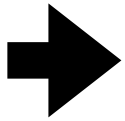
Part 7

Stack Machines

Programs

- Earlier, you developed a "model" for the structure of programs (the AST)

```
print 2 + 3 * 4;
```

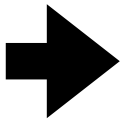


```
PrintStatement(
    BinOp('+',
          Integer(2),
          BinOp('*',
                Integer(3),
                Integer(4)
               )
         )
)
```

- Focus is on expressing program structure in the domain of the source language (syntax)

Modeling Computation

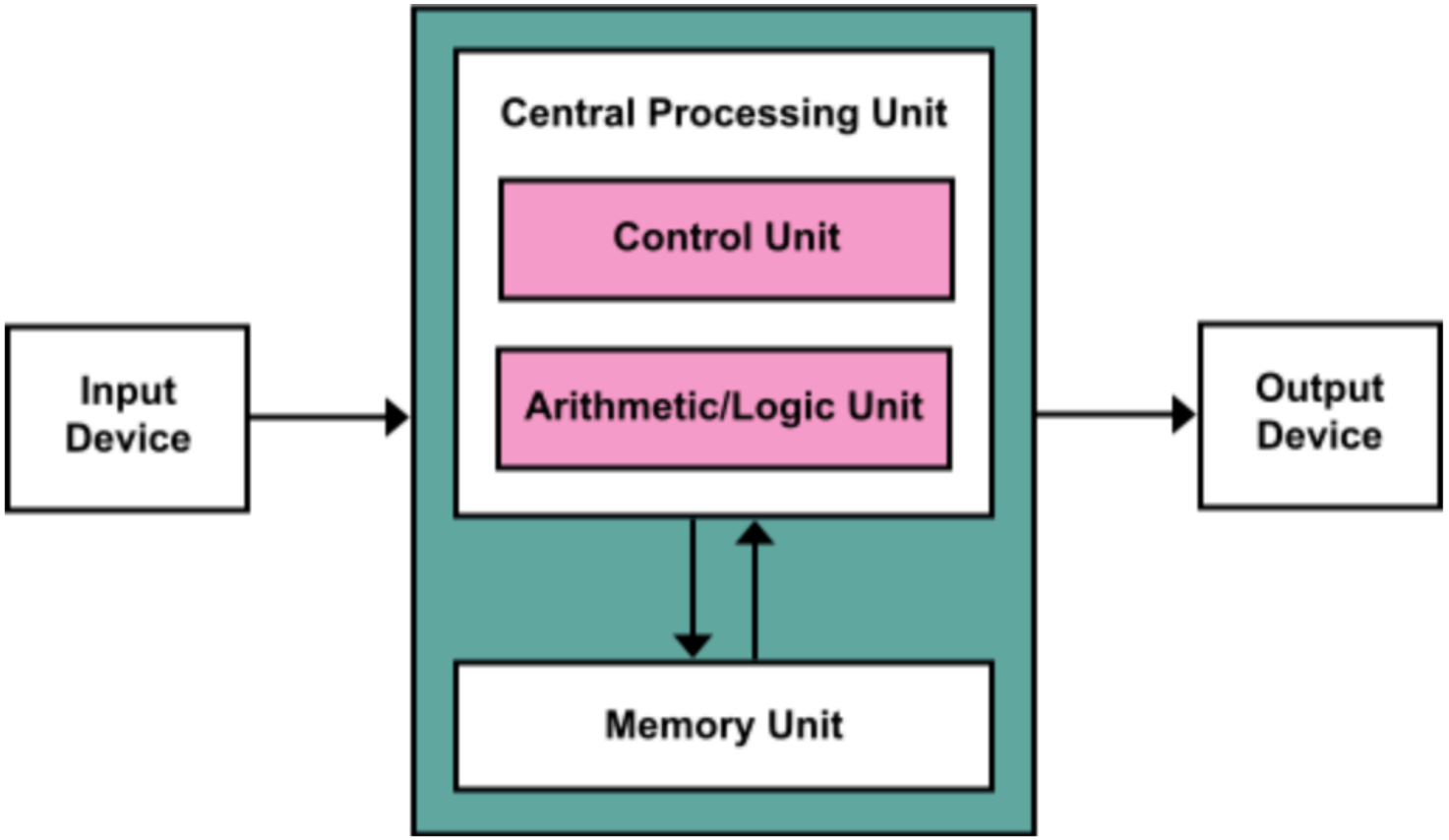
- There is a similar concept for machines.



