# Write a Compiler

David Beazley
http://www.dabeaz.com

November 2019

# Materials

- Download and extract the following zip file

  http://www.dabeaz.com/compilers.zip

- Software requirements:
  - Python 3.7 (Anaconda recommended)
  - llvmlite and clang
  - A modern web browser

# Deep Thought

Programming

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

"Metal"


How does it all work????

# Metal

Machine Code (bits)

```
01011111111111110111010000001111
11000111110000001101110111011111
01111101111110111101111111111111
11011111110000000000000000000000
11000000000000010000000000000000
00000000000000010000000000000000
01111100011110011001010110010000
10000000110000001111111111111111
11000000000000000001011011010101
01000000000000000000000001000001
01000000010100000000000000000000
```



"Metal"


CPU is low-level (a glorified calculator)

# Assembly Code

```
fact:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    $1, -8(%rbp)
L1:
    cmpl    $0, -4(%rbp)
    jle     L2
    movl    -4(%rbp), %eax
    imull   -8(%rbp), %eax
    movl    %eax, -8(%rbp)
    mov     -4(%rbp), %eax
    addl    $-1, %eax
    movl    %eax, -4(%rbp)
    jmp     L1
L2:
    movl    -8(%rbp), %eax
    popq    %rbp
    retq
```



Machine Code

```
01011111111111101110100000111
11000111110000011011101101111
01111101111110111101111111111
11011111110000000000000000000
11000000000000100000000000000
00000000000000100000000000000
01111100011110011001011001000
10000000110000001111111111111
11000000000000000010110110101
01000000000000000000000100000
01000000010100000000000000000
```

"Human" readable machine code

# High Level Programming

## Source Code

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

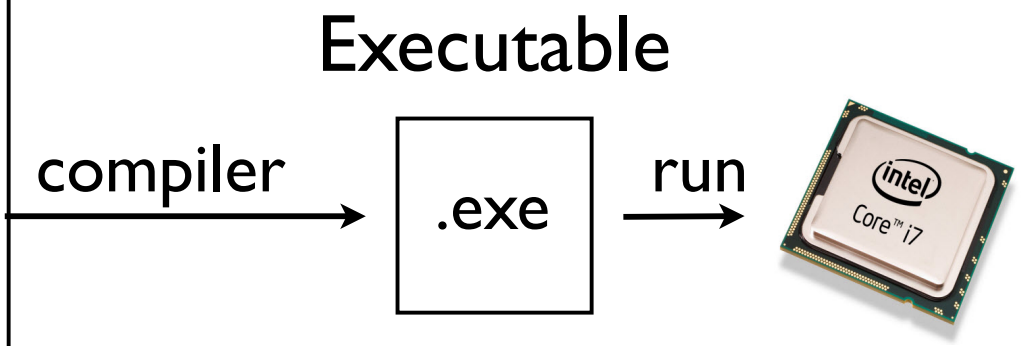
```
fact:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    $1, -8(%rbp)
L1:
    cmpl    $0, -4(%rbp)
    jle     L2
    movl    -4(%rbp), %eax
    imull   -8(%rbp), %eax
    movl    %eax, -8(%rbp)
    mov     -4(%rbp), %eax
    addl    $-1, %eax
    movl    %eax, -4(%rbp)
    jmp     L1
L2:
    movl    -8(%rbp), %eax
    popq    %rbp
    retq
```

"Human understandable" programming

# Compilers

Source Code

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```




Compiler: A tool that translates a high-level program into bits that can actually execute.

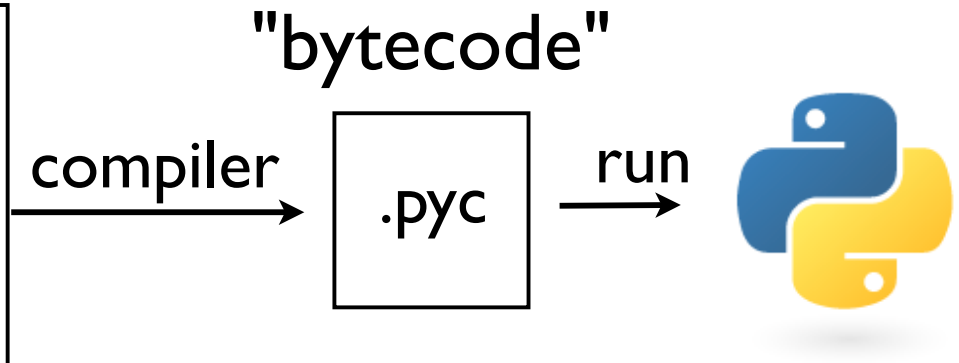
# Demo: C Compiler

```
#include <stdio.h>

int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}

int main() {
    int n;
    for (n = 0; n < 10; n++) {
        printf("%i %i\n", n, fact(n));
    }
    return 0;
}
```

# Virtual Machines

Source Code

```
def fact(n):
    r = 1
    while n > 0:
        r *= n
        n -= 1
    return r;
```




Many languages run virtual machines that work like high level CPUs (Python, Java, etc.)

# Demo: Python Bytecode

```
def fact(n):
    r = 1
    while n > 0:
        r *= n
        n -= 1
    return r
```

View bytecode:

```
>>> fact.__code__.co_code
b'd\x01}\x01x\x1c|\x00d\x02k\x04r |\x01|\x009\x00}\x01|
\x00d\x018\x00}\x00q\x06W\x00|\x01S\x00'
>>> import dis
>>> dis.dis(fact)
...
```

# Transpilers

Source Code

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```


Source Code

```
def fact(n):
    r = 1
    while n > 0:
        r *= n
        n -= 1
    return r
```

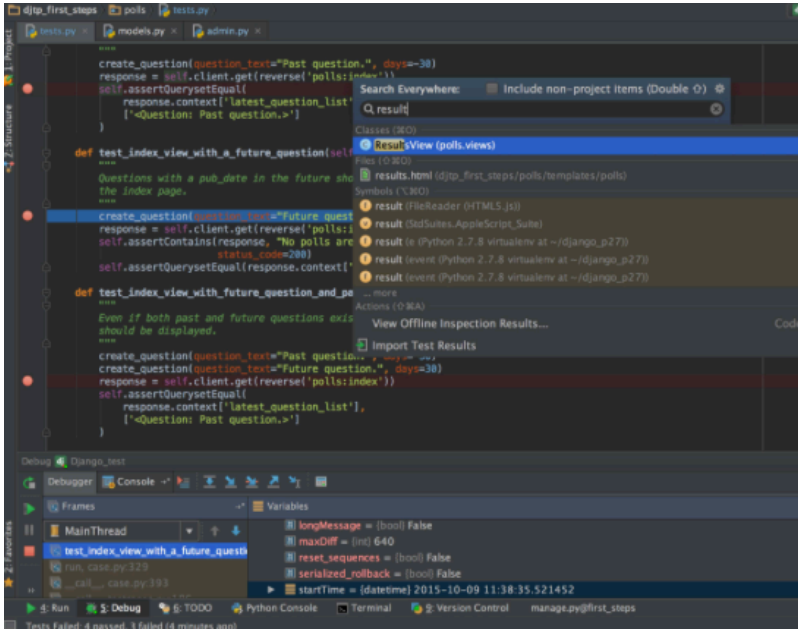
- Translation to a different language
- Example: Compilation to Javascript, C, etc.

# Other Tooling

## Source Code

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

checking/
analysis



- Code checking (linting, formatting, etc.)
- Refactoring, IDE tool-tips, etc.

# Background

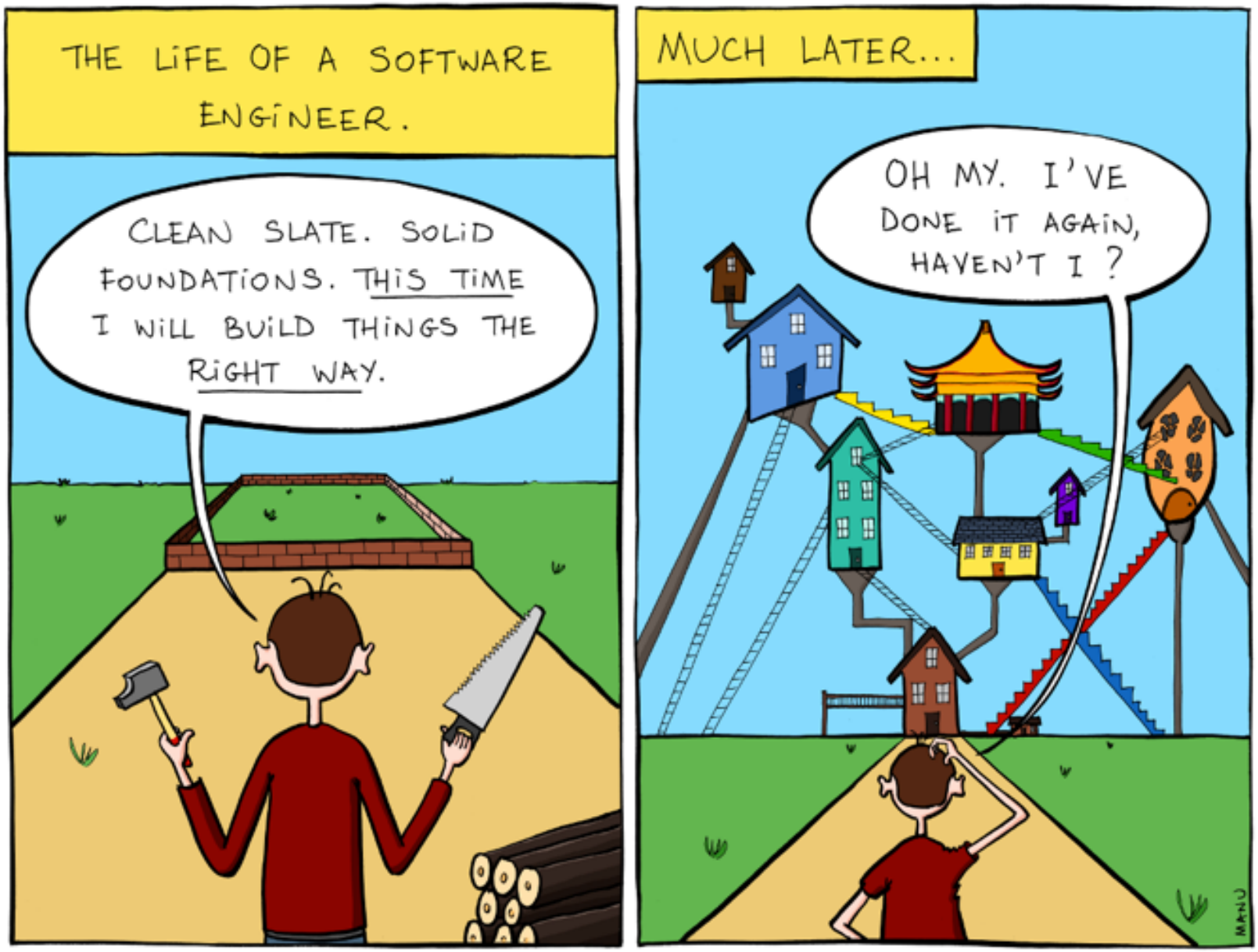
- Compilers are one of the most studied topics in computer science
- Huge amount of mathematical theory
- Tricky algorithms
- Programming language design/semantics
- The nature of computation itself

# Compilers are Current!

- Flurry of new languages (Rust, Go, Julia, etc.)
- New tech (WebAssembly, The Cloud, etc.)
- Opinion: We're in a period of transition

# Building a Compiler

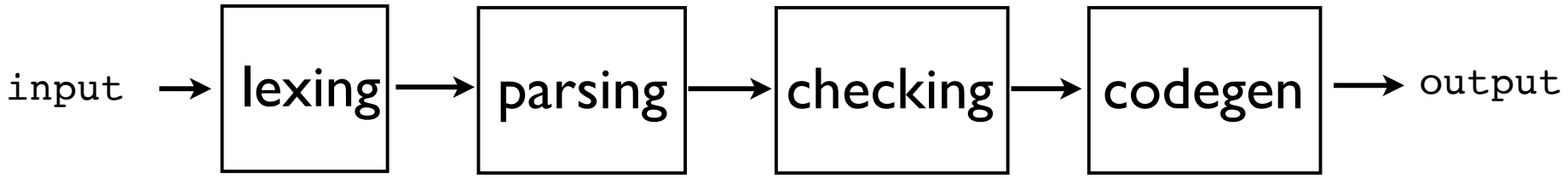
- It's one of the most messy programming projects you will ever undertake
- Many layers of abstraction (and often tooling)
- Involves just about every topic in computer science (algorithms, hardware, etc.)
- Difficult software design (lots of parts)




http://www.bonkersworld.net

# Behind the Scenes

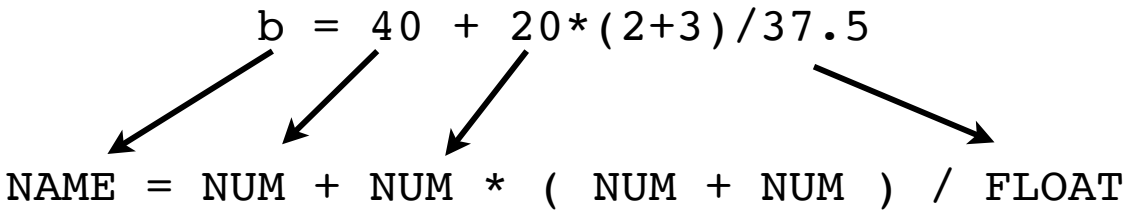
- Compilers are usually built as a workflow



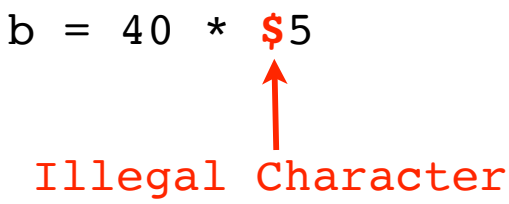

- Each responsible for a different problem.

# Lexing

- Splits input text into tokens




- Identifies words, detects illegal cruft




- Analogy: Take text of a sentence and break it down into valid words from the dictionary

# Parsing

"A ship shipping ship shipping shipping ships"

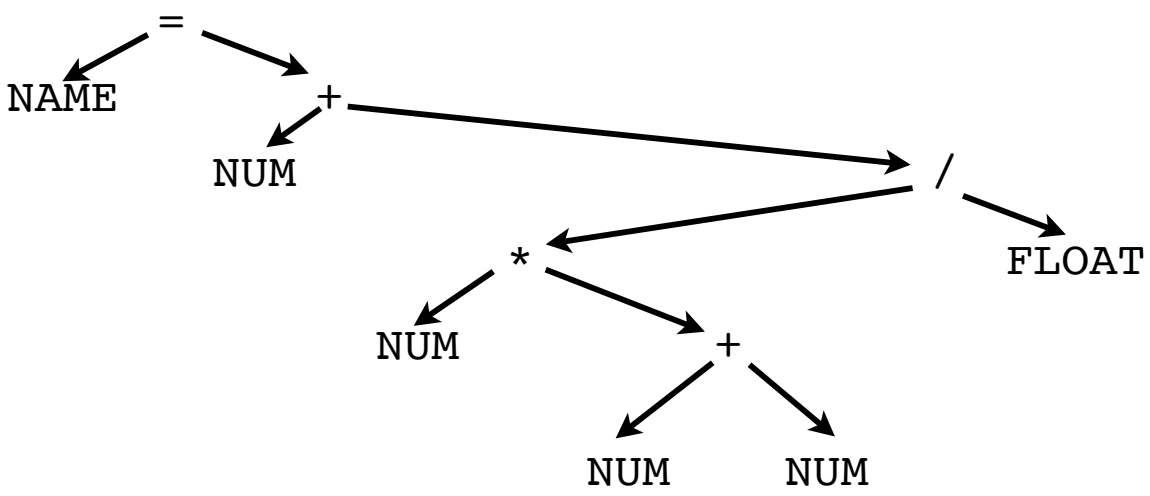
# Parsing

- Verifies that input is grammatically correct

```
b = 40 + 20*(2+3)/37.5
```

- Builds a data structure representing the code

```
NAME = NUM + NUM * ( NUM + NUM ) / FLOAT
```

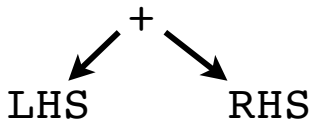


# Type Checking

- Enforces the rules (aka, the "legal department")

```
b = 40 + 20*(2+3)/37.5      (OK, Maybe?)
c = 3 + “hello”             (TYPE ERROR)
d[4.5] = 4                  (BAD INDEX)
```

- Example: + operator




1. LHS and RHS must be compatible types
2. The type must implement +

# Code Generation

- Generation of "output code":

```
b = 40 + 20*(2+3)/37.5
```

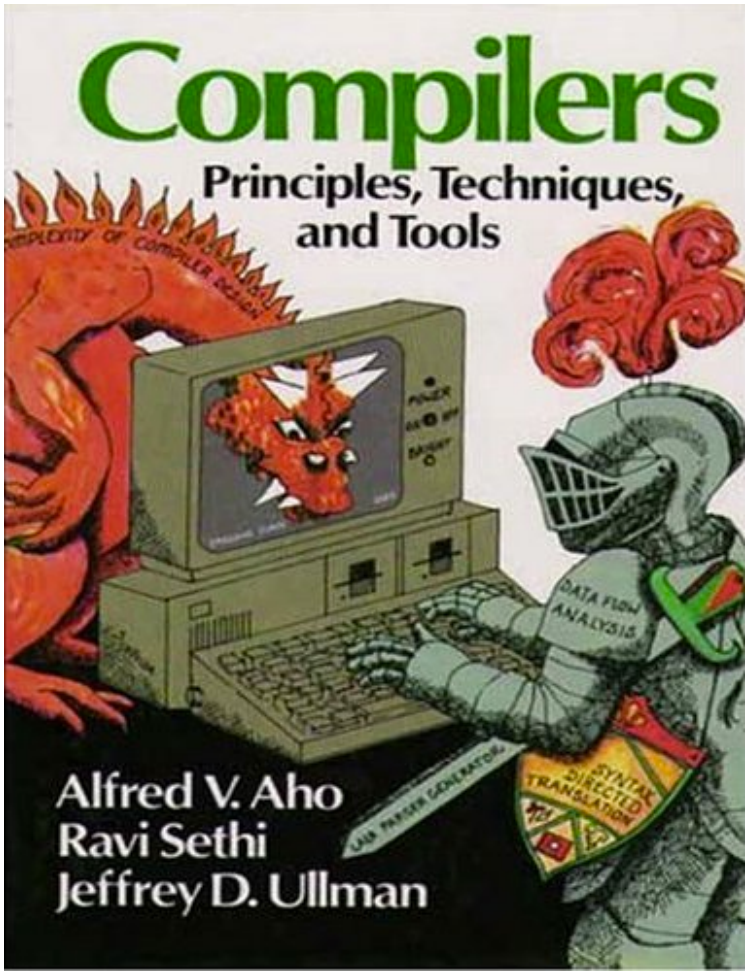
```
LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3  ; R3 = (2+3)
MUL  R2, R3, R2  ; R2 = 20*(2+3)
LOAD R3, 37.5
DIV  R2, R3, R2  ; R2 = 20*(2+3)/37.5
ADD  R1, R2, R1  ; R1 = 40+20*(2+3)/37.5
STORE R1, “b”
```

- Wide variety of possible "outputs" here

# Why Write a Compiler?

- Doing so demystifies a huge amount of detail about how computers and languages work
- It makes you a more informed developer
- It's a challenging software engineering project

# Books




- The "Dragon Book"
- Very mathematical
- Typically taught to graduate CS students
- Intense

# Teaching Compilers

- Mathematical approach
    - Lots of formal proofs, algorithms, possibly some implementation in a functional language (LISP, ML, Haskell, etc.)
- Systems approach (our approach)
    - Some math/algorithms, software design, computer architecture, implementation of a compiler in C, C++, Java.

# Heresy!

- Most compiler courses are taught in a narrative that follows the operation of a compiler
- Lexing -> Parsing -> Checking -> CodeGen
- Each stage builds upon the previous stage
- I am NOT going to follow that path
- Instead: The "Star Wars" narrative

# Now

Now

Day 1

understanding the problem

program checking

- Data Model
- Static Analysis
- Semantics

Now

understanding the problem

{

program checking

- Data Model
- Static Analysis
- Semantics

Day 1

program checking | Output

Day 3

Wasm, LLVM, etc.

