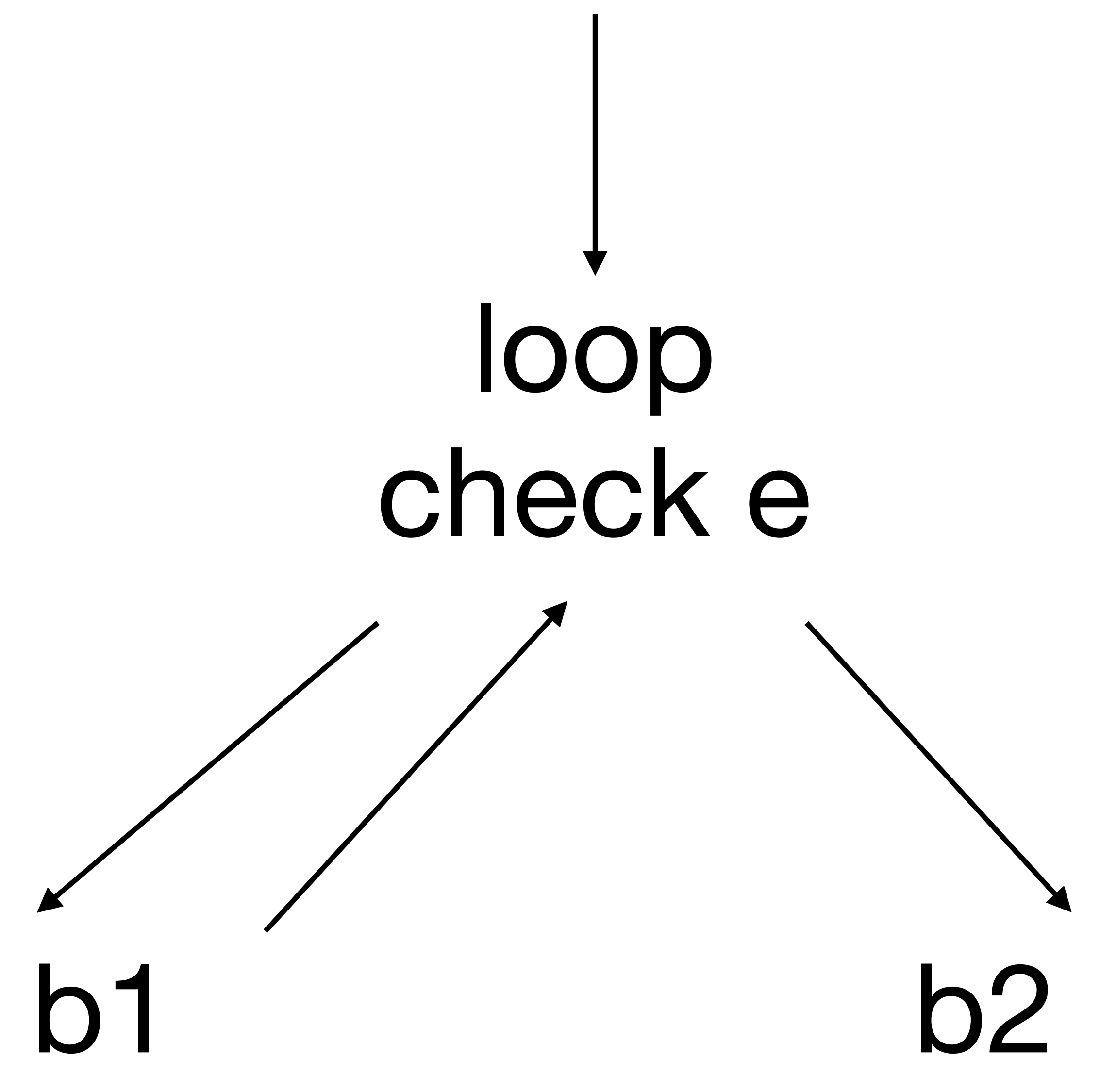
Imperative to SSA

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

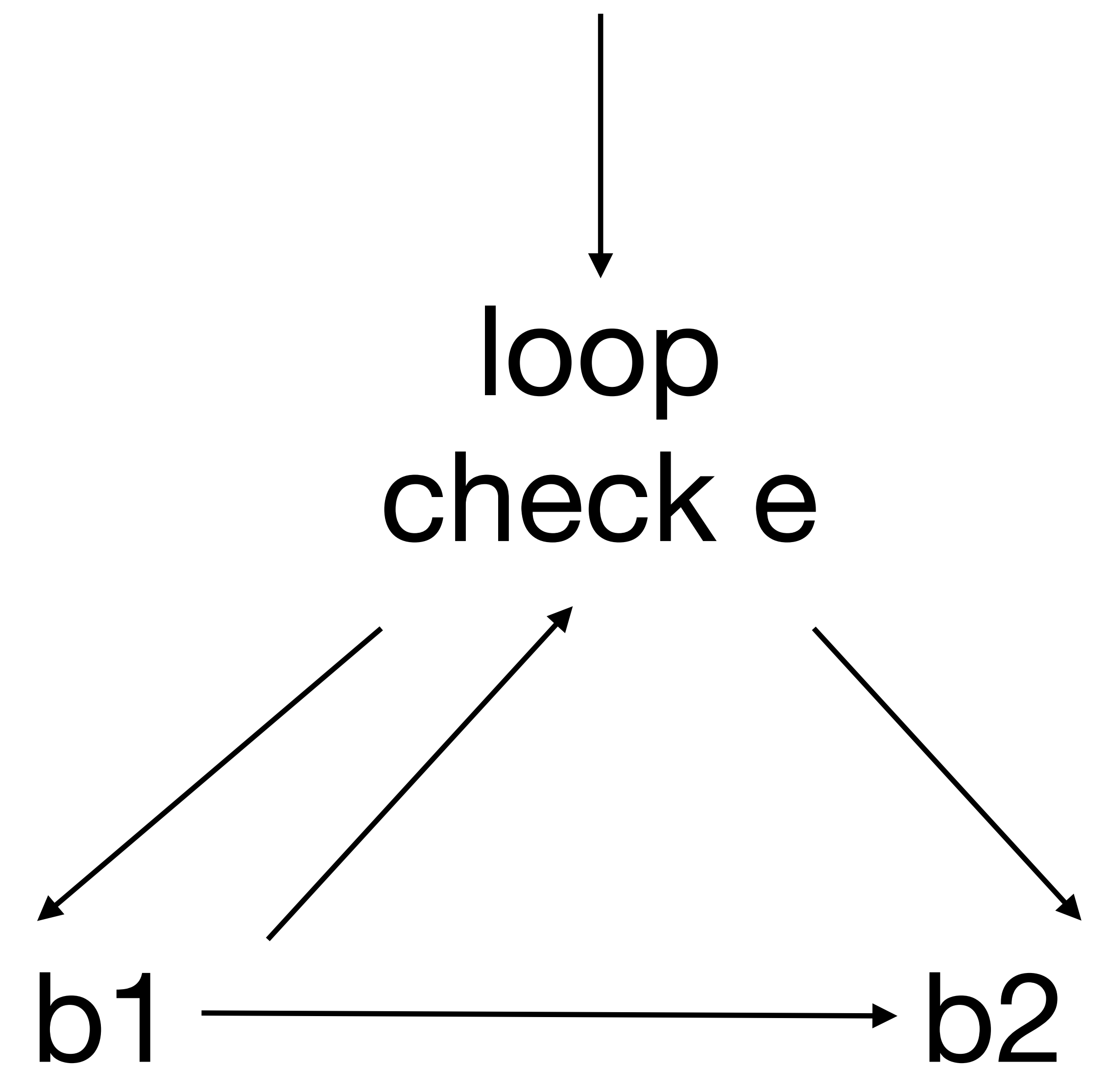
```
while e {
  b1
}
b2
```



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
```


$$\begin{tikzcd}
 & \arrow[d] & \\
 & \text{loop check } e \arrow[dl] \arrow[dr] & \\
b1 \arrow[ur] \arrow[rr] & & b2
\end{tikzcd}$$

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)
```

Imperative and Functional compile to same 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)
```

```
def loop(m, n):
  if m == n:
    m
  else: if m < n:
    loop (m , n - m)
  else:
    loop(m - n, m)
in
loop(100 , 25)
```

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 Form

Translating Imperative code to SSA using crude phi node insertion produces **very** non-minimal SSA: many extra block parameters

But because imperative code is well-structured, block sinking is not necessary, blocks are already nested inside their immediate dominators

Only need to implement parameter dropping.

Theorem: crude **phi** node insertion + parameter dropping produces minimal SSA

Imperative Parameter Dropping Example

```
...
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()
```

Why all the trouble?

Modern compiler infrastructure for imperative languages:

- input program: mutates variables directly, variables similar semantics to registers
- middle end: translates into SSA, functional intermediate representation where variables are never mutated
- backend: translate out of SSA, map variables to registers (or memory), mutate their values

SSA Benefits

Common Subexpression Elimination

Is this a valid optimization?