????

- Specifically, most computers follow a fairly standard "model" of computation

von Neumann Machines



Arithmetic/Logic

- CPUs have instructions that perform single arithmetic operations

```
add, sub, mul, div, and, or, xor, not, eq, lt, ...
```

- There are two main datatypes (of varying sizes)

```
Integers (i8, i16, i32, i64, i128, etc.)
Floats (f32, f64, f128, etc.)
```

- Instructions would be typed (and possibly signed)

```
i32.add, i32.sub, i32.smul, i32.sdiv, i32.umul, i32.udiv,
f64.add, f64.sub, f64.mul, f64.div
...
```

Expression Evaluation

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 - 2) + 5
```

Expression Evaluation

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 – 2) + 5

2 + 3 * 8 + 5
```

Expression Evaluation

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 - 2) + 5

2 + 3 * 8 + 5

2 + 24 + 5
```

Expression Evaluation

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 – 2) + 5

2 + 3 * 8 + 5

2 + 24 + 5

26 + 5
```

Expression Evaluation

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 – 2) + 5

2 + 3 * 8 + 5

2 + 24 + 5

26 + 5

31
```

Expression Evaluation

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 – 2) + 5

2 + 3 * 8 + 5

2 + 24 + 5

26 + 5

31
```

- Key idea: Single steps

Stack Machines

- A high level abstraction for evaluation
- Operations are carried out on the stack

```
2 + 3 * (10 - 2) + 5
```

```
i32.push 2     [2]
i32.push 3     [2, 3]
i32.push 10    [2, 3, 10]
i32.push 2     [2, 3, 10, 2]
i32.sub        [2, 3, 8]
i32.mul        [2, 24]
i32.add        [26]
i32.push 5     [26, 5]
i32.add        [31]
```

Stack Machines

- Stack machines are quite common
  - Python
  - Java JVM
  - .NET CIL
  - WebAssembly
- Most interpreters are stack-based

Variables

- Programming languages have variables

```
const pi = 3.14159;

func area(radius float) float {
   var a = pi * radius * radius;
   return a;
}
```

- Variables appear in different contexts
  - Globals (outside of any function)
  - Locals (inside a function)
  - Parameters (inside a function)

Managing Variables

- Variables are managed via tables

```
const pi = 3.14159;

func area(radius float) float {
   var a = pi * radius * radius;
   return a;
}
```

globals

0:	('pi', 'f64')
	...

locals

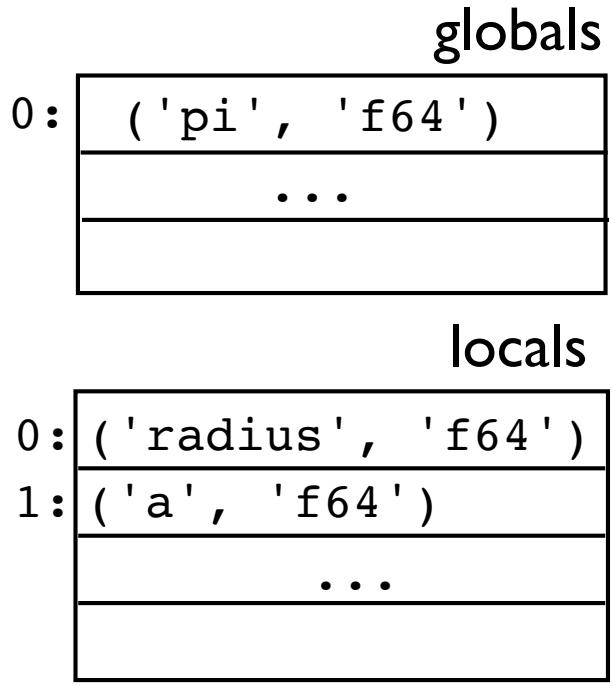
0:	('radius', 'f64')
1:	('a', 'f64')
	...