Now

understanding the problem

{

program checking

- Data Model
- Static Analysis
- Semantics

Day 1

program checking | Output

Day 3

Wasm, LLVM, etc.

Input | program checking | Output

Day 4

Lexing/Parsing

# Project Demo

# Tips

- We are going to be writing a lot of code.
- I will be trying to point you in the right direction and to set the pace.
- It's a green-field project. You get no code!

# Making Progress

- Parts of the project are tricky
- It's not always necessary to solve all problems at once
- I will push you to more forward and come back to various problems later (it's okay)

# Teaching Philosophy

- I'm a proponent of "learn by doing"
- Much of what we cover in this course will be "discovered" in the process of building the project (as opposed to watching in slides)
- There is often no one "right" way to solve a problem (many possible solutions, with nuance)

# Caution

- For success, you need as few distractions as possible (work, world cup, child birth, etc.)

# Common Project Fails

- Testing: Write tests. Think about tests. Test what you can. Do NOT write a test framework.
- DRY: There is a lot of repetition. It may be faster to just repeat code than to figure out how to not.
- Clever Code: Yes, you could use metaclasses and decorators. Or you could write a compiler.
- Overthinking: It's easy to over-architect (i.e., OO design). Keep it simple. Can fix it later.

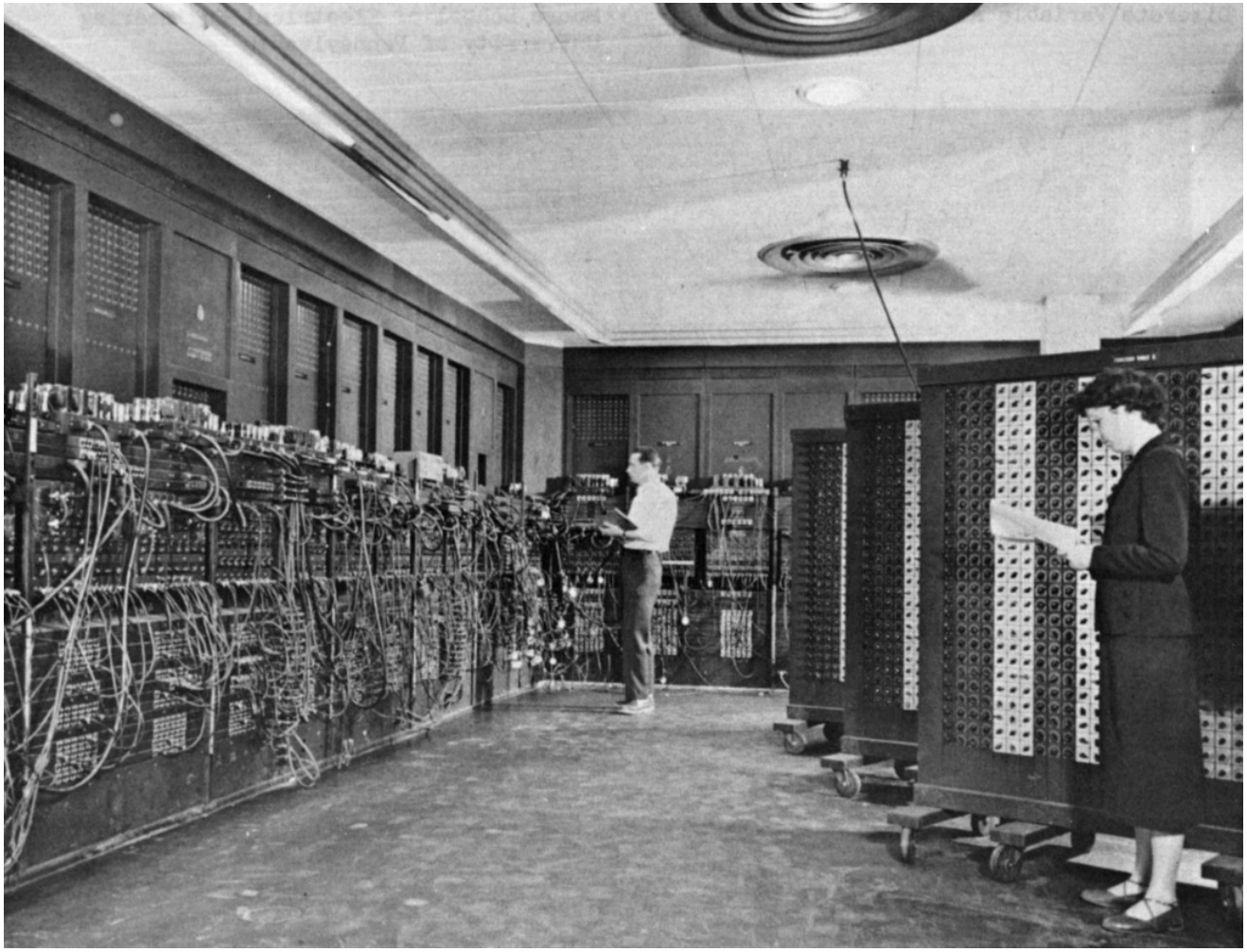
# Common Project Fails

- Silent Suffering: Get my attention if something is inexplicably broken. I have written the compiler about a dozen different ways. I have already suffered through the debugging and may have a quick answer ("oh, that's likely caused by X!").
- Going too fast: It's not a race. Spending extra time to solidify your understanding usually pays off later on.
- Food coma: Pace yourself ;-).

Part 0

# Preliminaries

# What is a Computer?



# What is a Computer?


CPU (Central Processing Unit)

registers

R0
R1
R2
R3
R4
R5
R6
R7

ALU

math

control unit

PC
SP

load

store

memory

data

```
5548 89e5 897d
fc00 0f8e 1800
45f8 8b45 fc83
8b45 f85d c300
```

program

```
ADD R1, R2, R3
SUB R3, R4, R5
...
STO R5, (R6)
```

# What is a Computer?

- Computers are not especially "smart"
- Glorified calculator (with program memory)
- Example:

```
2 + 3 * 4 - 5

MOV 2, R1              ; R1 = 2
MOV 3, R2              ; R2 = 3
MOV 4, R3              ; R3 = 4
MUL R2, R3, R4         ; R4 = R2 * R3 = 3 * 4 = 12
ADD R1, R4, R5         ; R5 = R1 + R4 = 2 + 12 = 14
MOV 5, R6              ; R6 = 5
SUB R5, R6, R7         ; R7 = R5 - R6 = 14 - 5 = 9
```

# What is Computation?

*"Computer Science is no more about computers than astronomy is about telescopes."*

What does it actually mean to "compute" something?

What can be computed?

# What is Computation?

- It's a process or procedure

```
Compute: 5!    -> 5 * 4 * 3 * 2 * 1

result = 1
n = 5
while n > 0:
    result = result * n
    n = n - 1
```

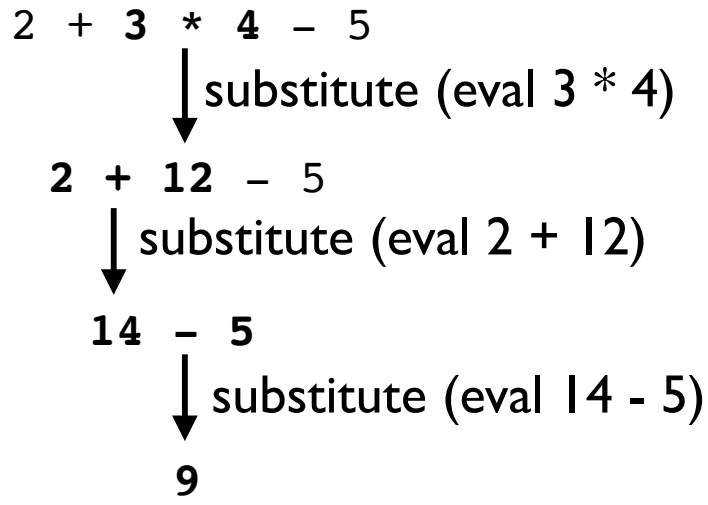
- Step-by-step sequence of operations

# What is Computation?

- But, what is the actual essence of "computation?"
- Give me a minimal definition of it...

# Deep Idea

- Computation is repeated substitution




- Maybe overly simplified, but computation is a process of substituting one thing for another
- Stops when no more steps are possible

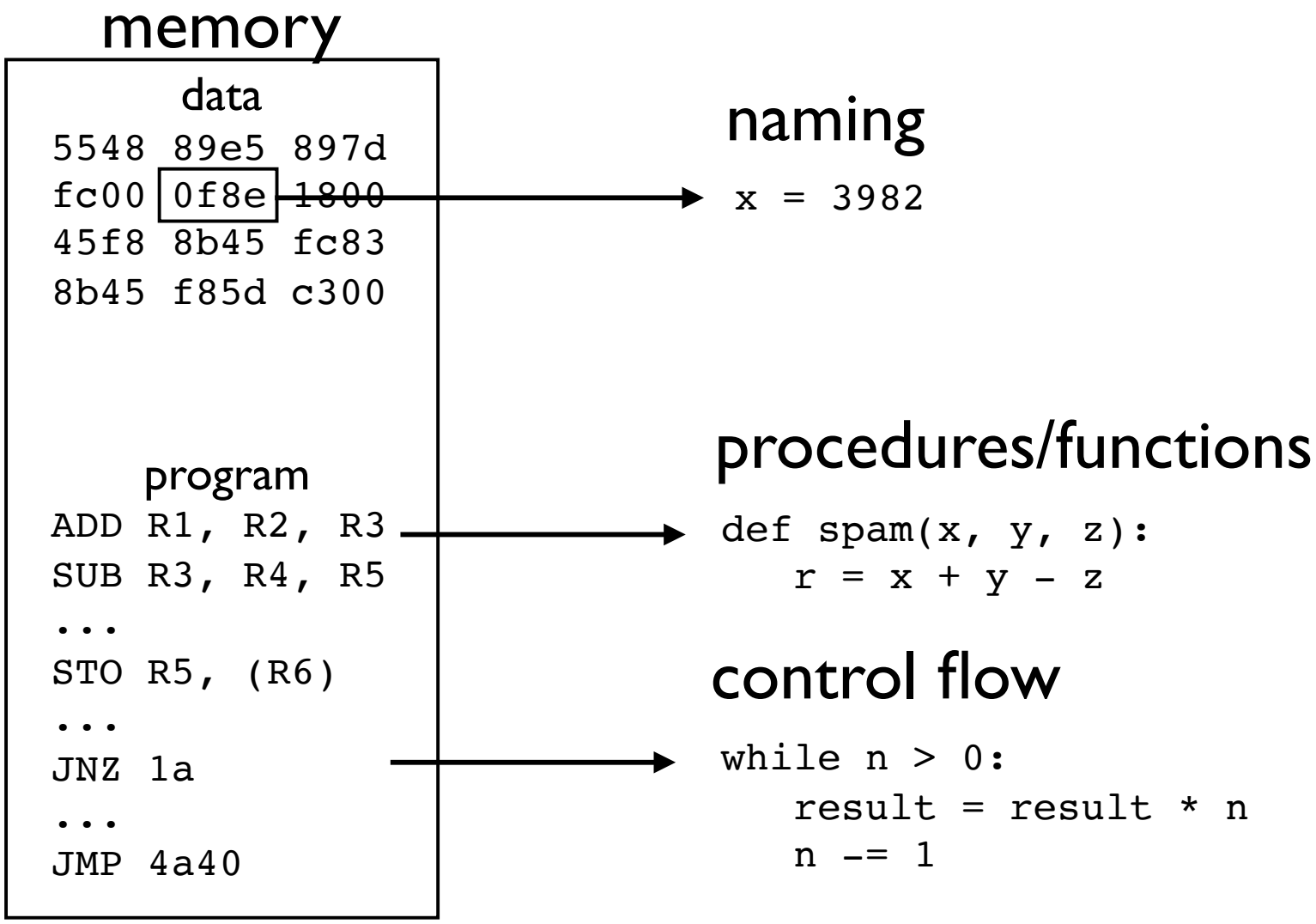
# Substitution

- Substitution is a major facet of compiler writing
- "High level" things are replaced by "lower level" things (repeat until you can't go any further)
- Part of moving from an abstract programming language down to the actual hardware
- Also a major part of the mathematical foundations of programming languages.

# What is Programming?

- Describes a computational process. Yes.
- How? By banging random keys on a keyboard??
- Key feature: Abstraction

# Programming is Abstraction



# Programming is Abstraction

C Python Go Perl Java Lisp Rust Haskell Scala Javascript

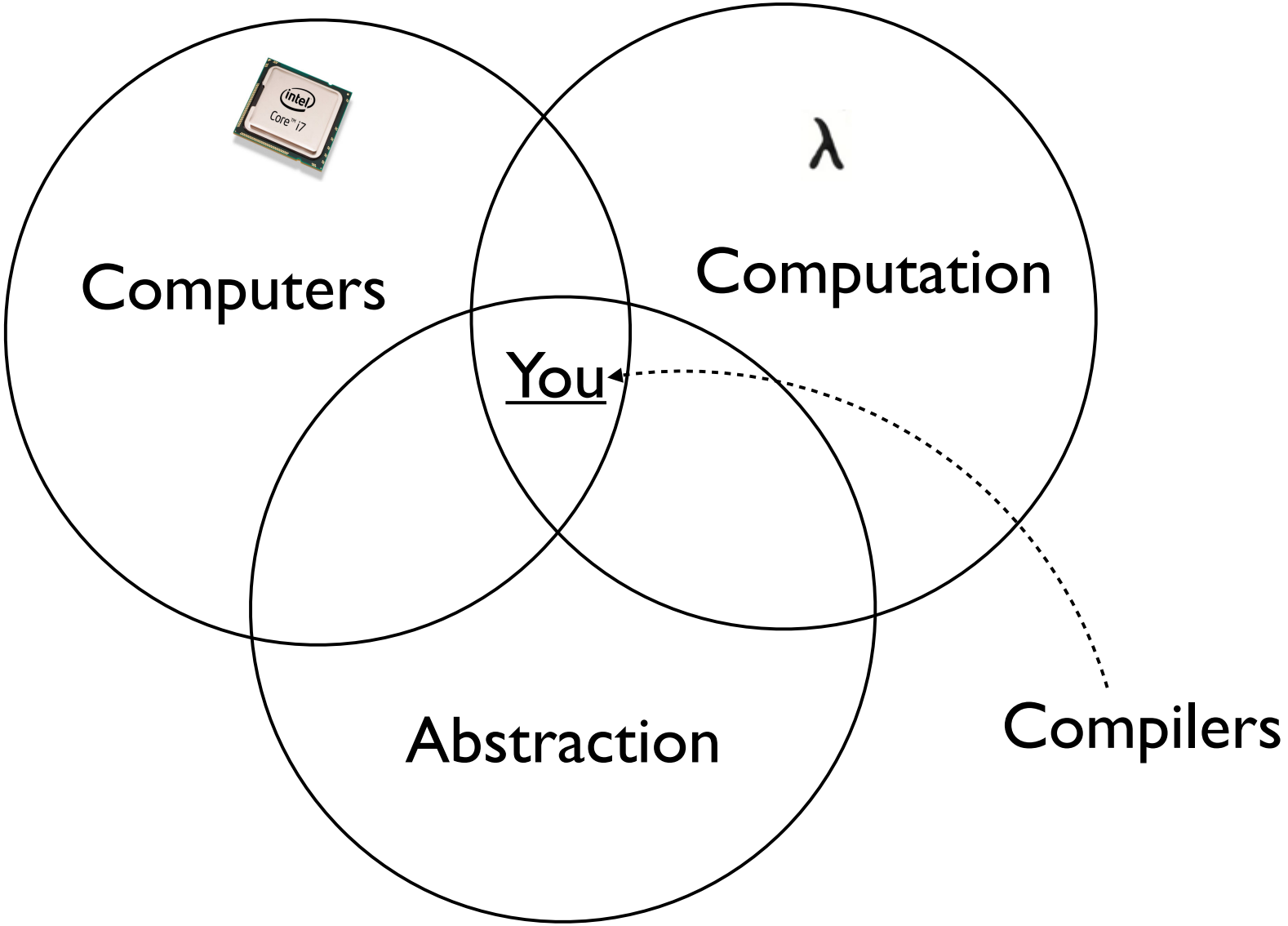
Abstraction


"Metal"

```
010111111                    00001111
110001111                    11011111
011111011                    11111111
1101111110000000000000000000000
1100000000000010000000000000000
```

Bits

# Big Picture



# Project 0

Find the file `metal.py`

Follow the instructions found inside.

Let's code...

# Project 0.5

Play with wabbit

Read docs/wabbit.html

Find the directory `SillyWabbit/`

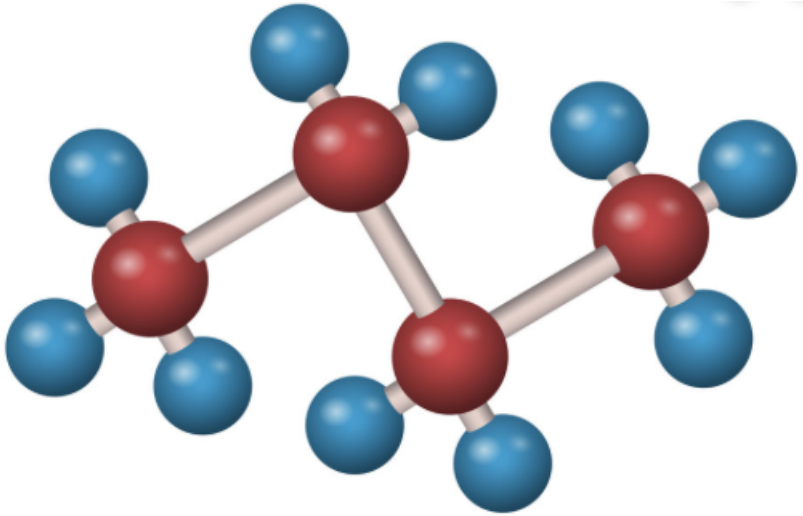
Follow instructions in the README

Part I

# The Structure of Programs

# The Elements

- Question: What are the basic elements that make up computer programs?
- Can you deconstruct a program down into a collection of objects?
- In essence: A data model



# Primitives

- There are primitive values

```
34                  # Integer
3.4                 # Float
'T'                 # Character
True                # Boolean
```

- The primitives are the most basic things
- Indivisible
- The foundation of everything else

# Types

- There is usually an underlying type system

```
int
float
char
bool
```

- Values always have an associated "type"
- It's required to map operations onto actual hardware (e.g., integer operations vs floating point operations).

# Names

- You can name things

```
r = 2.0;
pi = 3.14159;
area = pi * r * r;
```

- You don't hardcode values, use a name
- Naming -> "Abstraction"

# Expressions

- There are operators

```
3 + 4 * 5
tau = 2 * pi
```

- And rules for evaluation order (left-right)

```
3 - 4 - 5          # -> (3 - 4) - 5
```

- And rules for precedence

```
3 - 4 * 5          # -> 3 - (4 * 5)
```

- An expression produces values

# Assignment

- Computers have memory for storage
- Load/store operations

```
3 + x;          // Value is read from "x"
x = 4 + 5;      // Value is stored in "x"
```

- "storage location" is a complex concept

```
x               // Simple value
x[n]            // Indexing (arrays)
x.attr          // Attribute (structures)
```

- Locations can appear on either side of =

```
x[n] = y.attr;
```

- Mutable vs Immutable

# Data Structures

- There may be more complex types

```
int [10]          // Arrays

struct Point {    // Records
    x int;
    y int;
}
```

- Arrays, pointers, tuples, structures, classes.
- Built as extensions to the type system

# Control Flow

- You can make decisions

```
if a > b {
    m = a;
} else {
    m = b;
}
```

- And perform repeated operations

```
while n > 0 {
    print('T-minus', n);
    n = n - 1;
}
```

# Functions/Procedures

- Defining a function

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

- Applying a function (produces a value)

```
3 + square(10)
```

- How does it work? ("substitute the x")

# Declarations

- Names might need to be explicitly declared

```
// Declarations
var r float;
var area float;
const pi = 3.14159;

// Assignment (names must exist already)
r = 2.0;
area = pi*r*r;
```

- Declarations bind names to a scope
- Compiled languages like C require it
- Python does not (assignment is declaration)

# Environments/Scopes

- Names are stored in environments (scopes)

globals

```
const pi = 3.14159;
var x int;
```

locals

```
func fact(n int) int {
    var result int = 0;
    while n > 0 {
        result = result * n;
        n -= 1;
    }
    return result;
}
```

- Scopes are nested (lexical scoping)

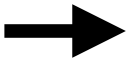
# Programs as Data

- Code elements can be represented as data
- Not text, but as concrete objects

```
23
```

```
class Integer:
    def __init__(self, value):
        self.value = value
```

```
location = value;
```



```
class Assignment:
    def __init__(self, location, value):
        self.location = location
        self.value = value
```

```
left + right;
```

```
class BinOp:
    def __init__(self, op, left, right):
        self.op = op
        self.left = left
        self.right = right
```

# Programs as Data

- Example

```
x = 23 + 42;

Assignment(
  NamedLocation('x'),
  BinOp('+', Integer(23), Integer(42))
)
```

- Commentary: A major part of writing a compiler is in designing and building the data model. It directly reflects the structure and features of the language that's being compiled.