```
x = 1
y = x + 1
...
z = x + 1
...
```

```
x = 1
y = x + 1
...
z = y
...
```

in SSA, yes!

in imperative programming languages, no.

SSA Benefits

Common Subexpression Elimination

Is this a valid optimization?

```
x = 1
y = x + 1
x *= 5
z = x + 1
...
```

```
x = 1
y = x + 1
x *= 5
z = y
...
```

To perform this optimization in imperative languages we need to perform an analysis to check if x has been updated. In SSA, the analysis is free!

SSA Benefits

Program analyses can be implemented more **efficiently**.

Can set up data structures that map variable uses directly to their definitions. Skips over a great deal of irrelevant information.

In an imperative program variables can be updated anywhere, putting the program in SSA form makes the dataflow information easier to access

$$
\begin{aligned}
&x_1 = 1;\\
&y = x_1 + 1;\\
&x_2 = 2;\\
&z = x_2 + 1;
\end{aligned}
$$

SSA Benefits

When program analysis is **easier**:

1. More efficient generated code: Easier for compiler writers to implement more and better analyses/ optimizations

2. More efficient compiler: accessibility of information in SSA form allows efficient data structures for program analysis, since more information is manifest in the program format

SSA History, Benefits

Further Reading: SSA Book Chapter 1

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

Extending the Snake Language

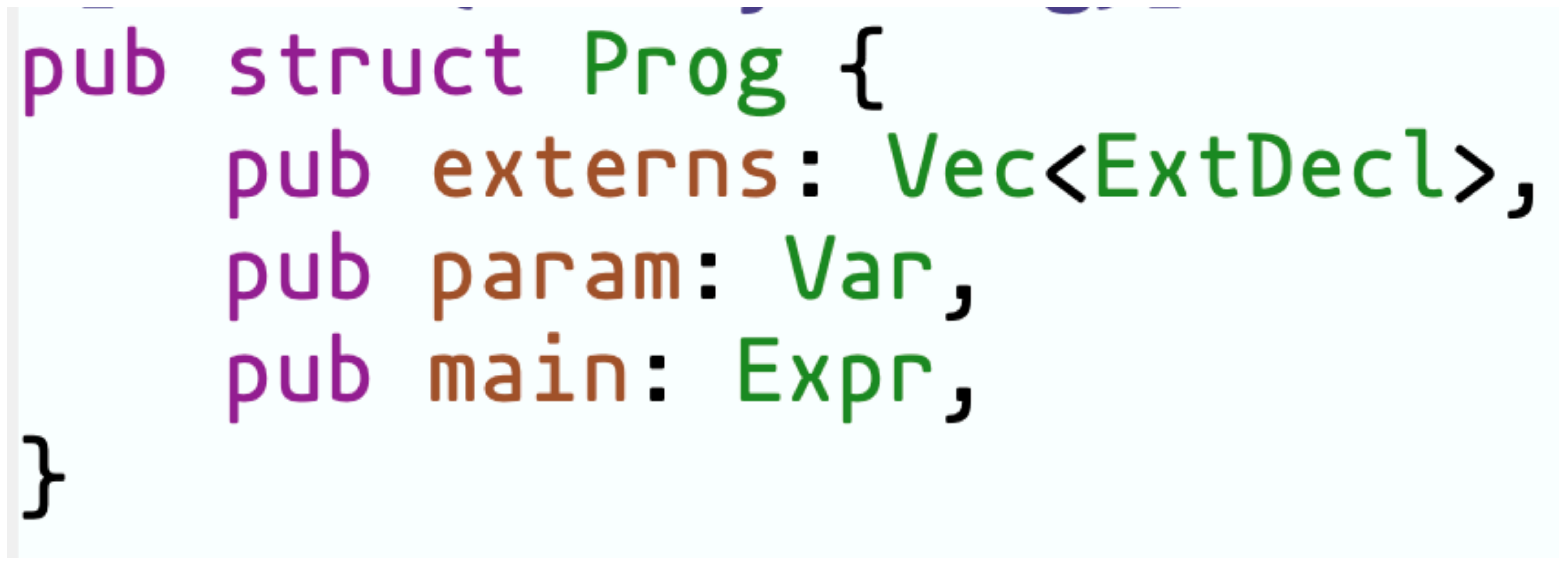
```
extern read()
extern print(x)