- Globals are shared by all
- Locals are per-function call
- Think about scoping rules - it mirrors that

Managing Variables

- Variables are managed via tables

```
const pi = 3.14159;

func area(radius float) float {
   var a = pi * radius * radius;
   return a;
}
```



- Abstract Instructions:

```
('load_global', slot)
('store_global', slot)
('load_local', slot)
('store_local', slot)
```

Managing Variables

- Variables are managed via tables

```
const pi = 3.14159;

func area(radius float) float {
   var a = pi * radius * radius;
   return a;
}
```

globals

0:	('pi', 'f64')
	...

locals

0:	('radius', 'f64')
1:	('a', 'f64')
	...

- Example:

```
load_global, 0       # "pi"
load_local, 0        # "radius"
f64.mul
load_local, 0        # "radius"
f64.mul
store_local, 1       # "a"
```

Control Flow

- Structured control flow (if, while, etc.)

```
if a < b {
   statements
} else {
   statements
}

while a < b {
   statements
}
```

- Introduces branching to the underlying code

Basic Blocks

- Consecutive statements often appear in groups

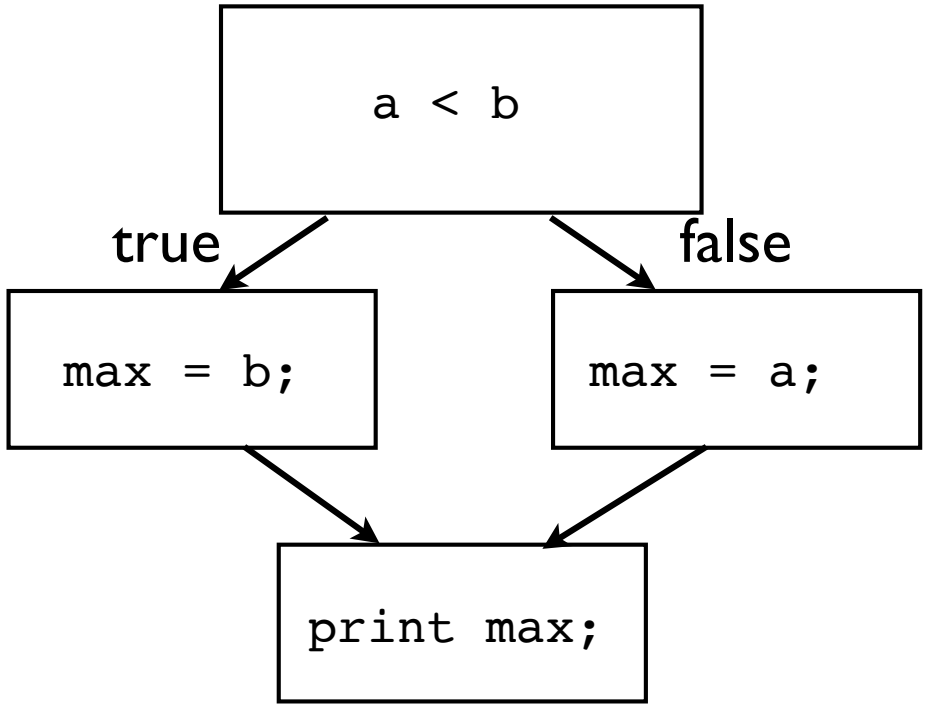
```
var a int = 2;
var b int = 3;
var c int = a + b;
print(2*c);
...
```