# Commentary

- A structurally correct program is not necessarily a correct program
- It just looks correct... (like the movies)

# Project 1

Reading

- `docs/wabbit.html`

Find the files

- `wabbit/model.py`
- `programs.py`

Follow the instructions inside (with guidance)

Part 2

# The Semantics of Programs

# Structure vs. Semantics

- Data model describes program structure

```
23 + 4.5  ⟶  BinOp('+', Integer(23), Float(4.5))
```

- Semantics are the legal department
- Is that operation allowed?
- Answer: It depends. What are the rules?

# Type Systems

- Programming languages have different types of data and objects

```
a = 42                # int
b = 4.2               # float
c = "fortytwo"        # str
d = [1,2,3]           # list
e = {'a':1,'b':2}     # dict
...
```

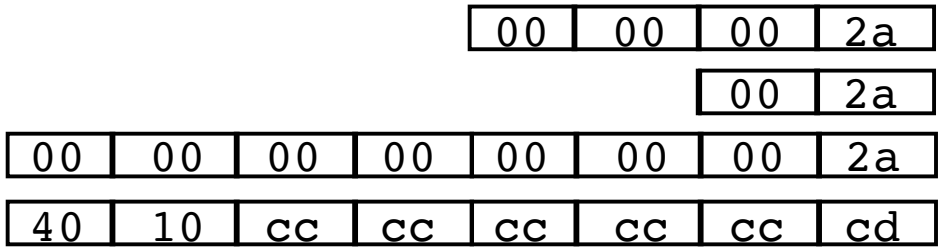
- Each type has different capabilities

```
>>> a - 10
32
>>> c - "ten"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'st
>>>
```

# What is a Type?

- Partly relates to underlying data encoding

```
int a = 42;
short b = 42;
long c = 42;
float d = 4.2;
```

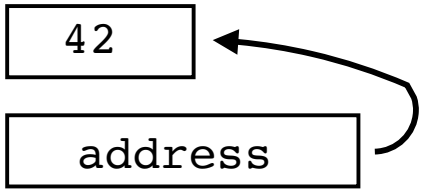



- Must map to low-level operations on CPU
- Different kinds of instructions (int vs. float)

# Derived Types

- Types might also encode data abstractions
- Pointers/References

```
int a = 42;

int *b = &a;
```

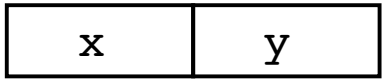



- Arrays

```
int items[4];
```


- Structures

```
struct Point {
   int x;
   int y;
}
```



# Complexity: Composition

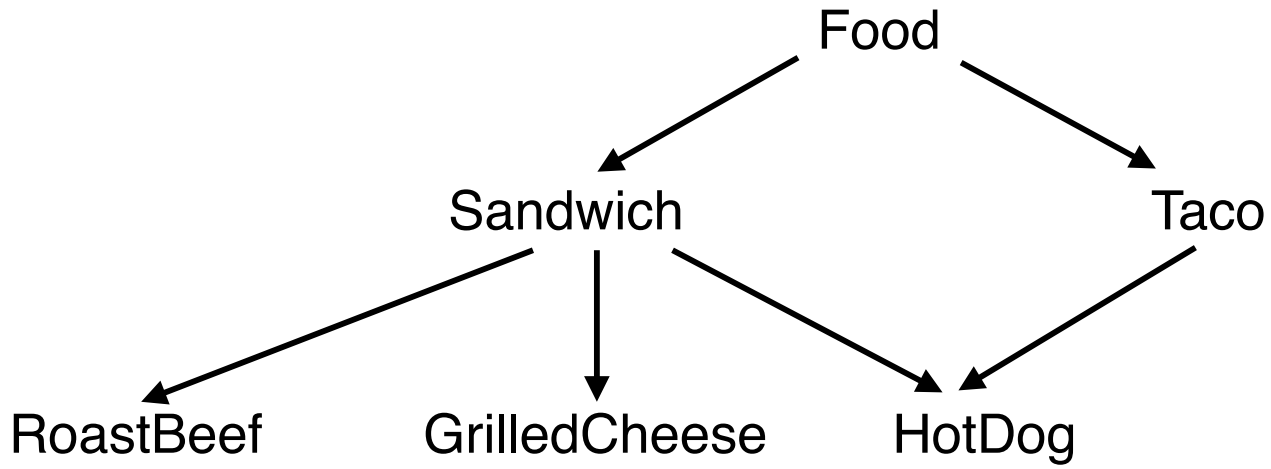
- Types combine together in arbitrary ways

```
int a;     // Base type
int *b;    // Pointer to int
int **c;   // Pointer to pointer to int
int d[4];  // Array of int
int *d[4]; // Array of pointers to int
```

- Might also have qualifiers (const, mutable, etc.)
- Comment: This gets quite complicated. How do you structure and reason about it?
- The topic of a different course (type theory)

# Complexity: Ontologies

- Type system might support "inheritance"




- Suddenly you're now in OO hell...
- We're not doing that (but be aware of it)

# Comment

- Python has an extremely simple and minimal type system (almost too simple).
- All values have an attached class (that's it)
- No pointers, no mutability, no records, etc.
- Inheritance and multiple inheritance
- Optional "type hinting" is more complex

# Type Checking

- Enforcing semantics on the program model
- A lot of it is common sense
  - Can't perform operations (+,-,*,/) if not supported by the underlying type
  - Can't overwrite immutable data
  - Array indices must be integers

# Dynamic Typing

- Rules are enforced at run-time
- Objects carry type info during execution

```
>>> a = 42
>>> a.__class__
<class 'int'>
>>> a + 10
52
>>> a.__add__(10)
52
>>> a.__add__('hello')
NotImplemented
>>> a + 'hello'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
>>>
```

# Static Typing

- Rules are enforced at compile-time
- Code is annotated with explicit types

```
/* C */
int fact(int n) {
    int result = 1;
    while (n > 0) {
        result *= n;
        n--;
    }
}
```

- Compiler executes a "proof of correctness"
- Types are discarded during execution

# Note

- Most programming languages involve a mix of both techniques (static/dynamic)
- Compiler does as much as it can
- Certain checks may be forced to run-time
- Example: Array-bounds checking

# Building a Type System

- A Few Minimal Requirements:
  - Must be able to specify types
  - Need to know type capabilities
  - Must be able to check the data model

# Type Specification

- There are named primitive types
- Types are "labels" that are attached to values

```
float x
int fact(int n);
string name;
```

- Keep in mind that the "type" could be much more complex (pointers, arrays, structs, etc.)
- We will keep it simple.

# Type Specification

- Types must be comparable

```
int != float
```

- A major part of checking is finding type-mismatches in the code (comparing types)
- A simple name comparison might suffice
- "Nominal Typing"

# Thought

- Are these two types the same type?

```
struct Point {
    float x;
    float y;
};

struct Coordinates {
    float x;
    float y;
};
```

- Related to "Structural Typing"

# Type Specification

- Types have different capabilities

```
int:
   binary_ops = { '+', '-', '*', '/' },
   unary_ops = {'+','-'}

string:
   binary_ops = {'+'},
   unary_ops = {}
```

- Checker will consult when validating

# Type Propagation

- Type information propagates

```
Integer(2)          ; Type=int
Integer(3)          ; Type=int

BinOp('+',          ; Type=int (determined from operands)
      Integer(2),
      Integer(3)
)
```

- Checker not only validates, but fills in additional information about what is known.

# Enforcing The Rules

- You must write a rule checking function for each kind of object in the data model

```
def check_BinOp(node):
    ...

def check_UnaryOp(node):
    ...

def check_Assignment(node):
    ...
```

- Each function both validates and propagates information (recursive)

# Example:

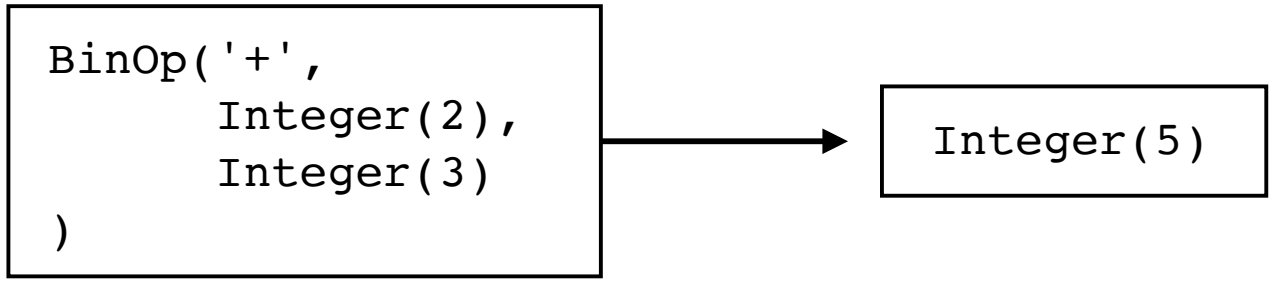
```
def check_BinOp(node):
    # Check the operands (recursively)
    check(node.left)
    check(node.right)

    # Check if the combination of types/operator
    # is allowed or not
    if supported_binop(node.op,
                       node.left.type,
                       node.right.type):
        # Set the type of the result
        node.type = result_type
    else:
        error('Unsupported operation')
```

# Confusion

- Program checking is NOT the same thing as running a program
- You don't carry out any calculations
- You don't load/store from memory
- A compiler ultimately creates a runnable program as output

# Simplification

- A type-checker *could* perform simplification




- Example: Constant folding
- It's tricky--calculation is performed in the compiler as opposed to in the final program

# Checking of Names

- Type checking also involves name checking
- You may have an expression:

```
42 + x
```

- What is "x"?
- Languages usually require a name declaration

```
var x int;
```

- No declaration -> Name Error

# Symbol Tables

- Names are managed in a symbol table
- In a nutshell: A dict mapping {name: decl}

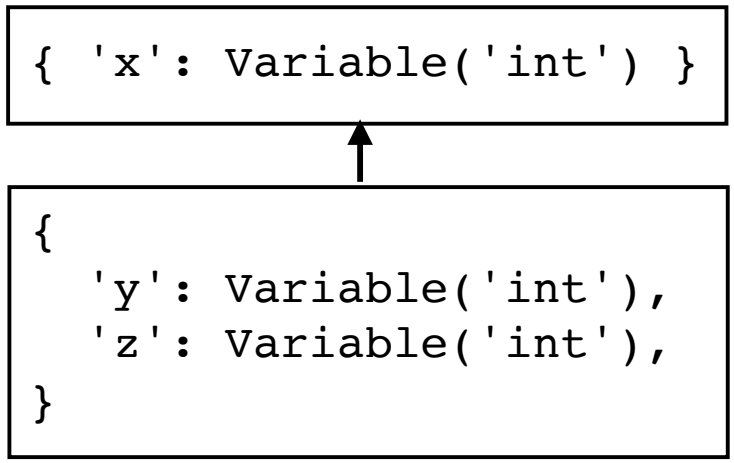
```
symbols = {
            'x': Variable(type="int")
          }
```

- Declarations insert names in the table
- Later name references consult the table

# Symbol Tables and Scopes

- There is a direct mapping between symbol tables and scopes

```
var x int;          // Global

func spam() int {
    var y int;      // Locals
    var z int;
    ...
}
```




- Implemented via chained dictionaries

# Project 2

- Find the files
  - `wabbit/checker.py`
  - `wabbit/typesys.py`
- Follow instructions inside.
- Our goal is to perform extensive validation on the program data model developed in the previous project. Ideally: We want to make it impossible to specify an invalid program.

Part 3

# Intermediate Code

# Let's Make Code

- A compiler ultimately has to make output
  - Assembly code
  - C code
  - Virtual machine instructions
- How do you do it?

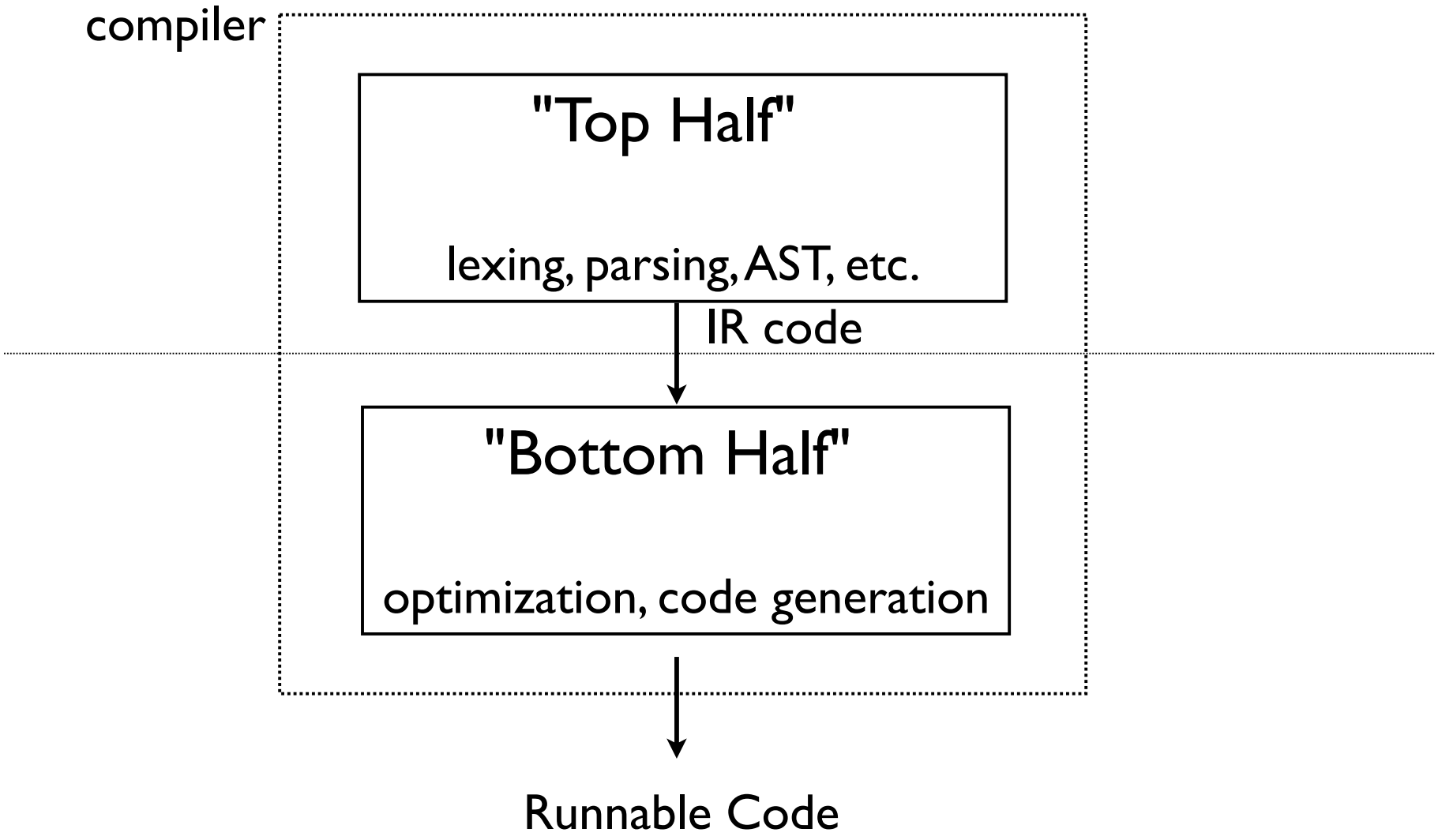
# Backing up....

- Writing a compiler is hard
- Do you make it target just one thing?
- For example, a single model of a CPU?
- Usually not
- There is an abstraction of "hardware"

# Intermediate Code

- Compilers usually generate an abstract intermediate code instead of directly emitting low-level HW instructions
- Intermediate code is sort of a generic "machine code"
- Easier to analyze and translate

# Compiler Design



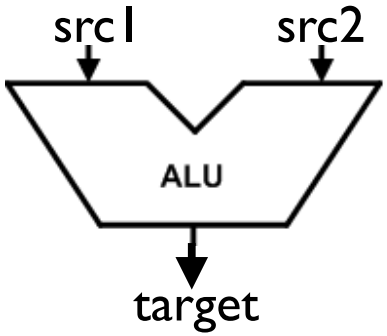
# Designing an IR

- Intermediate Representation is usually modeled after actual computer hardware
- Registers, memory, low-level operations
- However, a lot of constraints are removed (for example, infinite memory)

# Three-Address Code

- A common IR where most instructions are tuples `(op,src1,src2,target)`

```
('ADD', a, b, c)        # c = a + b
('SUB', x, y, z)        # z = x - y
('LOAD', a, b)          # b = a
```




- Closely mimics design of an ALU
- The initial warmup exercise was 3AC

# Three-Address Code

- Example of three-address code IR

```
c = 2*a + b - 10

    r1 = 2              ('CONST',2,'r1')
    r2 = a              ('LOAD','a','r2')
    r3 = r1 * r2        ('MUL','r1','r2','r3')
    r4 = b              ('LOAD','b','r4')
    r5 = r3 + r4        ('ADD','r3','r4','r5')
    r6 = 10             ('CONST',10,'r6')
    r7 = r5 - r6        ('SUB','r5','r6','r7')
    c = r7              ('STORE','r7','c')
```

- Calculations are broken down into steps that carry out one operation at a time

# SSA Code

- Single Static Assignment
- A restriction of 3-address code
  - Infinite registers
  - Registers are immutable
  - Can never reassign a previously used register
- This is basis of systems such as LLVM

# Stack Machine

- Another common IR abstraction: get rid of registers altogether and use a stack
- General idea
  - Operands get pushed onto stack
  - Operators consume stack items
  - Result gets pushed back onto stack

# Example : Stack Machine

- Example: Compute: 2 + 3 * 4

| Instructions | Stack |
|---|---|
| `PUSH 2` | `[2]` |
| `PUSH 3` | `[2, 3]` |
| `PUSH 4` | `[2, 3, 4]` |
| `MUL` | `[2, 12]` |
| `ADD` | `[14]` |

- Common stack machines
  - JVM (Java)
  - Python
  - WebAssembly

# Code Generation

- Code generation for stack machines is often not much more than a basic traversal of the program structure (data model)
- Walk the nodes and emit instructions

# Example:

```
def emit_Integer(node, code):
    code.append(('PUSH', node.value))

def emit_BinOp(node, code):
    emit(node.left)
    emit(node.right)
    if node.op == '+':
        code.append(('ADD',))
    elif node.op == '-':
        code.append(('SUB',))
    elif node.op == '*':
        code.append(('MUL',))
    ...
```

# Digression

- Stack machines can be converted into 3AC

| Stack IR | 3AC | Stack |
|---|---|---|
| `PUSH 2` | `('CONST', 2, 'r1')` | `[r1]` |
| `PUSH 3` | `('CONST', 3, 'r2')` | `[r1, r2]` |
| `PUSH 4` | `('CONST', 4, 'r3')` | `[r1, r2, r3]` |
| `MUL` | `('MUL','r2','r3','r4')` | `[r1, r4]` |
| `ADD` | `('ADD','r1','r4','r5')` | `[r5]` |

- Use a stack to track the register names
- Emit 3AC codes with names from stack

# The World of Registers

- IR is where a lot of optimization takes place
  - Identifying repeated patterns
  - Reusing values
  - Optimal register allocation
- Frankly, this is a LOT of magic (topic of a more advanced compiler course)