def main(x):
  def loop(sum):
    let _ = print(sum) in
    loop(sum + read())
  in
  loop(0)
```

Implement read, print in stub.rs

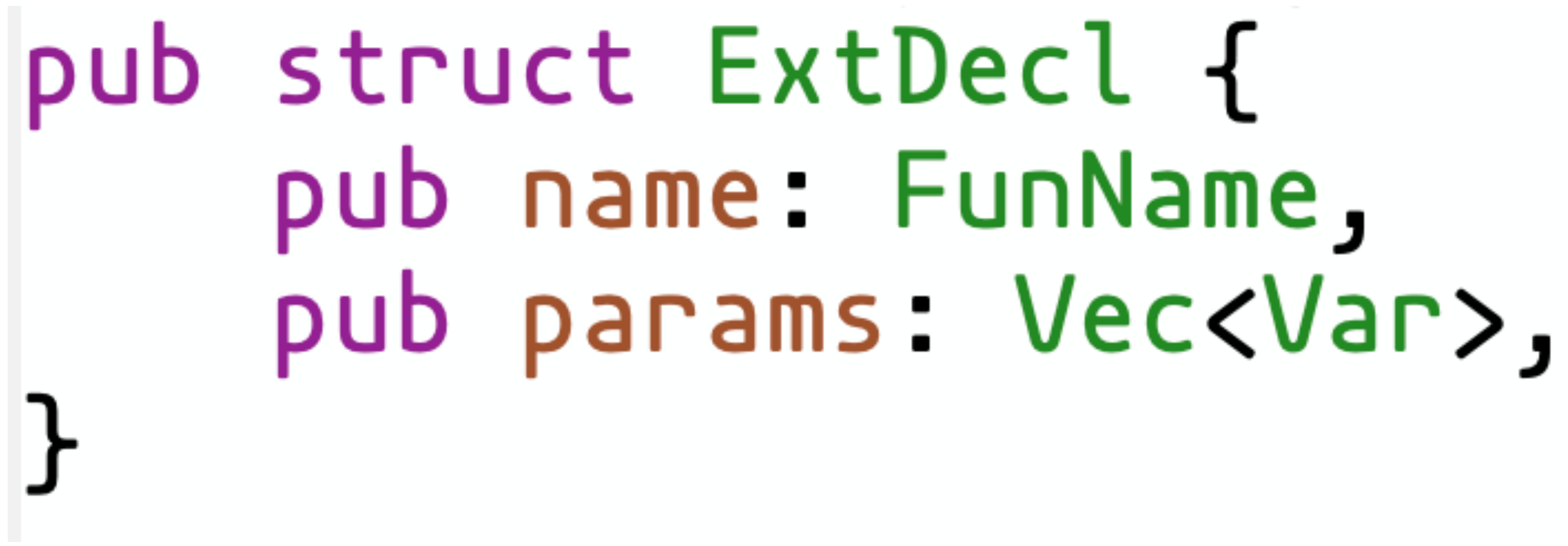
Concrete Syntax

```
‹prog›:
        | ‹externs› def main ( IDENTIFIER ) : ‹expr›
        | def main ( IDENTIFIER ) : ‹expr›
‹extern›:
          | extern IDENTIFIER ( )
          | extern IDENTIFIER ( ‹ids› )
‹externs›:
           | extern
           | extern ‹externs›
‹expr›: ...
‹ids›: IDENTIFIER | IDENTIFIER , ‹ids›
```

Abstract Syntax



```
pub struct Prog {
    pub externs: Vec<ExtDecl>,
    pub param: Var,
    pub main: Expr,
}
```



```
pub struct ExtDecl {
    pub name: FunName,
    pub params: Vec<Var>,
}
```

Well-formedness for Extern

1. Extern functions should be in the same scope as local function definitions. Local function definitions can shadow external function declarations

2. Can't have multiple external functions with the same name

2. In the name resolution phase, do **not** change the names of external functions, as the name is used for linking

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)
```