- A sequence of statements with no change in control-flow is known as a "basic block"

Control-Flow

- Structured control flow organizes code into basic blocks connected in a graph

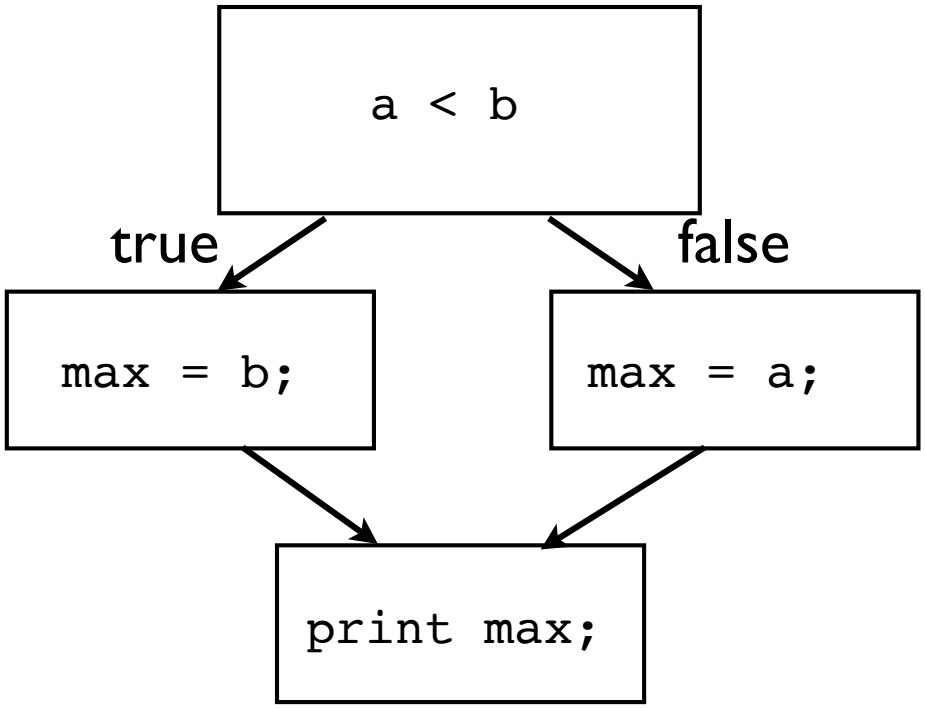
```
var a int = 2;
var b int = 3;
var max int;

if a < b {
   max = b;
} else {
   max = a;
}

print max;
```



- Control flow graph

Encoding Control Flow

- Label each block and emit jump/gotos




```
b1: test = a < b
    if (test) goto b2;
    goto b3;

b2: max = b;
    goto b4;

b3: max = a;
    goto b4;

b4: print max;
```

Functions

- Languages have functions

```
func square(x int) int {
    return x * x;
}

func main() int {
    var n = 0;
    while n < 10 {
        print square(n);
        n = n + 1;
    }
    return 0;
}
```

- Introduce runtime complexity.

Function Runtime

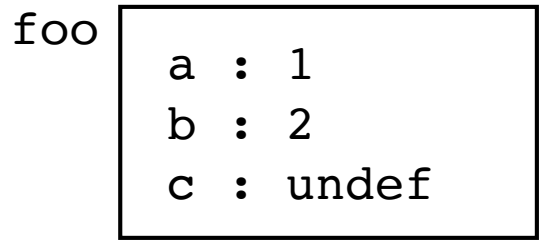
- Each invocation of a function creates a new environment of local variables
- Known as an activation frame (or record)
- Activation frames make up the call stack

Activation Frames

```
func foo(a int, b int) int {
    var c = a + bar(b);
    return c;
}

func bar(x int) int {
    y = 2*x;
    return spam(y);
}

func spam(z int) int {
    return 10*z
}

print foo(1,2);
```