# Optimization

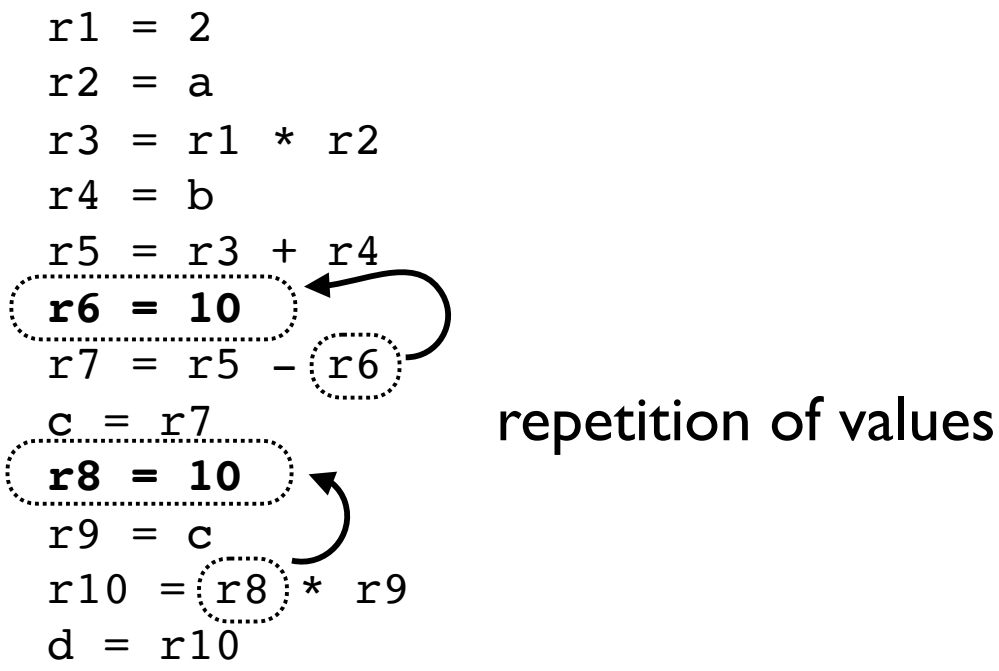
- Example: peephole optimization

```
r1 = 2
r2 = a
r3 = r1 * r2
r4 = b
r5 = r3 + r4
r6 = 10
r7 = r5 - r6
c = r7
r8 = 10
r9 = c
r10 = r8 * r9
d = r10
```

- A search for unnecessary instructions

# Optimization

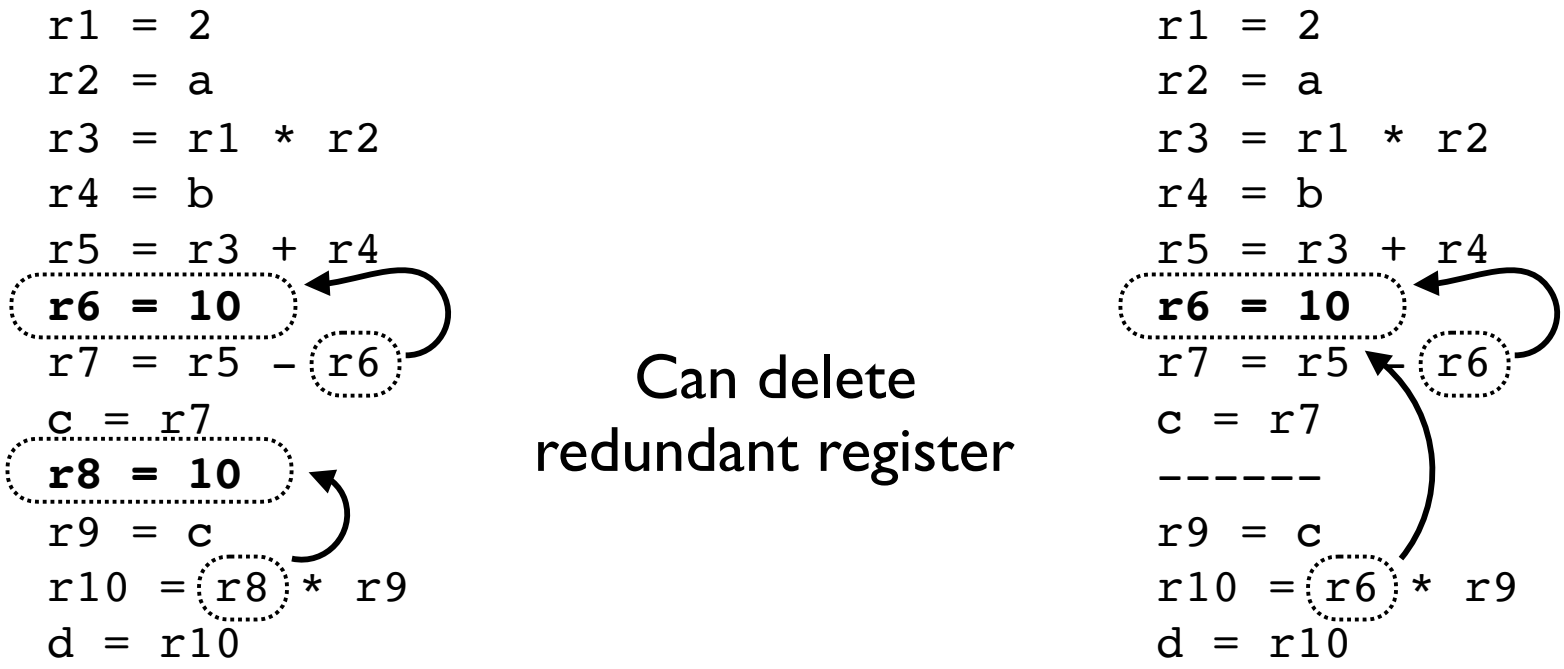
- Example: peephole optimization




- A search for unnecessary instructions

# Optimization

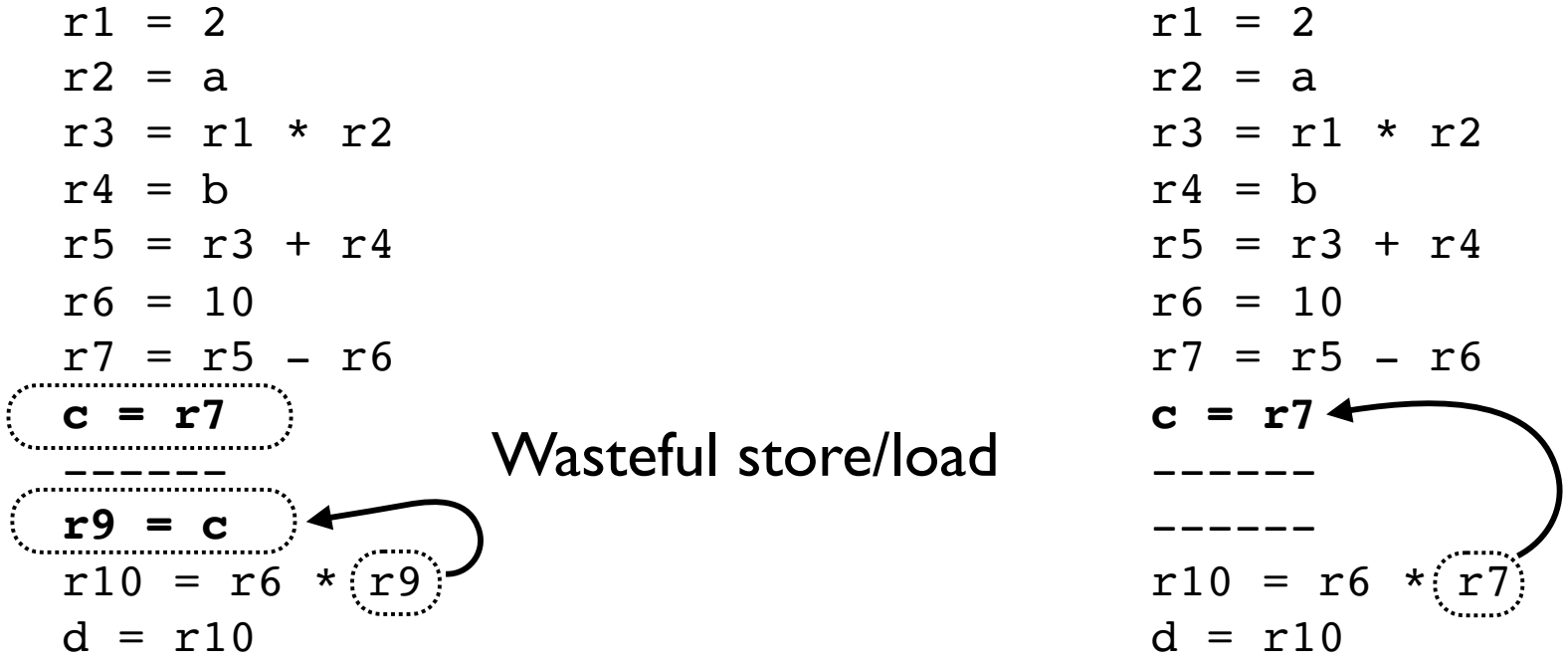
- Example: peephole optimization




- A search for unnecessary instructions

# Optimization

- Example: peephole optimization




- A search for unnecessary instructions

# Optimization

- Example: Subexpression elimination

```
(x+y)/2 + (x+y)/4
```

```
r1 = x
r2 = y
r3 = r1 + r2
r4 = 2
r5 = r3 / r4
r6 = x
r7 = y
r8 = r1 + r2
r9 = 4
r10 = r8 / r9
```

```
r1 = x
r2 = y
r3 = r1 + r2
r4 = 2
r5 = r3 / r4


r9 = 4
r10 = r3 / r9
```

- Only safe if x/y guaranteed not to change

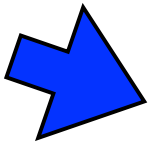
# Packaging of IR code

- Intermediate code is more than instructions
  - Functions
  - Variables
- There is a lot of metadata associated with the instructions (names, types, scopes, etc.)

# Functions

- Instructions are attached to a function
- There is metadata (name, types, etc.)

```
func bar(a int, b int) int {
    var z int;
    z = a * b;
    return z;
}
```



```
function:
name: "bar"
parameters: [int, int]
returns: int
locals: [int]
code:  [
    ('LOAD', 'a', 'r1')
    ('LOAD', 'b', 'r2')
    ('MUL', 'r1', 'r2', 'r3')
    ('STORE', 'r3', 'z')
    ...
    ]
```

- You will make function "objects"

# Modules

- A "module" is the product of compilation
- A container for the compiled declarations

```
function: foo
function: bar
global: x
...
```

- It is helpful to think of Python here. Python code is organized into modules. A module represents a file of source code. Modules contain definitions of functions, variables, classes. Compilers do something similar.

# Project 3

Reading: `docs/ircode.html`

Find the file `wabbit/ircode.py`

Follow the instructions inside

Part 4

# Metal

# Mapping to Hardware

- Question: How does IR code get mapped to real hardware???
- CPUs have registers, memory, etc.
- At some point, something has to generate actual hardware instructions
- Largely about dealing with encodings and HW limitations

# Type Mapping

- Hardware has a very limited set of types

```
64-bit integers
64-bit floats
```

- Everything must map to these types
- Two major issues:
  - Computational Representation
  - Storage Representation

# Computational Repr

- Data must be represented during actual calculations on the CPU itself.
- Example: An 8-bit ASCII character might be represented by 64-bit integer when stored in a CPU register and used in calculation
- CPU doesn't have different sized registers (i.e., they're all 64-bits wide)

# Computational Repr

- Compiler might have to emit extra code to work around hardware limitations
- Example: Performing 64-bit integer math on a CPU with 32-bit registers
- To do it, the compiler would need to break the problem into multiple steps.
- Often a source of "Clever Hacks"

# Storage Representation

- Data might be stored differently in memory
- Example: An 8-bit ASCII character is stored as a single byte in RAM (off CPU).
- Related issue: Byte-ordering

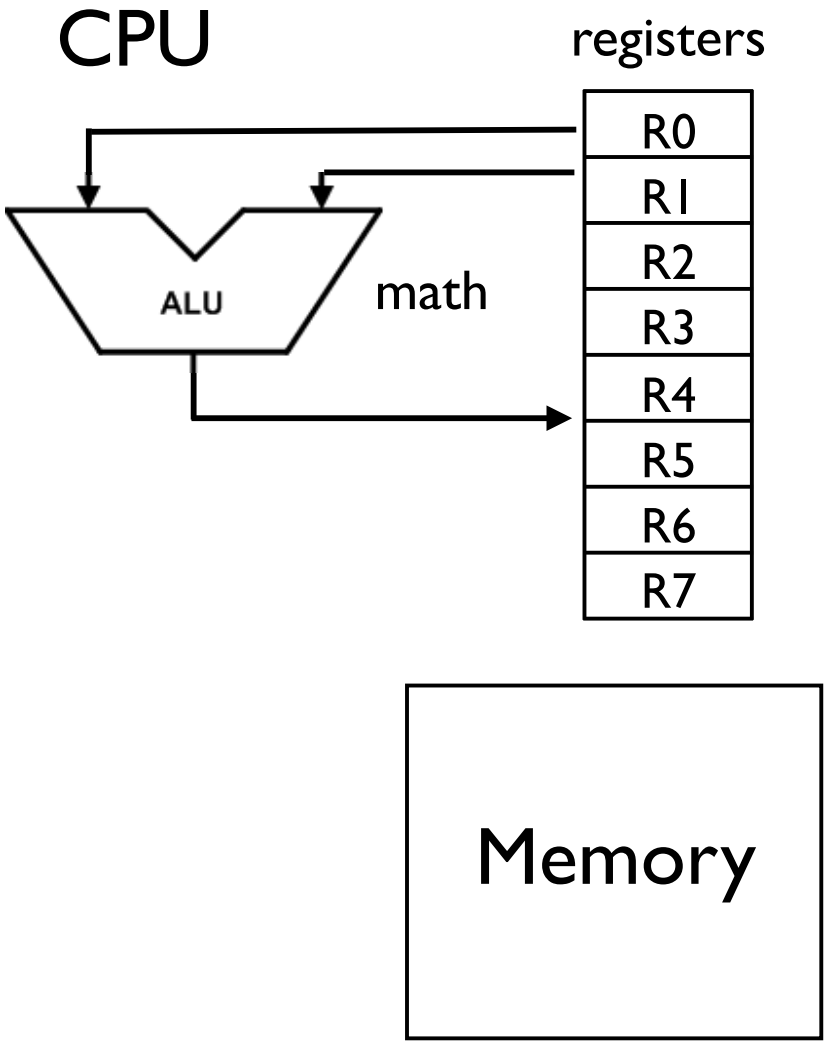
```
0x1234    # -> 0x34 0x12    (little endian)
0x1234    # -> 0x12 0x34    (big endian)
```

- Transparent to end-users, but a compiler might have to worry about it (a lot)

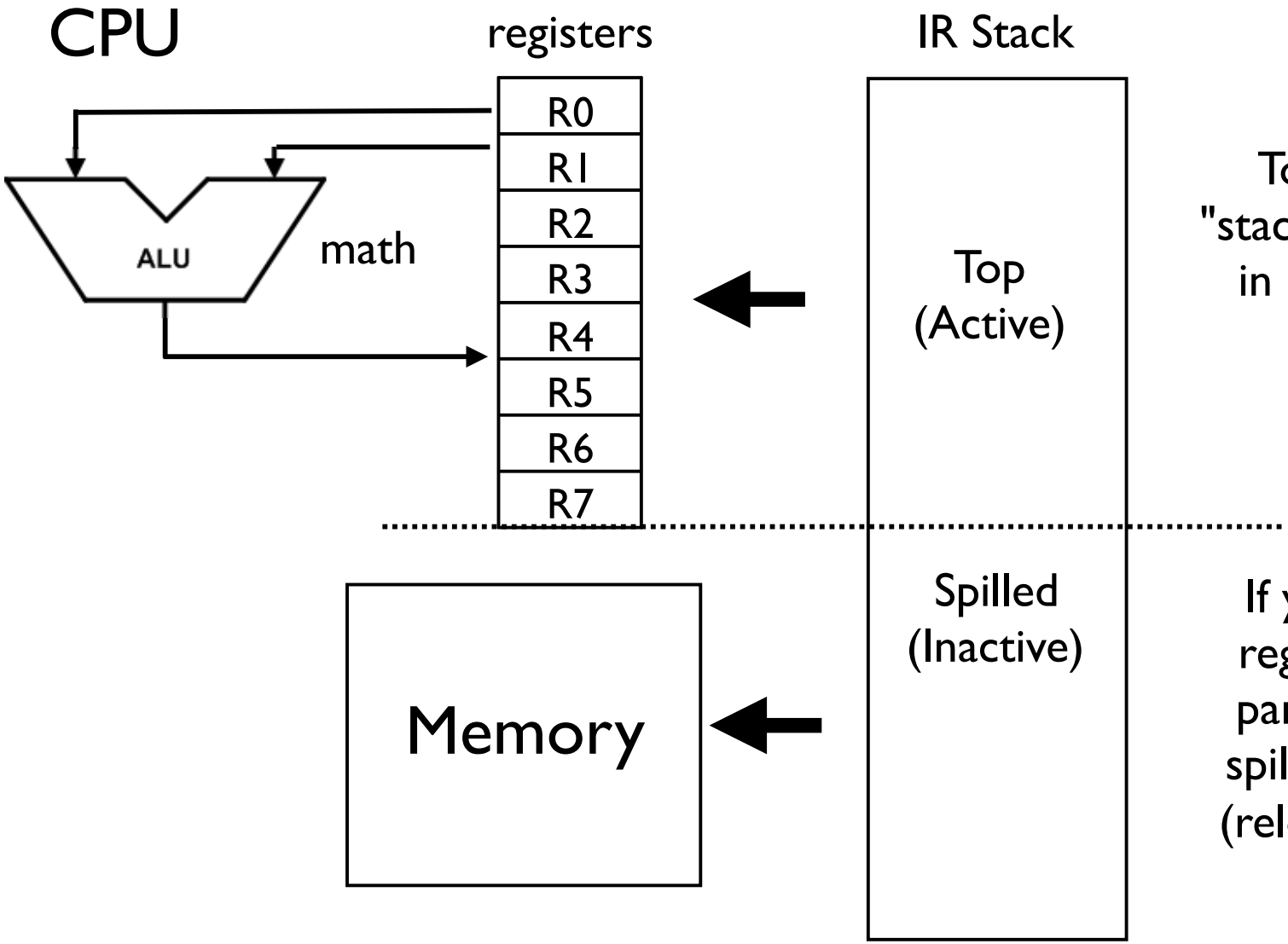
# Stacks -> Registers

- Stack machines are an amazing abstraction
- Don't worry about details of registers
- Or limitations (push/pop all you want)
- Real hardware is usually not a stack machine.

# Mapping to Hardware



# Mapping to Hardware




Top part of the "stack" is always kept in CPU registers.

If you run out of registers, bottom part of stack gets spilled to memory (reload when stack shrinks)

# Mapping to Hardware

```
2 + 3 * (4 + (5 * 6))
```

Suppose only 4 CPU registers

```
PUSH 2
PUSH 3
PUSH 4
PUSH 5
PUSH 6
MUL
ADD
MUL
ADD
```

```
LOADI 2, R0
LOADI 3, R1
LOADI 4, R2
LOADI 5, R3
STORE R0, spill0
LOADI 6, R0
MUL R3, R0, R3
ADD R2, R3, R2
MUL R1, R2, R1
LOAD spill0, R0
ADD R0, R1, R0
```

Stack "wraps around" here. Must save old registers.