$\Rightarrow$

```
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

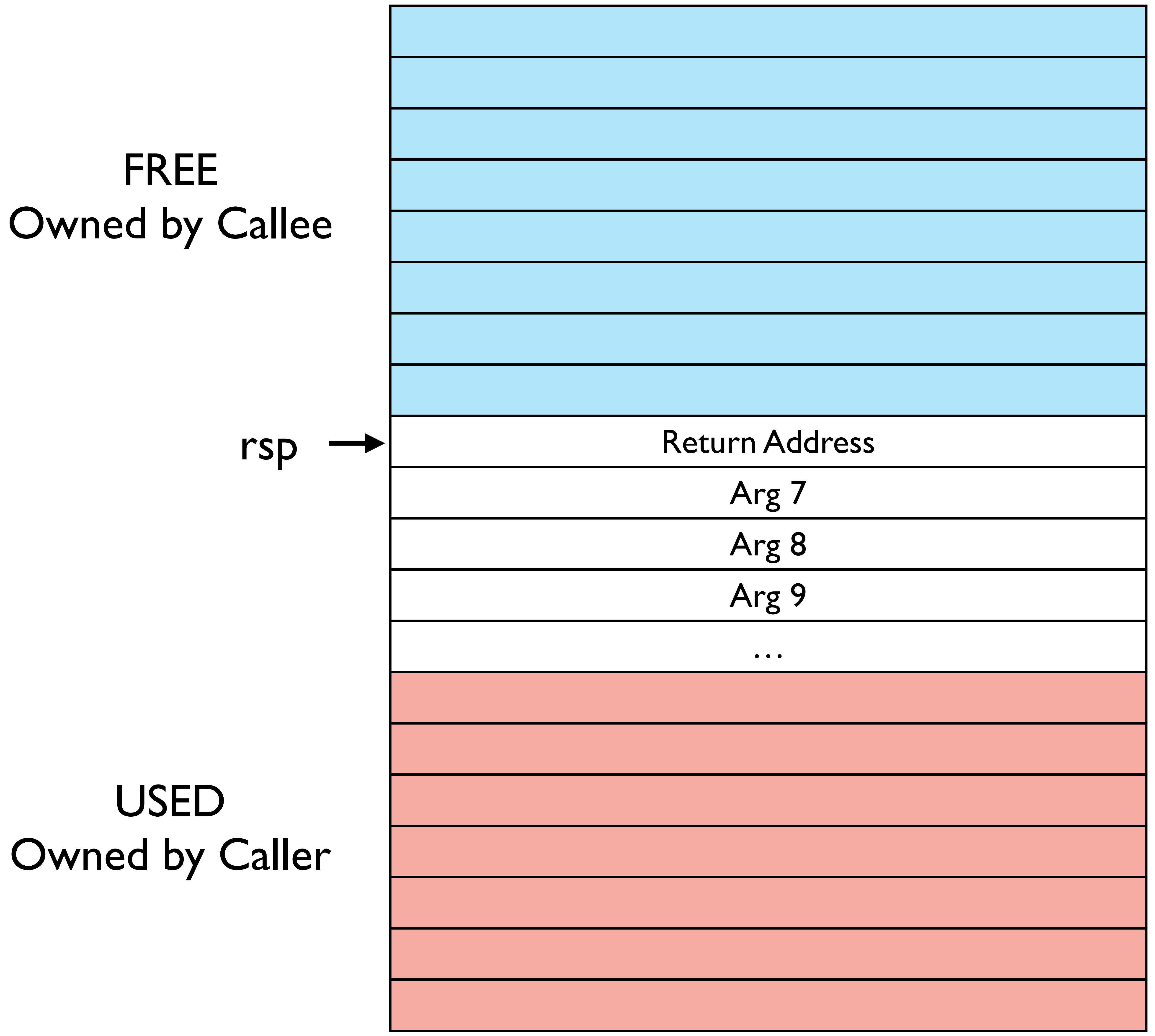
System V AMD 64

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

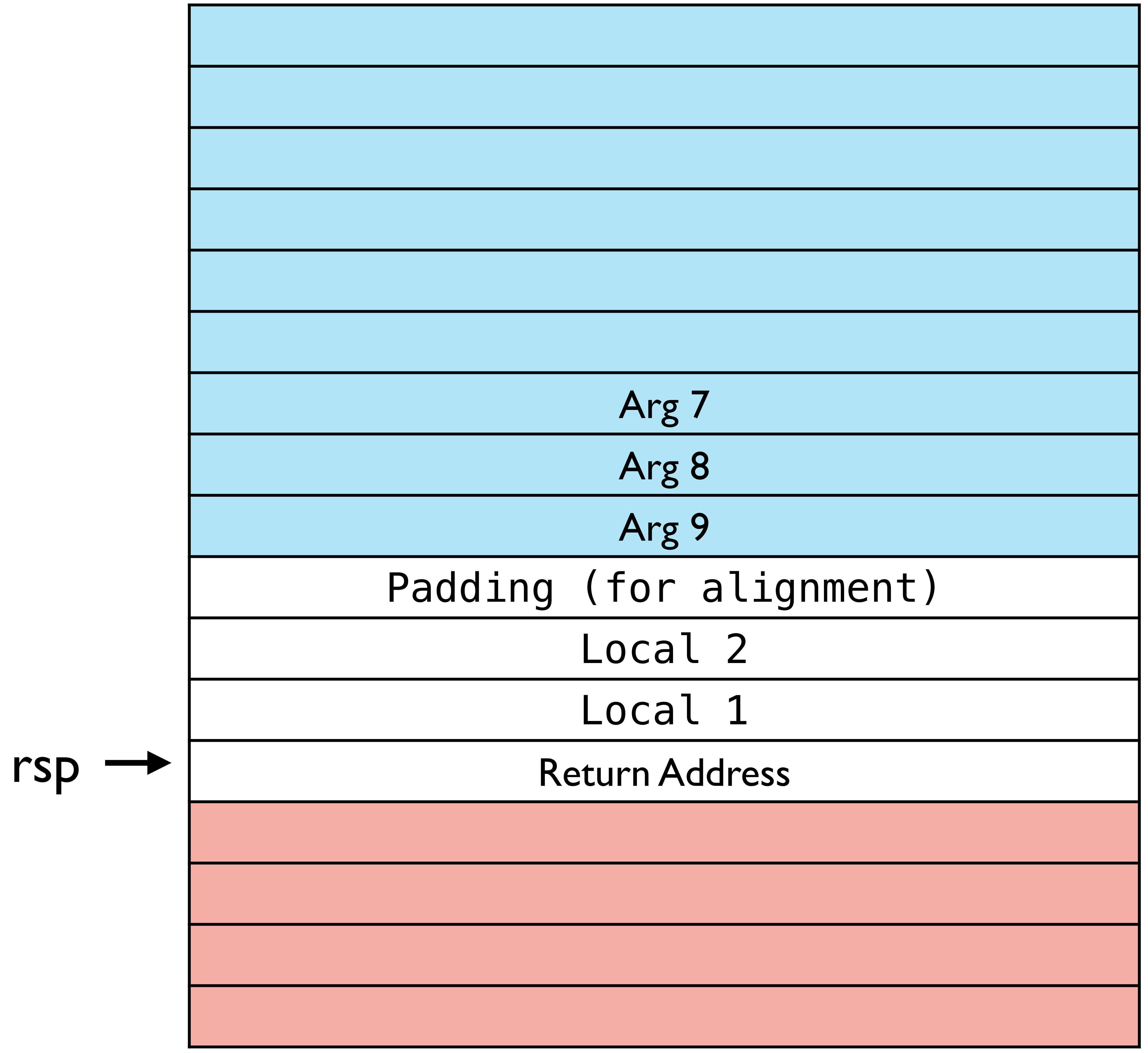
1. (Modern) Use rsp as the base pointer of the stack frame
2. (C style) Use rbp as the base pointer of the stack frame and rsp as the pointer to the **top** of the stack frame