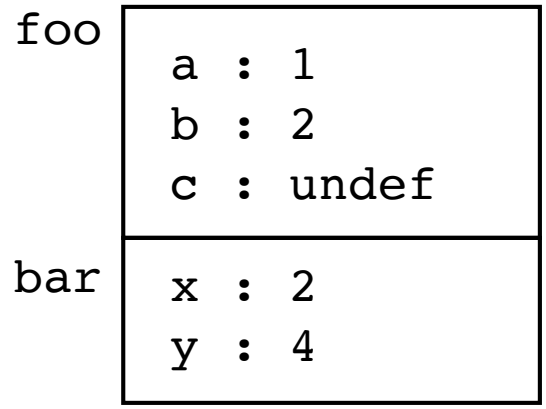
Activation Frames

```
func foo(a int, b int) int {
    var c = a + bar(b);
    return c;
}

func bar(x int) int {
    y = 2*x;
    return spam(y);
}

func spam(z int) int {
    return 10*z
}

print foo(1,2);
```



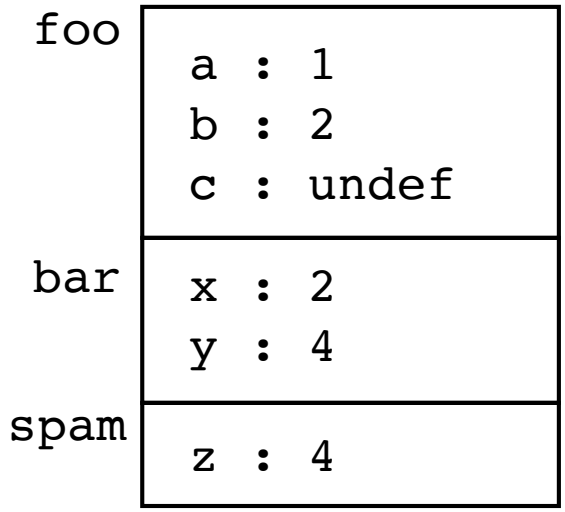
Activation Frames

```
func foo(a int, b int) int {
    var c = a + bar(b);
    return c;
}

func bar(x int) int {
    y = 2*x;
    return spam(y);
}

func spam(z int) int {
    return 10*z
}

print foo(1,2);
```



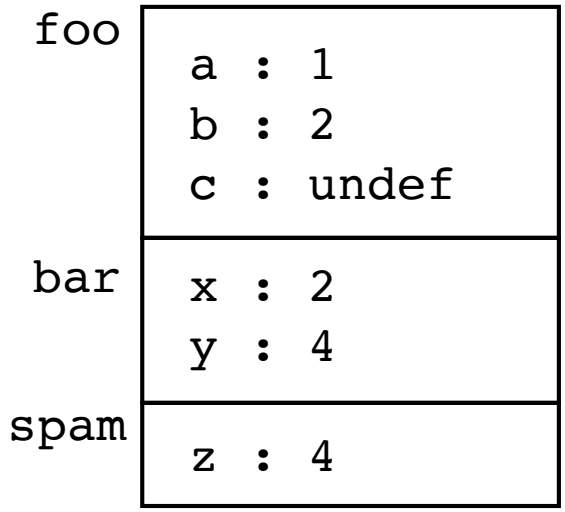
Activation Frames

```
func foo(a int, b int) int {
    var c = a + bar(b);
    return c;
}

func bar(x int) int {
    y = 2*x;
    return spam(y);
}

func spam(z int) int {
    return 10*z
}

print foo(1,2);
```



Activation Frames

```
func foo(a int, b int) int {
    var c = a + bar(b);
    return c;
}

func bar(x int) int {
    y = 2*x;
    return spam(y);
}

func spam(z int) int {
    return 10*z
}

print foo(1,2);
```



Note: Frames are NOT related to scoping of variables (functions don't see the variables defined inside other functions).

Project

- Compile Wabbit to a stack-based VM
  - See `wabbit/wvm.py`