# Commentary

- The IR->hardware mapping is really interesting
- Many different CPUs and architectures
- GPUs, SIMD, FPUs, etc.
- Instruction scheduling
- Hardware bug workarounds

# Possible Demo

- Mapping Stack IR to `metal.py`

# Project 4

- Tutorial: `docs/codegen.html`
- Learn how to make an interpreter
- Learn how to make a transpiler
- Learn how to generate LLVM
- Learn how to encode Web Assembly

Part 5

# Control Flow

# Control Flow

- Programming languages have control-flow

```
if a < b:
   ...
else:
   ...

while a < b:
   ...
```

- Introduces branching to the underlying code

# Relations

- First, you need relational operations

```
a < b
a <= b
a > b
a >= b
a == b
a != b
```

- And you need booleans

```
a and b
a or n
not a
```

# Type System (Revisited)

- Relations add new complexity to type system
- Different result type for relations

```
a = 2
b = 3

a < b       # (int < int) -> bool
```

- What is truth?

```
if a:               # Legal or not?
   ...
```

- Both require thought in type system

# Project 5.1

- Reading: `docs/relations.html`
- Add booleans and relational operators
- Make necessary changes to type system
- Make changes to data model and checker
- Modify the code generator

# Basic Blocks

- So far, we have focused on simple statements

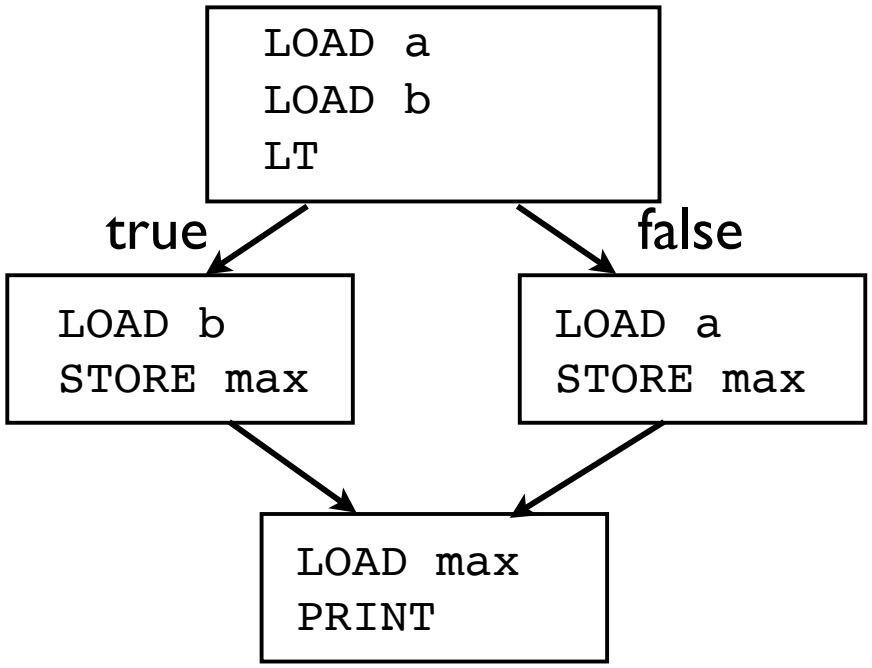
```
var a int = 2;
var b int = 3;
var c int = a + b;
print(2*c);
...
```

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

# Control-Flow

- Control flow statements break code into basic blocks connected in a graph

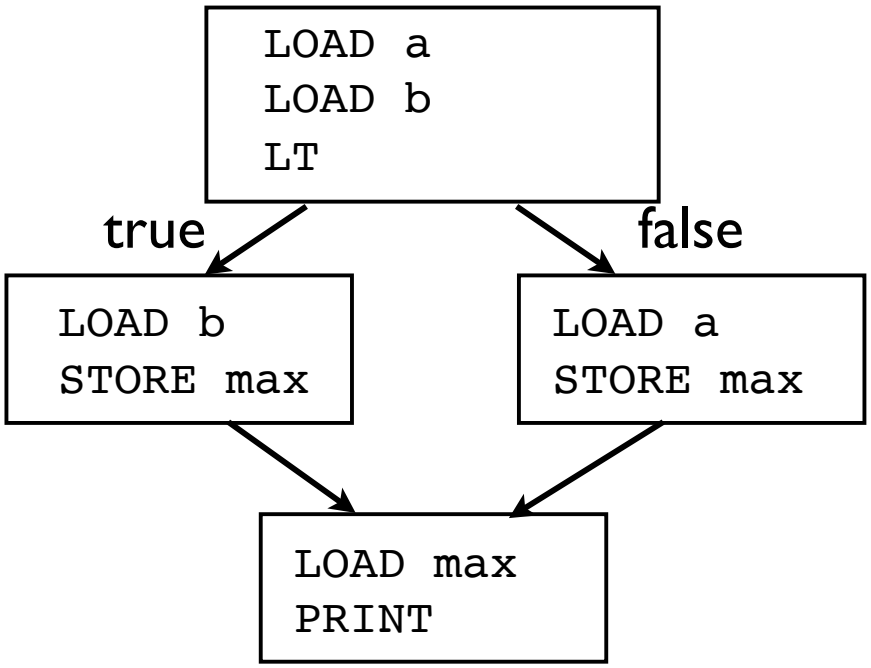
```
var a int = 2;
var b int = 3;
var max int;

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

print max;
```




- Control flow graph

# Problem

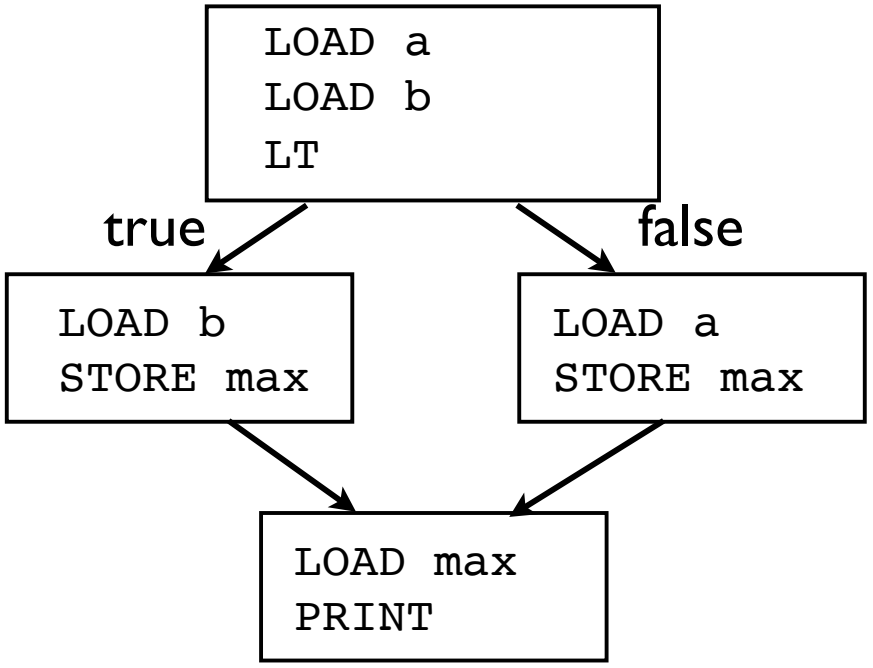
- How do you encode the control-flow graph into intermediate code?




- How is control-flow expressed?

# One Approach: Gotos

- Label each block and emit jump/gotos




```
b1: LOAD a
    LOAD b
    LT
    JMP_TRUE b2
    JMP b3

b2: LOAD b
    STORE max
    JMP b4

b3: LOAD a
    STORE max
    JMP b4

b4: LOAD max
    PRINT
```

# Implementation

- Code generator must emit unique block labels
- Blocks must be linked by jump instructions
- Visit all code branches

# Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
```

current block

...

# Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
```

current block

```
...
LOAD a
LOAD b
LT
```

# Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
```

current block

```
...
LOAD a
LOAD b
LT
```

## Create labels

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

# Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
```

## Emit jump

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

current block

```
...
LOAD a
LOAD b
LT
JMP_TRUE b2
JMP b3
```

# Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
```

Visit "true" branch

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

current block

```
...
LOAD a
LOAD b
LT
JMP_TRUE b2
JMP b3

LABEL b2
statements1
JMP b4
```

# Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
```

Visit "false" branch

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

current block

```
...
LOAD a
LOAD b
LT
JMP_TRUE b2
JMP b3

LABEL b2
statements1
JMP b4

LABEL b3
statements2
JMP b4
```

# Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
```

Create merge block

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

current block

```
...
LOAD a
LOAD b
LT
JMP_TRUE b2
JMP b3

LABEL b2
statements1
JMP b4

LABEL b3
statements2
JMP b4

LABEL b4
...
```

# Commentary

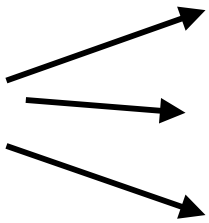
- Emitting jumps and branch statements is how low-level CPUs work
- Can result in a kind of low-level "spaghetti code" that is very hard for humans to follow
- It's not about the "style-guide" at this level

# Structured Control Flow

- Alternative: Use structured code blocks in IR

```
if a < b {
   statements1
} else {
   statements2
}
```

Create instructions to express the block-structure of the conditional



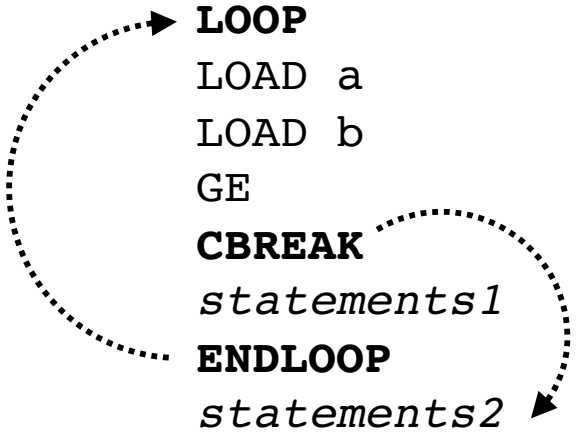
current block

```
...
LOAD a
LOAD b
LT
IF
statements1
ELSE
statements2
ENDIF
...
```

# Structured Control Flow

- Example of a loop

```
while a < b {
    statements1
}
statements2
```




- Looks a bit funny, but it's same idea as this

```
while True {
    if not (a < b) {
        break
    }
    statements1
}
statements2
```

# Commentary

- Both approaches are used
- LLVM: Blocks linked by gotos/branches
- WebAssembly: Structured control flow
- Our compiler: Structured control flow (it's a bit easier to manage and can be mapped to lower-level constructs)

# Project 5.2

- Reading: `docs/controlflow.html`
- Add control-flow constructs to the model
- Modify type-checker as appropriate
- Modify code generator to emit structured control flow primitives.
- Make control-flow work in targets

Part 6

# Functions

# Functions

- Programming languages let you define functions

```
def add(x,y):
    return x+y

def countdown(n):
    while n > 0:
        print("T-minus",n)
        n -= 1
    print("Boom!")
```

- Two problems:
  - Scoping of identifiers
  - Runtime implementation

# Function Scoping

- Most languages use lexical scoping
- Pertains to visibility of identifiers

```
a = 13
def foo():
    b = 42
    print(a,b)          # a,b are visible

def bar():
    c = 13
    print(a,b)          # a,c are visible
                        # b is not visible
```

- Identifiers defined in enclosing source code context of a particular statement are visible

# Python Scoping

- Python uses two-level scoping
  - Global scope (module-level)
  - Local scope (function bodies)

```
a = 13                  # Global
def foo():
    b = 42              # Local
    print(a,b)
```

# Block Scoping

- Some languages use block scoping (e.g., C)

```
int a = 1;                  / Global
int foo() {
    int n = 0;              / Local. Visible in entire func
    while (n < 10) {
        int x = 2;          / Block. Visible only in 'while'
        ...
    }
    printf("%d\n",x); / Error. x not defined
}
```

- Not in Python though...

# Function Runtime

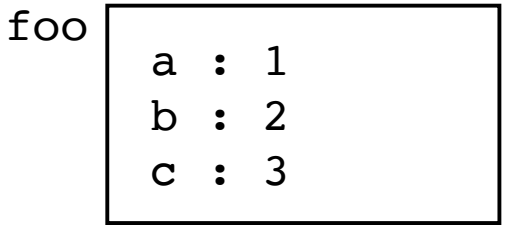
- Each invocation of a function creates a new environment of local variables
- Known as an activation frame (or record)
- Activation frames make up the call stack

# Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```

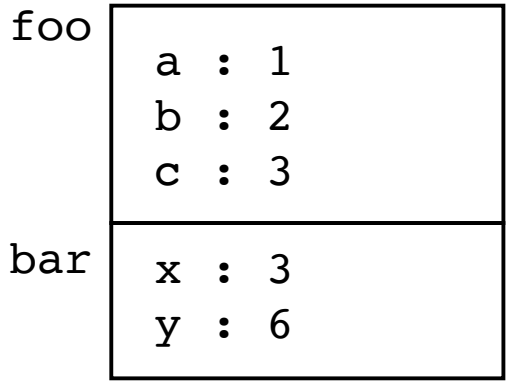
# Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



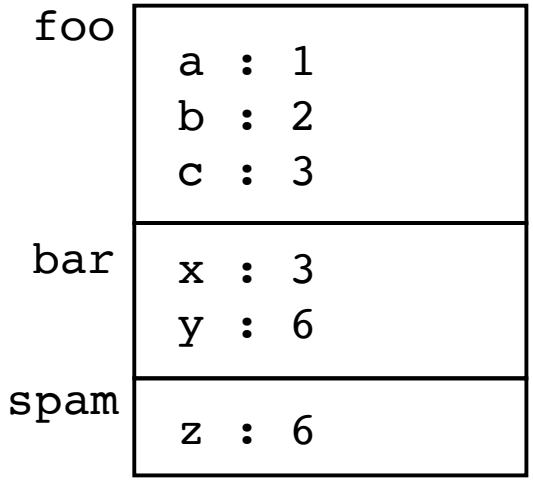
# Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



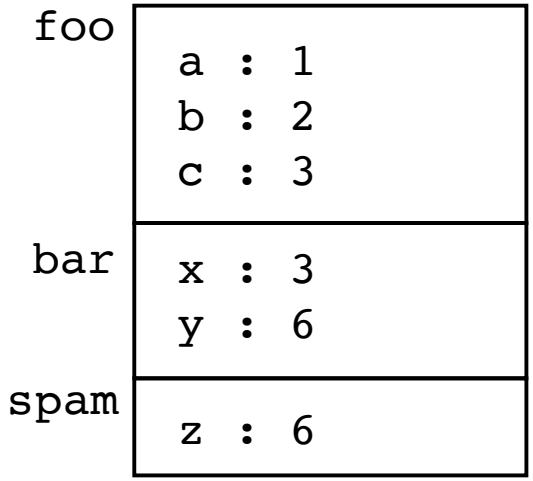
# Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



# Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```




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

# Activation Frames

- You see frames in tracebacks

```
File "expr.py", line 20, in <module>
  exprcheck.check_program(program)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 410, in check_program
  checker.visit(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 163, in visit_Program
  self.visit(node.statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
  self.visit(item)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 350, in visit_FuncDeclar
  self.visit(node.statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
  self.visit(item)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 303, in visit_IfStatemer
  self.visit(node.if_statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
  self.visit(item)
```

# Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)          (caller)

def foo(x,y):              (callee)
    z = x + y
    return z
```

# Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```


```
x : 1
y : 2
return : None
```

Caller is responsible for creating a new frame and populating it with input arguments.

# Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```


```
x : 1
y : 2
return : None
```

Semantic Issue: What does the frame contain?

Copies of the arguments? (Pass by value)

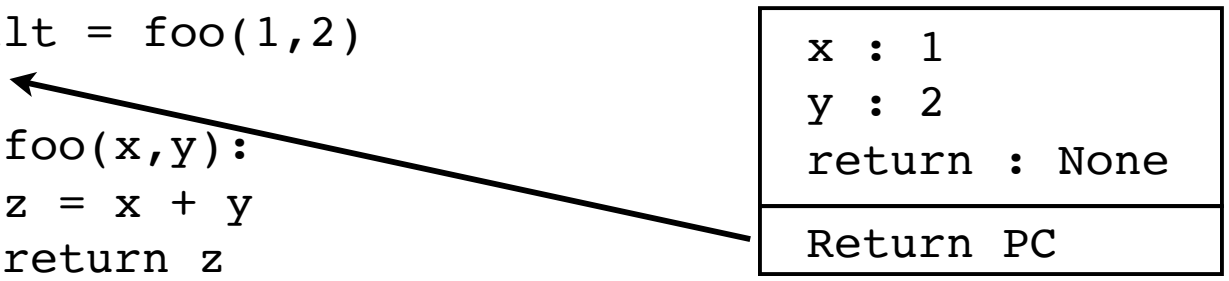
Pointers to the arguments? (Pass by reference)

Depends on the language

# Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

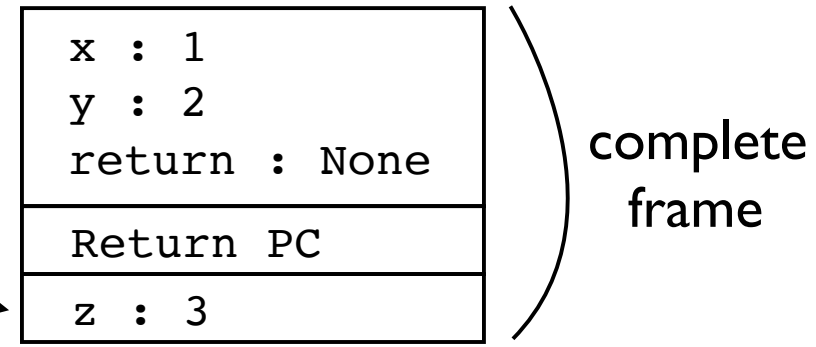



Return address (PC) recorded in the frame (so you can get back to the caller upon return)

# Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

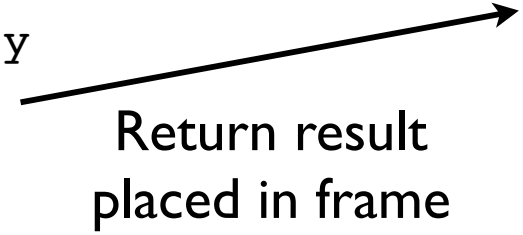



Local variables get added to the frame by the callee

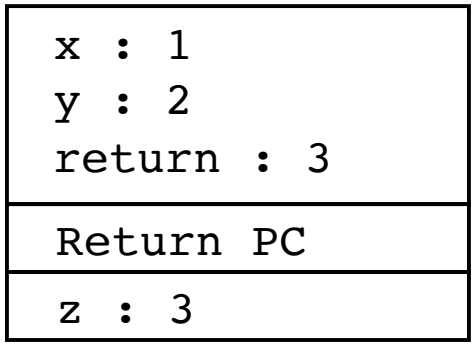
# Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

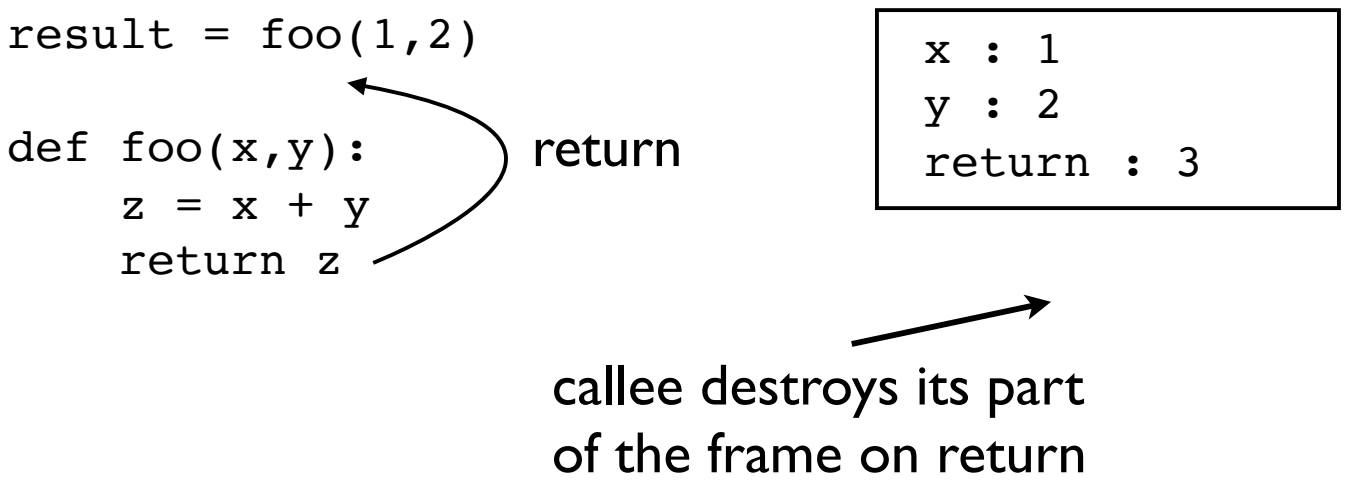


Return result placed in frame



# Frame Management

- Management of Activation Frames is managed by both the caller and callee



# Frame Management

- Management of Activation Frames is managed by both the caller and callee

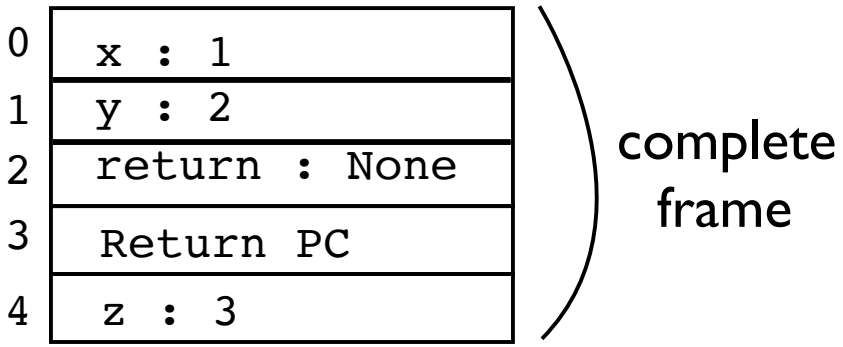
```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

caller destroys remaining frame on assignment of result

# Frame Management

- Implementation Detail : Frame often organized as an array of numeric "slots"

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```



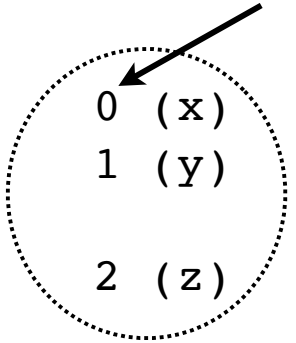

- Slot numbers used in low-level instructions
- Determined at compile-time

# Frame Example

- Python Disassembly