Sys V AMD64 Calls

Live code examples: summer, big_fun

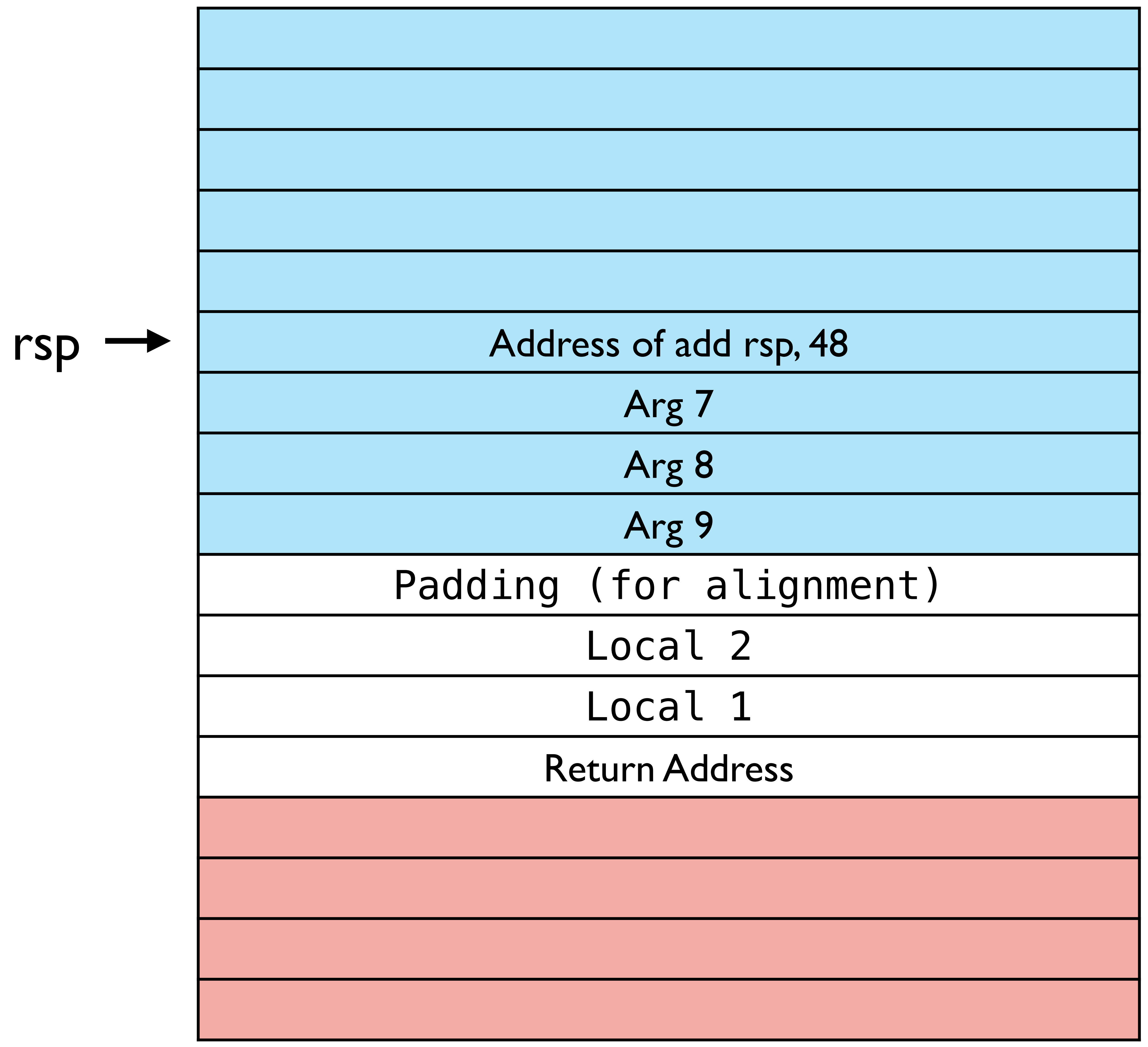
Sys V AMD64 Call

```
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov rcx, arg4
mov r8, arg5
mov r9, arg6
mov QWORD [rsp - 48], arg7
mov QWORD [rsp - 40], arg8
mov QWORD [rsp - 32], arg9
sub rsp, 48
call big_fun
add rsp, 48
```



Sys V AMD64 Call

```
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov rcx, arg4
mov r8, arg5
mov r9, arg6
mov QWORD [rsp - 48], arg7
mov QWORD [rsp - 40], arg8
mov QWORD [rsp - 32], arg9
sub rsp, 48
call big_fun
add rsp, 48
```


rsp
Address of add rsp, 48
Arg 7
Arg 8
Arg 9
Padding (for alignment)
Local 2
Local 1
Return Address