```
def foo(x,y):
    z = x + y
    return z

>>> import dis
>>> dis.dis(foo)
  2           0 LOAD_FAST                0 (x)
              3 LOAD_FAST                1 (y)
              6 BINARY_ADD
              7 STORE_FAST               2 (z)

  3          10 LOAD_FAST                2 (z)
             13 RETURN_VALUE
>>>
```

numbers refer to "slots" in the activation frame



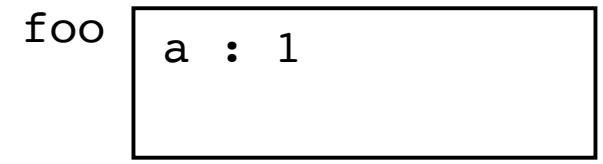
# ABIs

- Application Binary Interface
- A precise specification of function/procedure call semantics related to activation frames
- Language agnostic
- Critical part of creating programming libraries, DLLs, modules, etc.
- Different than an API (higher level)

# Tail Call Optimization

- Sometimes the compiler can eliminate frames

```
def foo(a):
    ...
    return bar(a-1)

def bar(a):
    ...
    return result

foo(1)
```

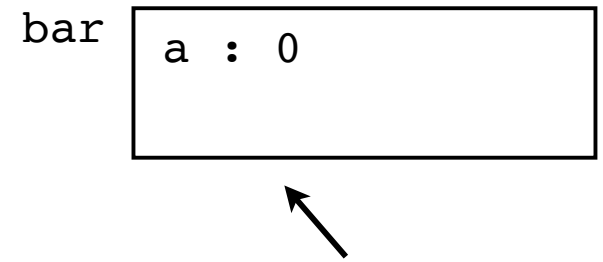



compiler detects that no more statements follow

# Tail Call Optimization

- Sometimes the compiler can eliminate frames

```
def foo(a):
    ...
    return bar(a-1)

def bar(a):
    ...
    return result

foo(1)
```




compiler reuses the same stack frame and just jumps to the next procedure (goto)

- Note: Python does not do this (although people often wish that it did)

# Closures

- Nested functions are "interesting"

```
def add(x):
    def f(y):
        return x + y
    return f
```

- Example:

```
>>> a = add(2)
>>> a(3)
5
>>>
```

- The "x" variable must live someplace
- It does not exist on the stack.

# Closures

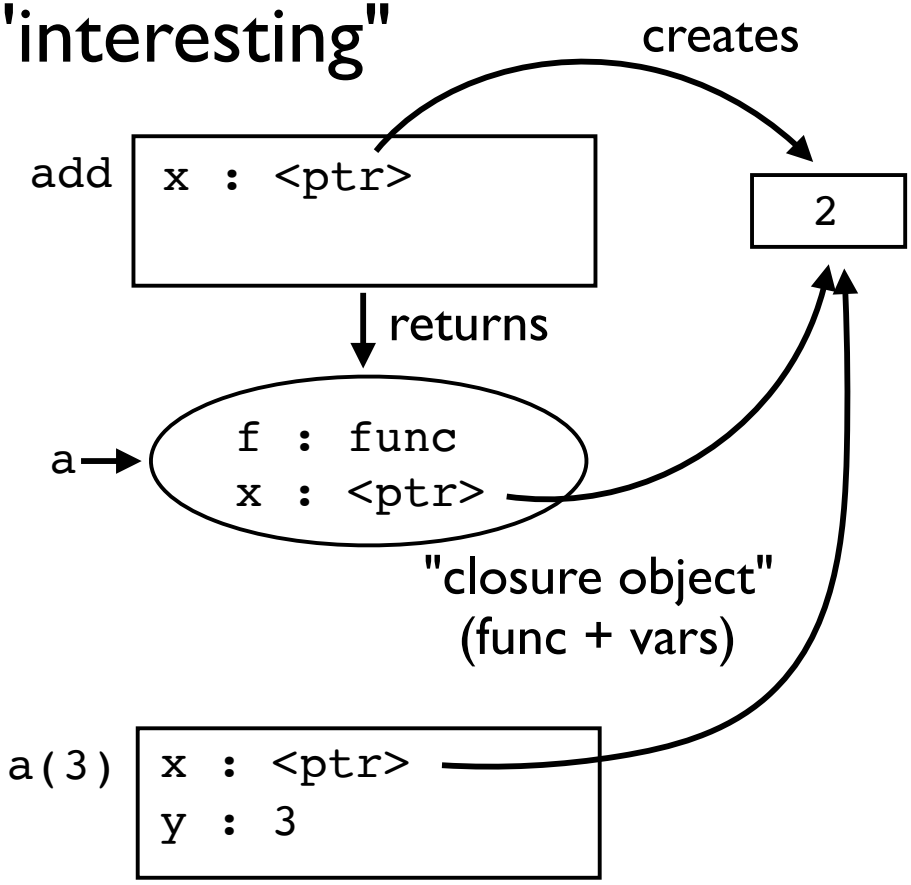
- Nested functions are "interesting"

```
def add(x):
    def f(y):
        return x + y
    return f
```

- Example:

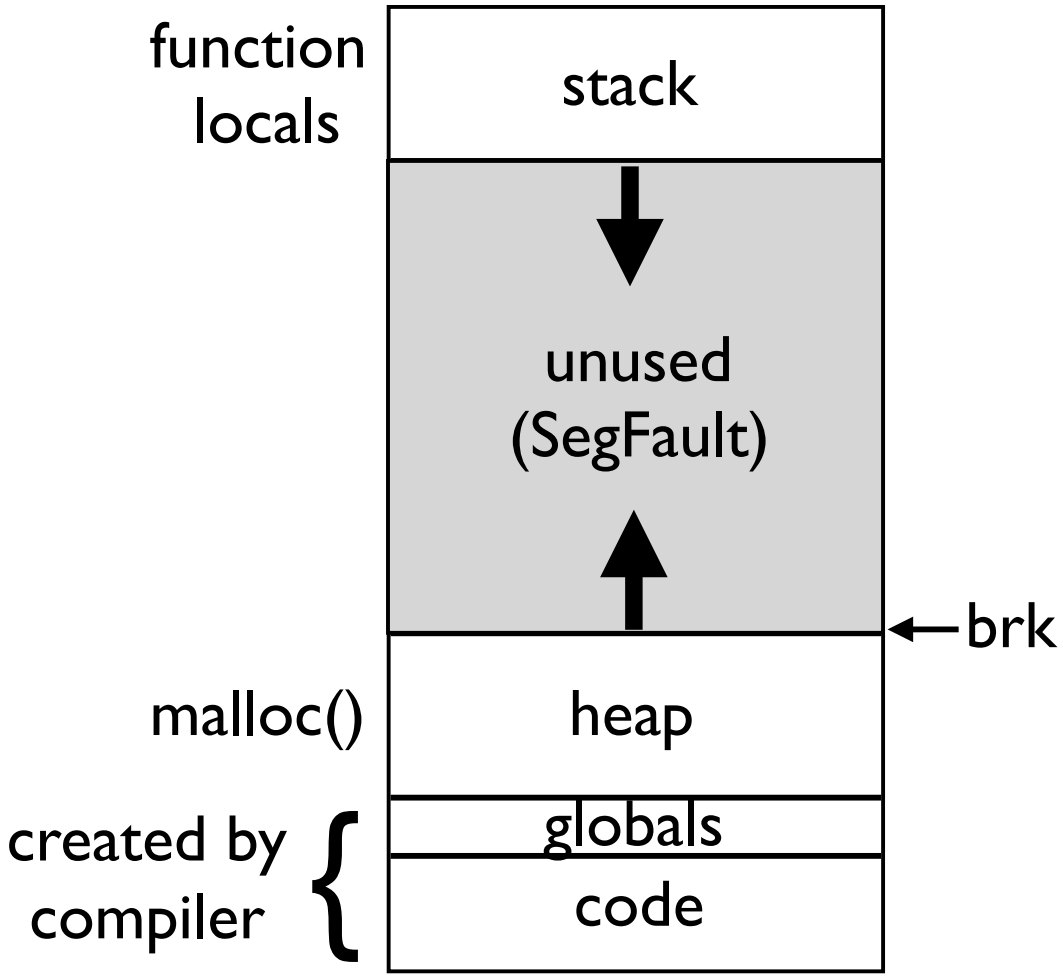
```
>>> a = add(2)
>>> a(3)
5
>>>
```

- Indirect reference to a value stored "off stack"



# Memory Management

- Runtime memory layout
- Managed by the code emitted from the compiler and operating system.
- Related: Garbage Collection



# Program Startup

- Most programs have an entry point
- Often called main()
- Must be written by the user

# Program Startup

- Compiler generates a hidden startup/ initialization function that calls main()

```
func main() int {
    // Written by the programmer
    ...
    return 0;
}

func __start() int {
    // Initialization (created by compiler)
    ...
    return main();
}
```

- Primary purpose is to initialize globals

# Program Startup

- Initialization example:

```
var x int = v1;
var y int = v2;
...
func main() int {
    // Written by the programmer
    ...
    return 0;
}

func __start() int {
    // Initialization (created by compiler)
    x = v1;     // Setting of global variables
    y = v2;
    return main();
}
```

# Compilation Steps

- Compiler processes each function, one at a time and creates basic blocks of IR code
- Compiler tracks global initialization steps
- Automatically create special startup/init function upon completion of all other code
- Final result is a collection of functions

# Project 6

Reading: docs/functions.html

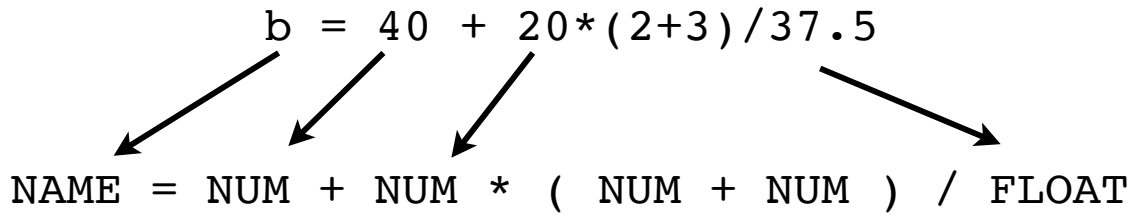
Coding: Modify IR-code and other parts of compiler to support function definitions and function call.

Part 7

# Lexing

# Lexing in a Nutshell

- Convert input text into a token stream



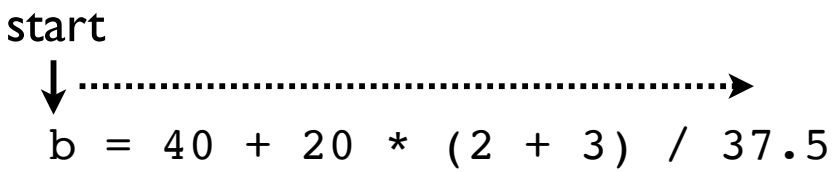

- Tokens have both a type and value

```
b    ------>  ('NAME','b')

=    ------>  ('ASSIGN','=')

40   ------>  ('NUM','40')
```

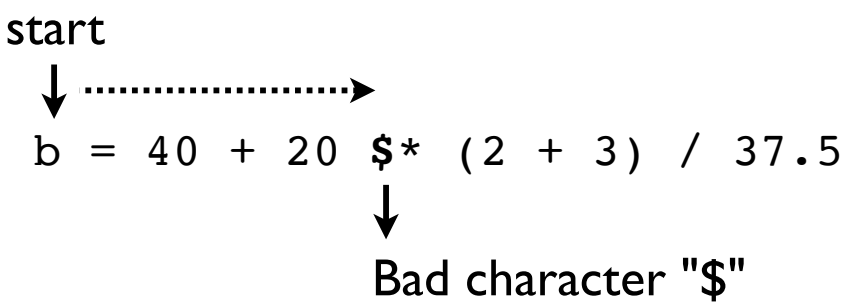
- Question: How to do it?

# Text Scanning

- Must perform a linear text scan

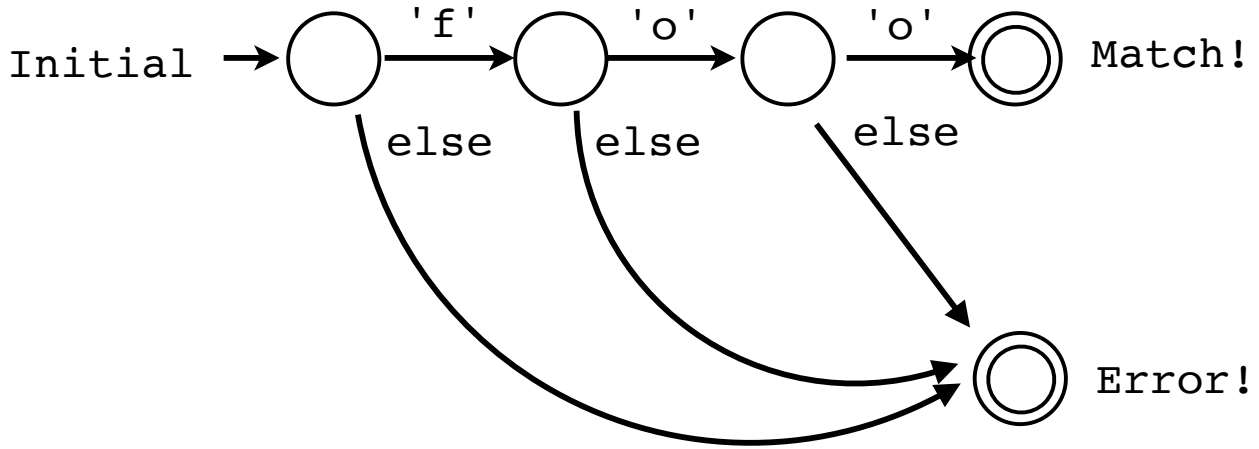



- ALL characters must be consumed
- Otherwise error:



# Text Matching

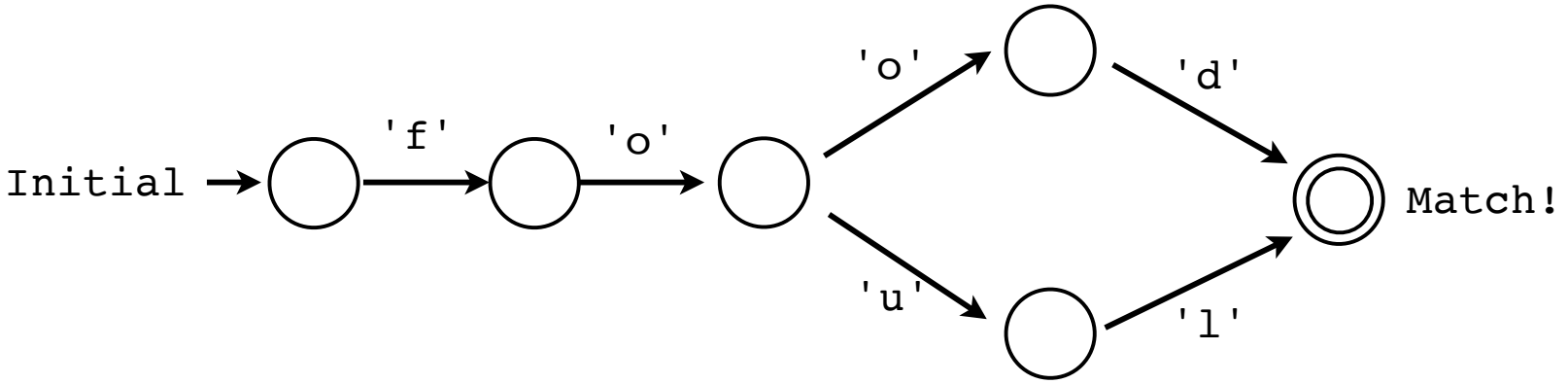
- Characters are processed using a state machine
- Example:  Match the text "foo"




You work one character at a time through various "states." You either end up with a match or an error.

# Matching Alternatives

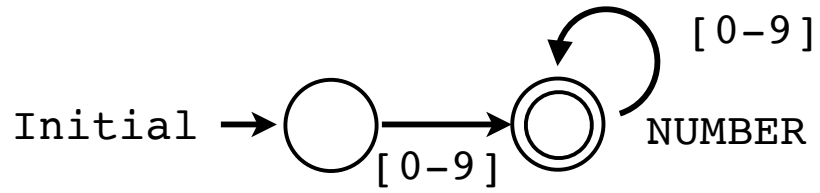
- Example: Match the text "food" or "foul




You don't go backwards. Forward progress only. You might take one or more branches along the way.

# Cycles/Repetition

- Example: Match numbers [0-9]+

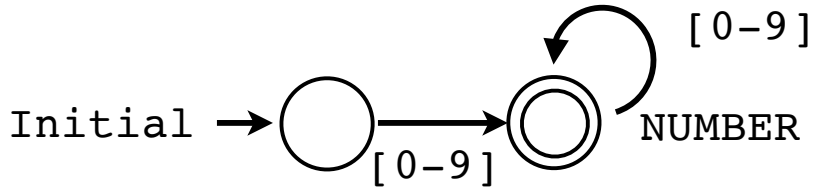



You stay in a matched state to collect additional characters

- Notation:
  - {pat}* -> Zero or more
  - {pat}+ -> One or more

# Coding Example

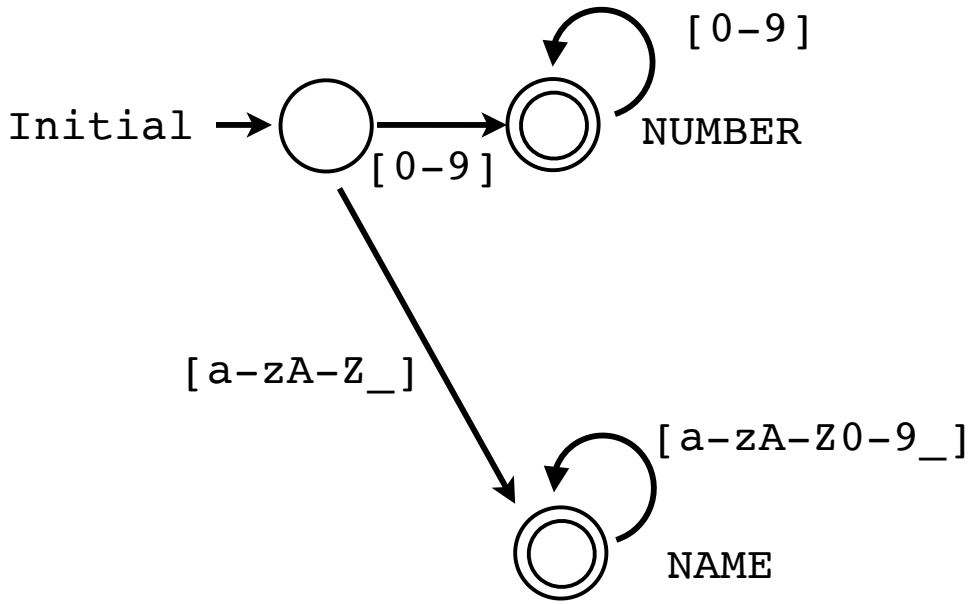
- Example: Match numbers [0-9]+




- Encode this state machine into a function that finds and prints all numbers in a text string

# Coding Example

- Find all numbers [0-9]+ and
- Find all variable names [a-zA-Z_][a-zA-Z0-9_]*



# Commentary

- Tokens are formally described by regex

```
NAME   = r'[A-Za-z_][A-Za-z0-9_]*'
NUMBER = r'\d+'
```

- This is basis of the re module
- Writing lexing code is so tedious that tools are often used to automate the process (e.g., SLY, PLY, PyParsing, etc.).

# Deeper Thought

- What is the computational nature of lexing?
- Do you need to do anything "fancy?"
  - Recursion?
  - Extra data structures? (trees, stacks, etc.)
- Short answer: Not really.

# Project 7

- Find the file `wabbit/tokenize.py`
- Follow instructions inside

Part 8

# Parsing

# The Parsing Problem

- Recognize syntactically correct input

```
b = 40 + 20*(2+3)        # YES!
c = 40 + * 20            # NO!
d = 40 + + 20            # ???
```

- Need to transform this input into the structural representation of the program
- Text -> Data model

# Disclaimer

- Parsing theory is a huge topic
- Highly mathematical
- Covered in excruciating detail the first 3-5 weeks of a compilers course
- I'm going to cover the highlights and try to motivate some of the practical problems

# Problem: Specification

- How do you even describe syntax?
- Example: Describe Python "assignment"

```
a = 0
b = 2 + 3
c.name = 2 + 3 * 4
d[1] = (2 + 3) * 4
e['key'] = 0.5 * d
```

- By "describe"--a precise specification
- By "precise"--rigorous like math/coding

# Problem: Specification

- Example: Describe "assignment"

```
location = expression
```

- That is extremely high-level (vague)
  - What is a "location"?
  - What is an "expression"?
- How do you describe it terms of tokens?

# Grammar Specification

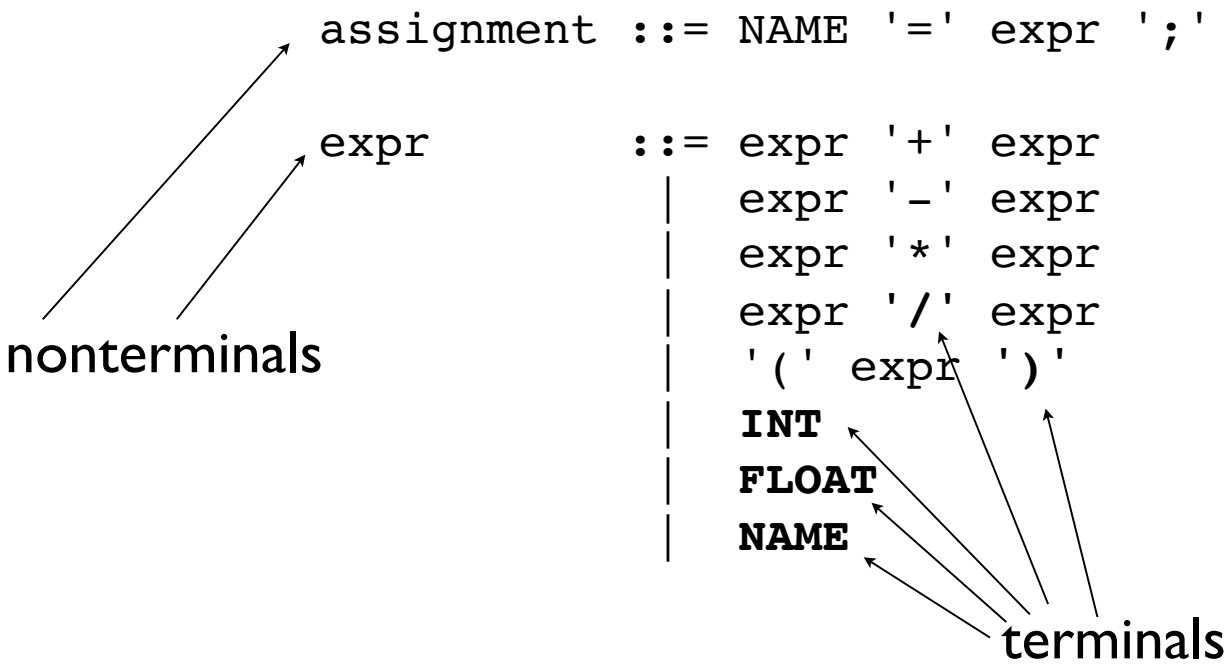
- Syntax often specified as a Context Free Grammar

```
assignment ::= NAME '=' expr ';'

expr       ::= expr '+' expr
             |  expr '-' expr
             |  expr '*' expr
             |  expr '/' expr
             |  '(' expr ')'
             |  INT
             |  FLOAT
             |  NAME
```

- Notation known as BNF (Backus Naur Form)

# Terminals/Nonterminals

- Tokens are called "terminals"
- Rule names are called "nonterminals"



# Terminology

- "terminal" - A symbol that can't be expanded into anything else (tokens).
- "nonterminal" - A symbol that can be expanded into other symbols (grammar rules)

# Grammar Specification

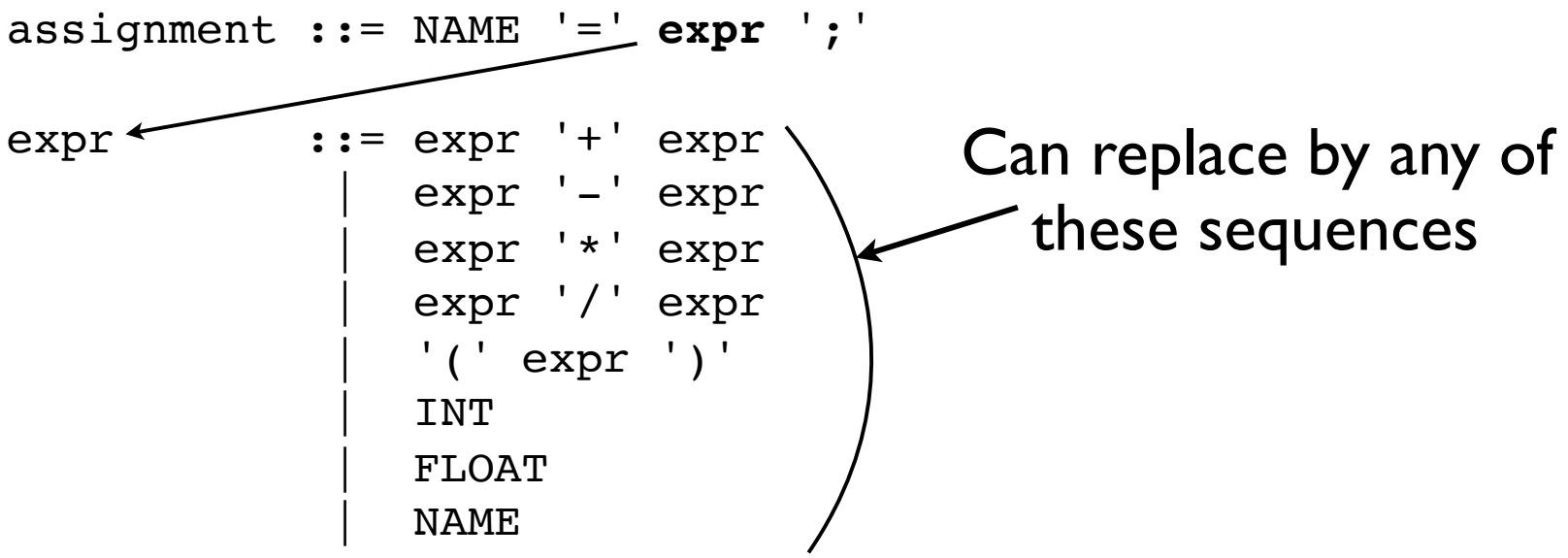
- A BNF specifies substitutions

```
assignment ::= NAME '=' expr ';'

expr       ::= expr '+' expr
             |  expr '-' expr
             |  expr '*' expr
             |  expr '/' expr
             |  '(' expr ')'
             |  INT
             |  FLOAT
             |  NAME
```

- Name on left can be replaced the sequence of symbols on the right (and vice versa).

# Grammar Specification

- A BNF specifies substitutions




- Name on left can be replaced the sequence of symbols on the right (and vice versa).

# Grammar Specification

- A BNF specifies substitutions

```
assignment ::= NAME '=' expr ';'

expr       ::= expr '+' expr
             |  expr '-' expr
             |  expr '*' expr
             |  expr '/' expr
             |  '(' expr ')'
             |  INT
             |  FLOAT
             |  NAME
```

**Examples**

```
spam = 42;
spam = 4+2;
spam = (4+2)*3
```

- Name on left can be replaced the sequence of symbols on the right (and vice versa).

# Grammar Specification

- Substitutions are recursive

```
expr        ::= expr '+' expr
              |  expr '-' expr
              |  expr '*' expr
              |  expr '/' expr
              |  '(' expr ')'
              |  INT
              |  FLOAT
              |  NAME
```

- Can self-expand as needed (off to infinity...)

```
expr
expr + expr
expr + expr + expr
expr + expr + expr * expr
expr + expr + expr * expr - expr
```

# Problem: Ambiguity

- Consider:

```
expr                        # Expand
expr + expr                 # Expand
expr + expr + expr          # Expand (which one?)
```

- Was it the left expression?

```
expr + expr    --->  (expr + expr) + expr
```

- Or the right expression?

```
expr + expr    --->  expr + (expr + expr)
```

- Why you care: associativity of operators

# Associativity

- Programming languages have rules about order

```
1 + 2 + 3 + 4 + 5
```

- Left associativity (left-to-right)

```
(((1 + 2) + 3) + 4) + 5
```

- Right associativity (right-to-left)

```
1 + (2 + (3 + (4 + 5)))
```

- Does it matter? Yes.

# Associativity

- A tricky example

```
1 - 3 - 4
```

- Left associativity  (left-to-right)

```
(1 - 3) - 4  --> -6
```

- Right associativity (right-to-left)

```
1 - (3 - 4)  --> 2
```

- Q: Can this be encoded in the grammar?

# Associativity

- Expression grammar with left associativity

```
expr ::= expr + term
       |  expr - term
       |  expr * term
       |  expr / term
       |  term

term ::= INT
       |  FLOAT
       |  NAME
       |  ( expr )
```

- Idea: The recursive expansion of expressions is forced onto the left-hand side.

# Problem: Precedence

- Consider:

```
1 + 2 * 3 + 4
```

- Is this to be parsed as follows?

```
((1 + 2) * 3) + 4
```

- No, assuming the rules of math class
- It should be this (order of evaluation)

```
(1 + (2 * 3)) + 4
```

- Q: Can this also be encoded in the grammar?

# Precedence

- Expression grammar with precedence levels

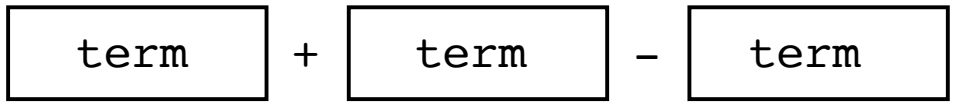
```
expr ::= expr + term
       |  expr - term
       |  term

term ::= term * factor
       |  term / factor
       |  factor

factor ::= INT
       |  FLOAT
       |  NAME
       |  ( expr )
```

- Idea: Layering from low->high precedence

# Precedence

- Expression grammar with precedence levels

```
expr ::= expr + term
       |  expr - term
       |  term

term ::= term * factor
       |  term / factor
       |  factor

factor ::= INT
       |  FLOAT
       |  NAME
       |  ( expr )
```

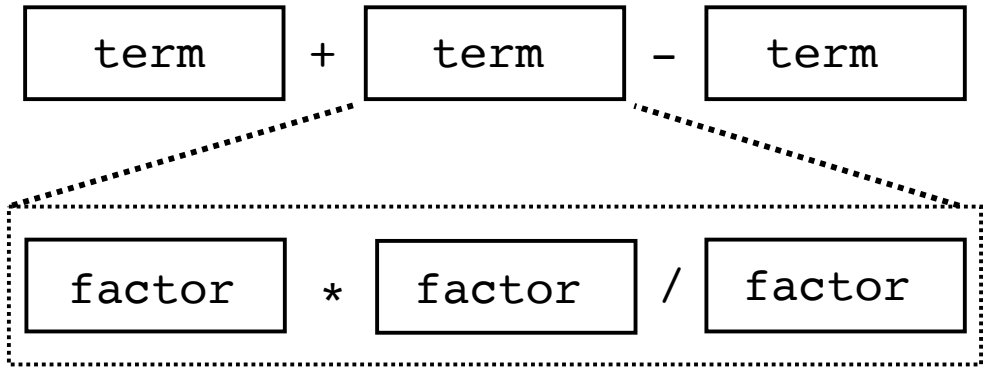



- Idea: Layering from low->high precedence

# Precedence

- Expression grammar with precedence levels

```
expr ::= expr + term
       |  expr - term
       |  term

term ::= term * factor
       |  term / factor
       |  factor

factor ::= INT
       |  FLOAT
       |  NAME
       |  ( expr )
```

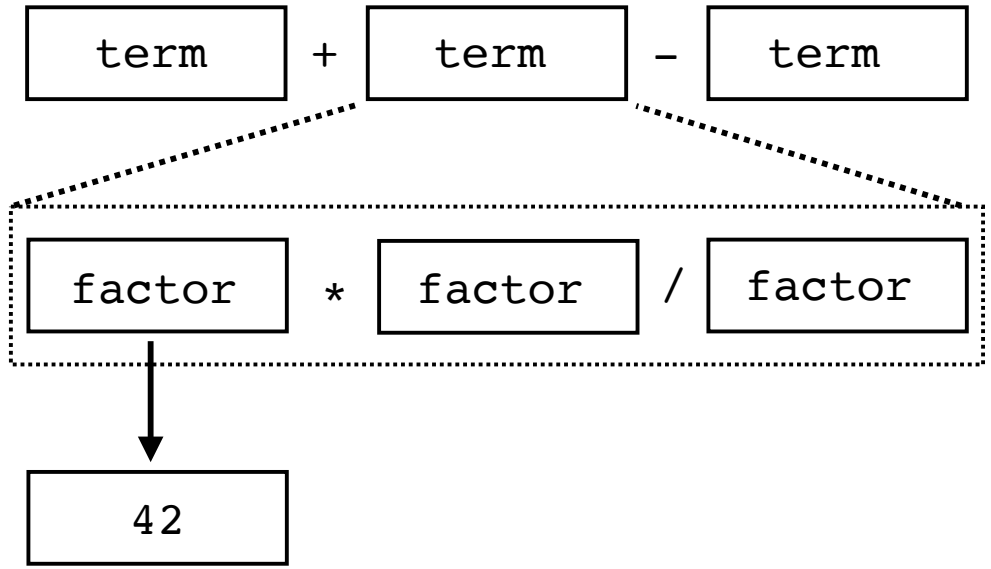



- Idea: Layering from low->high precedence

# Precedence

- Expression grammar with precedence levels

```
expr ::= expr + term
       |  expr - term
       |  term

term ::= term * factor
       |  term / factor
       |  factor

factor ::= INT
       |  FLOAT
       |  NAME
       |  ( expr )
```




- Idea: Layering from low->high precedence

# Precedence

- There may be many more levels

```
a + b < c + d and e * f > h or i == j
```

```
(a + b < c + d and e * f > h) or (i == j)
```

```
((a + b < c + d) and (e * f > h)) or (i == j)
```

```
(((a + b) < (c + d)) and ((e * f) > h)) or (i == j)
```

- All of this can be encoded in the grammar
- Need a separate rule for each precedence layer

# Notational Simplification

- What is *actually* being expressed by this rule?

```
expr ::= expr + term
       |  expr - term
       |  term
```

- Repetition (of terms).
- Alternative notation: EBNF

```
expr = term { "+"|"-" term }
```

- Notational guide

```
a | b | c          # Alternatives
{ ... }            # Repetition (0 or more)
[ ... ]            # Optional (0 or 1)
```

# EBNF Example

- Grammar as a EBNF

```
assignment = NAME '=' expr ';'
expr = term { '+'|'-' term }
term = factor { '*'|'/' factor }
factor = INTEGER | FLOAT | NAME | '(' expr ')'
```

- EBNF is a fairly common standard for grammar specification
- You see it a lot in standards documents
- Mini exercise: Look at Python grammar

# Parsing Explained

- Problem: match input text against a grammar

```
a = 2 * 3 + 4;
```

- Example: Does it match the assignment rule?

```
assignment ::= NAME '=' expr ';'
```

- How would you go about doing that?
- Specifically:  Can you make a concrete algorithm?

# Parsing Algorithms

*"Why did the parser cross the road?"*

# Parsing Algorithms

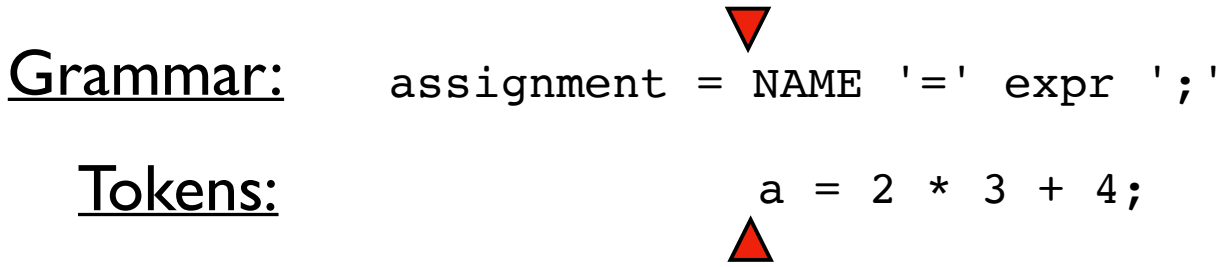
*"Why did the parser cross the road?"*

*"To get to the other side."*

- This a surprisingly accurate description of parsing ("getting to the other side").
- Let's elaborate further...

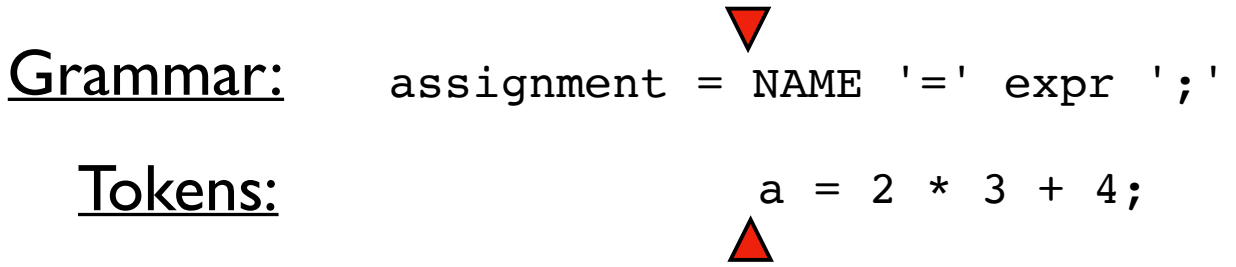
# Parsing Algorithms

- In the beginning, you know nothing...

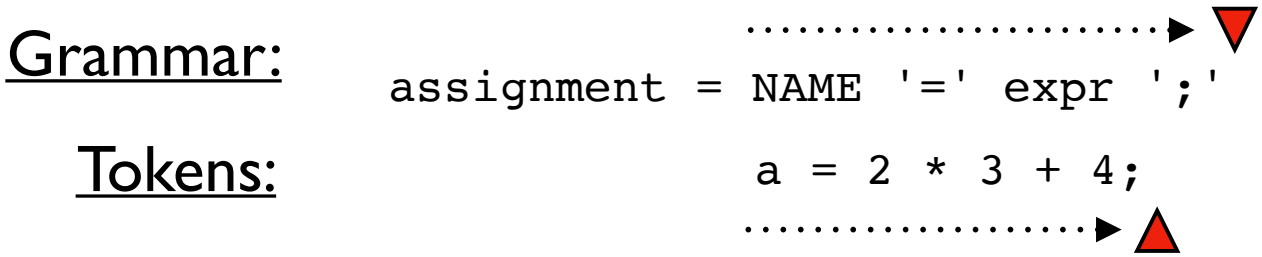


# Parsing Algorithms

- In the beginning, you know nothing...

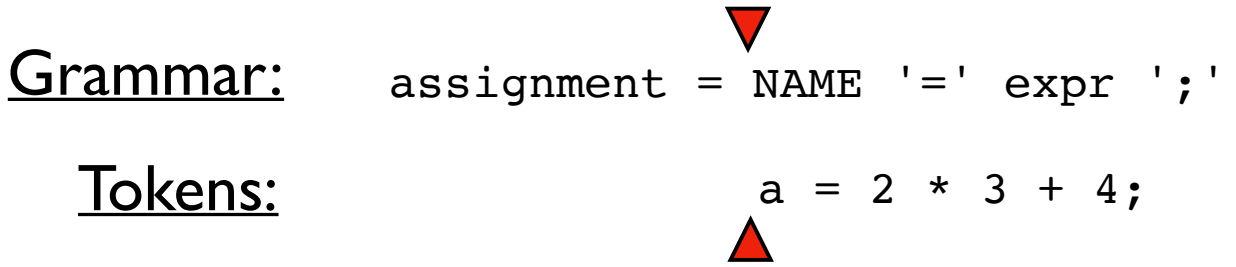



- The goal: move both markers to the other side

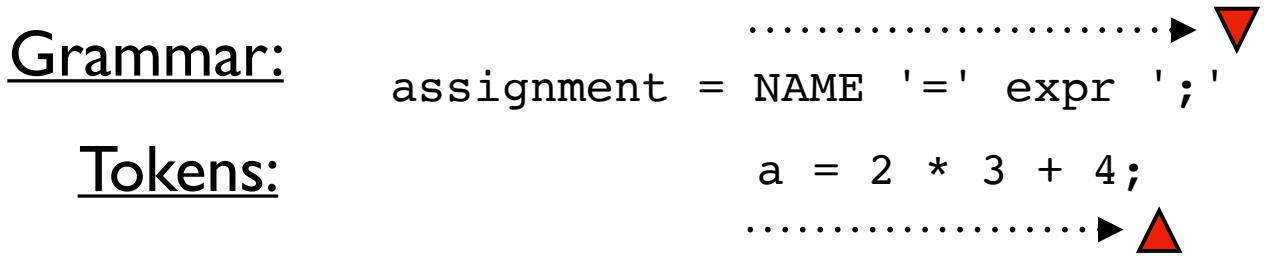


# Parsing Algorithms

- In the beginning, you know nothing...



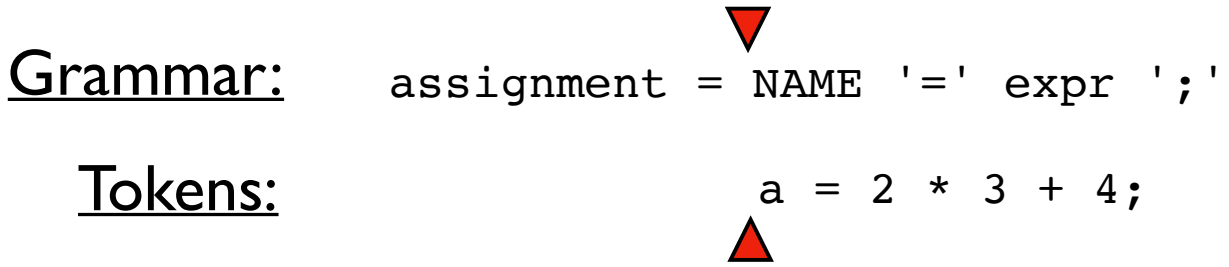

- The goal: move both markers to the other side




- But, can only follow the grammar rules

# Parsing Illustrated

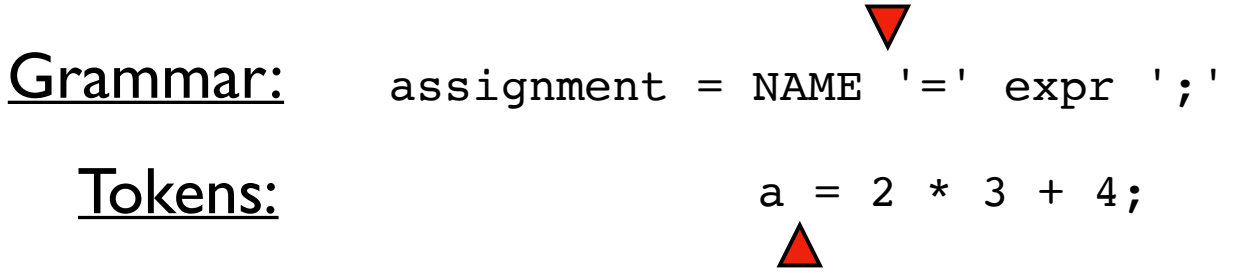
- Parsing involves stepping through tokens




- You try to match to grammar as you go

# Parsing Illustrated

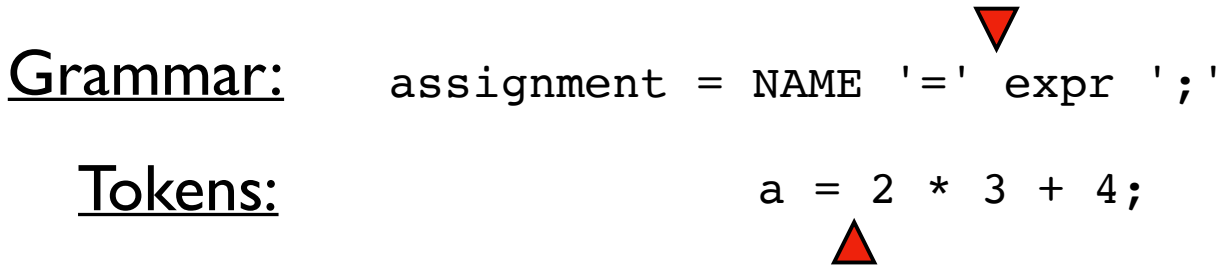
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Forward progress if there is a token match

# Parsing Illustrated

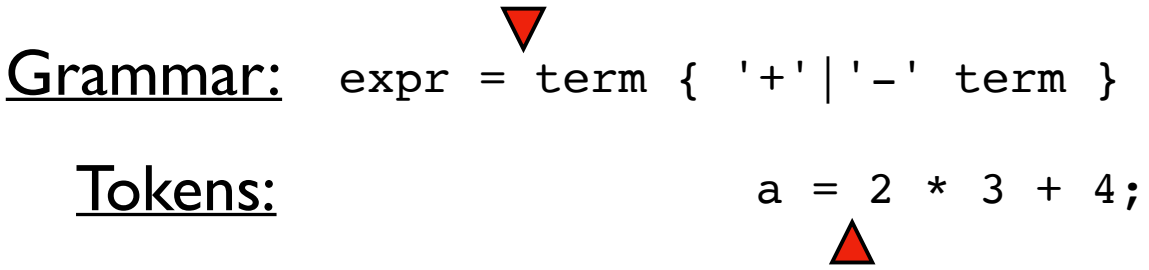
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Forward progress if there is a token match

# Parsing Illustrated

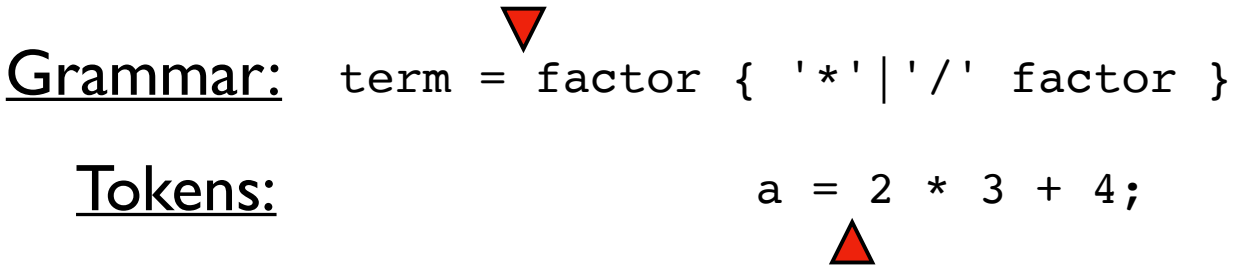
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules

# Parsing Illustrated

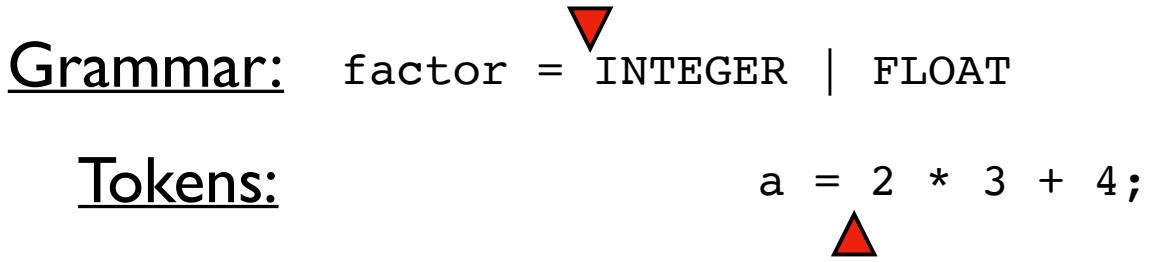
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules

# Parsing Illustrated

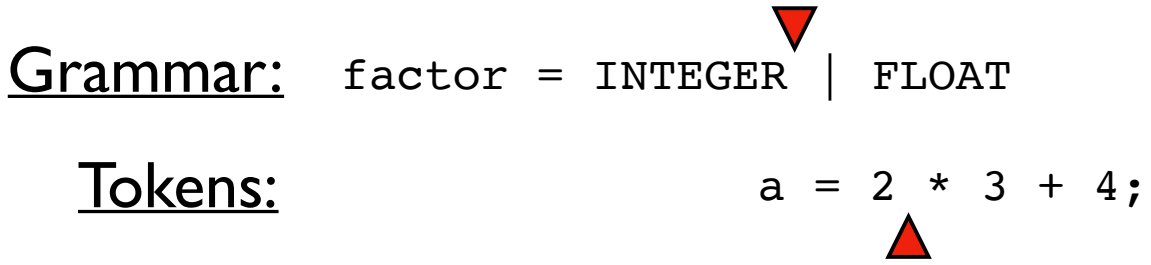
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules

# Parsing Illustrated

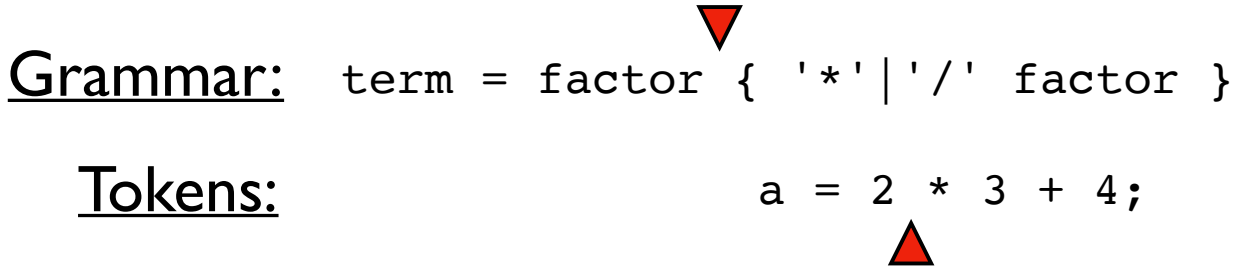
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

# Parsing Illustrated

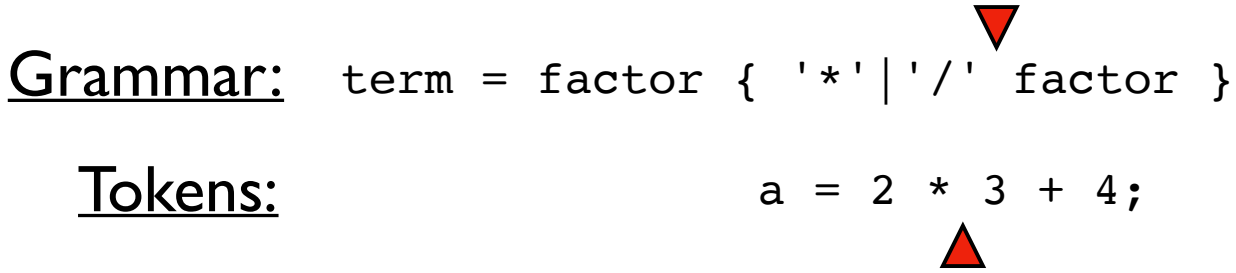
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

# Parsing Illustrated

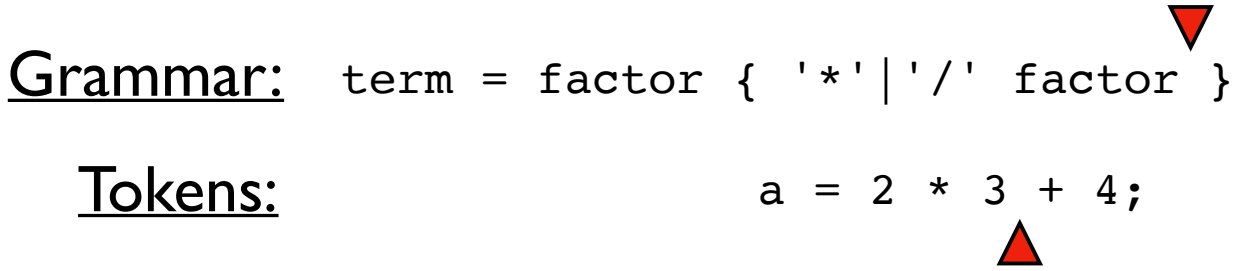
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

# Parsing Illustrated

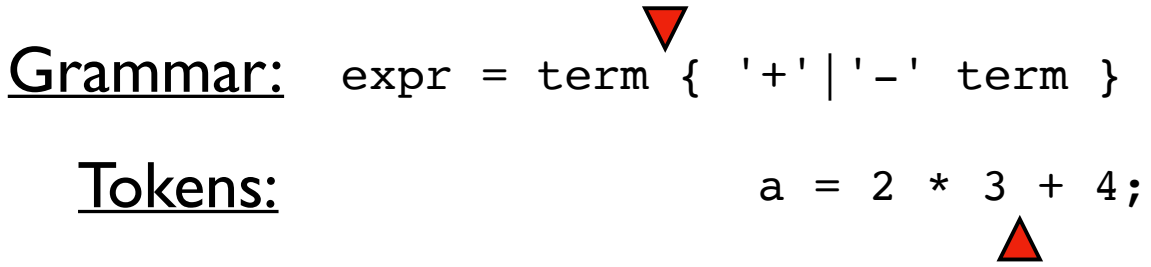
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

# Parsing Illustrated

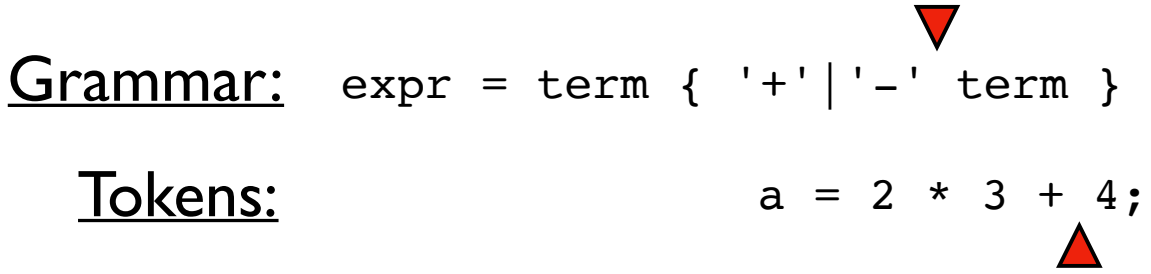
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

# Parsing Illustrated

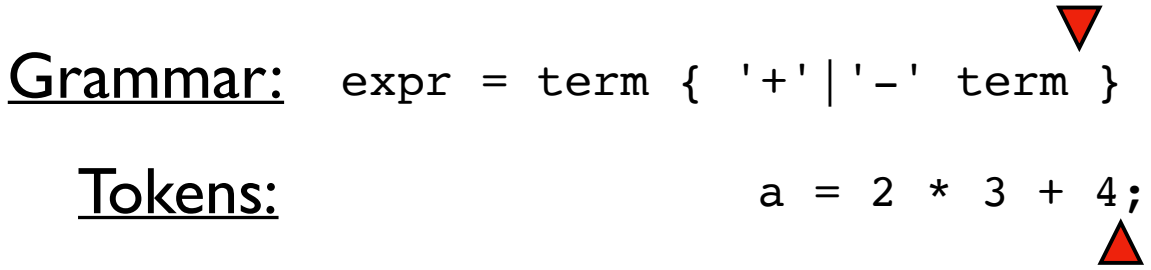
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

# Parsing Illustrated

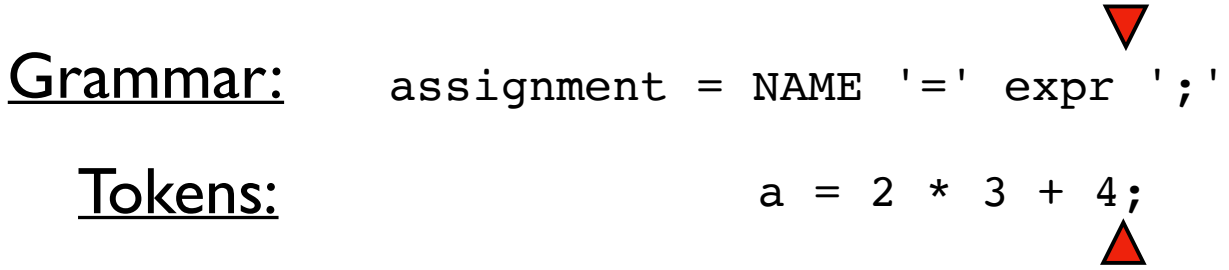
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

# Parsing Illustrated

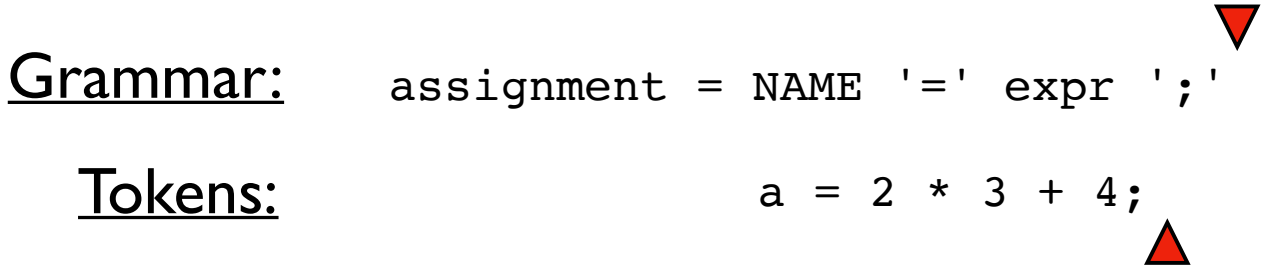
- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens

# Parsing Illustrated

- Parsing involves stepping through tokens




- You try to match to grammar as you go
- Matching descends into grammar rules
- Can only make forward progress on tokens
- You made it! A successful parse.

# Algorithms

- There are MANY different parsing algorithms and strategies, with varying degrees of power and implementation difficulty
- Usually given cryptic names
  - LL(1), LL(k)
  - LR(1), LALR(1), GLR
- Honestly, details aren't that important here

# Parsing Strategies

- Top Down: Start with the grammar rules. Make forward progress by looking at what tokens you expect (according to the rules).
- Bottom Up: Start with the tokens. Make progress by matching the tokens seen so far with the grammar rules that they might match.

# Digression

- Let's talk about CFGs...
- Originally invented by Noam Chomsky for formalizing linguistic structure of natural language
- Adopted as a formalism for CS later...

# Complexities of CFGs

- A major complexity: ambiguity/choice

```
assignment ::= NAME '=' expr ';'

expr       ::= expr '+' expr
             |  expr '-' expr
             |  expr '*' expr
             |  expr '/' expr
             |  '(' expr ')'
             |  INT
             |  FLOAT
             |  NAME
```

- The "|" introduces choices
- Choices imply branching (fork in the road)
- Branching greatly complicates parsing algorithms

# Alternative: PEGs

- Parsing Expression Grammar (PEG)

```
assignment <- NAME '=' expr ';'
expr       <- term (('+'/'-') term)*
term       <- factor (('*'/'/') factor)*
factor     <- NUMBER / NAME / '(' expr ')'
```

- Looks somewhat similar to an EBNF + Regex
- Choice ("|") is replaced by first match ("/")
- No branching. No ambiguity is allowed.

# Alternative: PEGs

- Example:

```
rule <- e1 / e2 / e3
```

- Rules specifies a first-match strategy

```
1. Try to parse e1.  If success, done.
2. Else, try to parse e2. If success, done.
3. Else, try to parse e3. If success, done.
4. Else, parse error.
```

- Requires backtracking.  If parsing fails for an item, backtrack to start, try parsing next item
- Specification order has significance (rules listed first have higher priority)

# Alternative: PEGs

- PEGs are much more modern
- Bryan Ford, "*Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*", POPL 2004 (ACM).
- Not found in most traditional compiler books

# Project 8

Find the file `wabbit/parse.py`

Follow instructions inside.