Write a Compiler (in Python)

David Beazley
http://www.dabeaz.com

June 2019

Materials

- Download and extract the following zip file

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

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

Deep Thought

Programming

"Metal"

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

How does it all work????

Metal

Machine Code

```
5548 89e5 897d fcc7 45f8 0100 0000 837d
fc00 0f8e 1800 0000 8b45 fc0f af45 f889
45f8 8b45 fc83 c0ff 8945 fce9 deff ffff
8b45 f85d c300 0000 0000 0000 0000 0000
3500 0000 0000 0001 0000 0000 0000 0000
0000 0000 0000 0000 1400 0000 0000 0000
017a 5200 0178 1001 100c 0708 9001 0000
2400 0000 1c00 0000 88ff ffff ffff ffff
3500 0000 0000 0000 0041 0e10 8602 430d
0600 0000 0000 0000 0000 0000 0100 0006
0100 0000 0f01 0000 0000 0000 0000 0000
```

"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

```
5548 89e5 897d fcc7 45f8 0100
fc00 0f8e 1800 0000 8b45 fc0f
45f8 8b45 fc83 c0ff 8945 fce9
8b45 f85d c300 0000 0000 0000
3500 0000 0000 0001 0000 0000
0000 0000 0000 0000 1400 0000
017a 5200 0178 1001 100c 0708
2400 0000 1c00 0000 88ff ffff
3500 0000 0000 0000 0041 0e10
0600 0000 0000 0000 0000 0000
0100 0000 0f01 0000 0000 0000
```

"Human" readable machine code

High Level Programming

Source Code

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

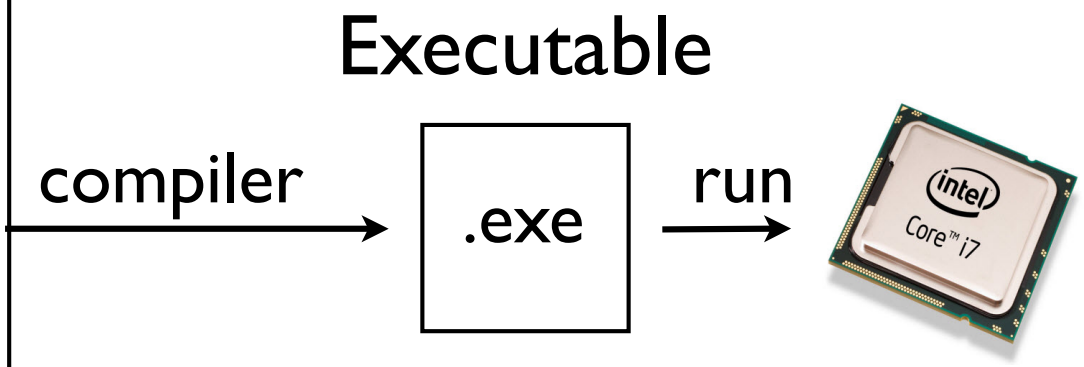
```
fact:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    $1, -8(%rbp)
L1:
    cmpl    $0, -4(%rbp)
    jle     L2
    movl    -4(%rbp), %eax
    imull   -8(%rbp), %eax
    movl    %eax, -8(%rbp)
    mov     -4(%rbp), %eax
    addl    $-1, %eax
    movl    %eax, -4(%rbp)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

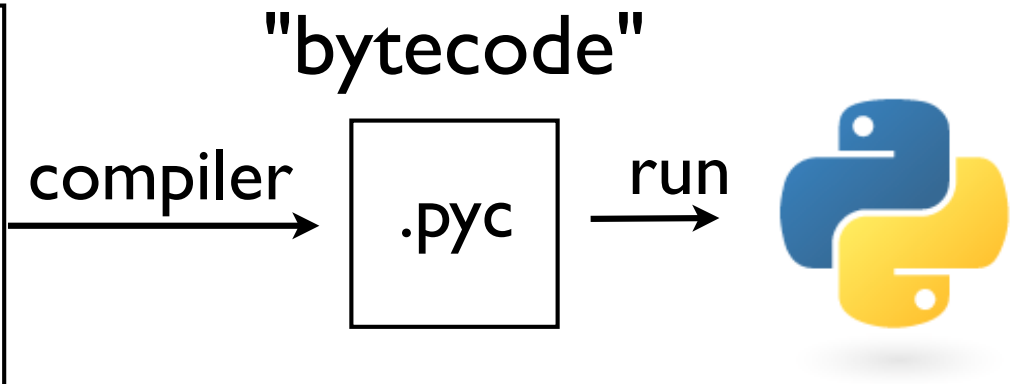


Compiler: A tool that translates a high-level program into bits that can actually execute.

Virtual Machines

Source Code

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



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

Transpilers

Source Code

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

translate

Source Code

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

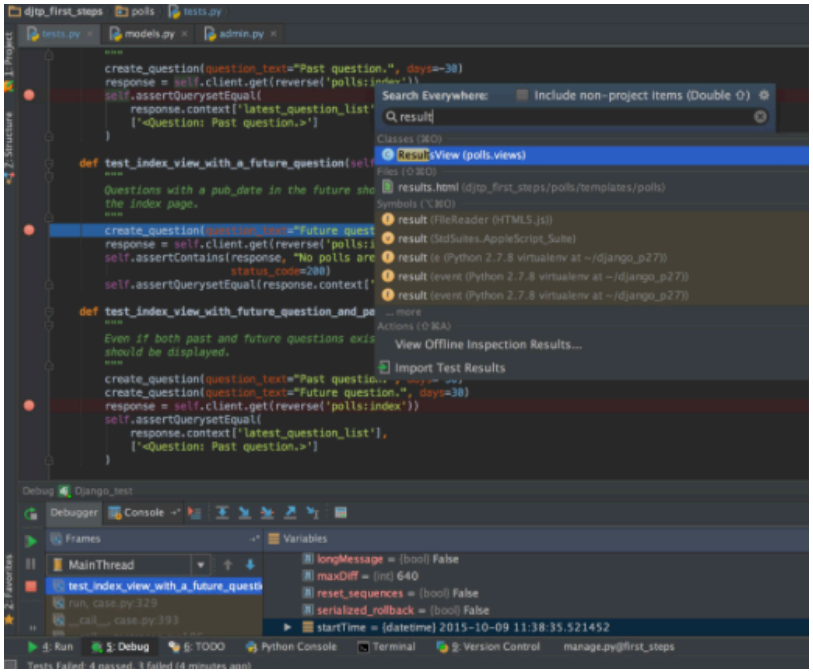
- Translation to a different language
- Example: Compilation to Javascript, C, etc.

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

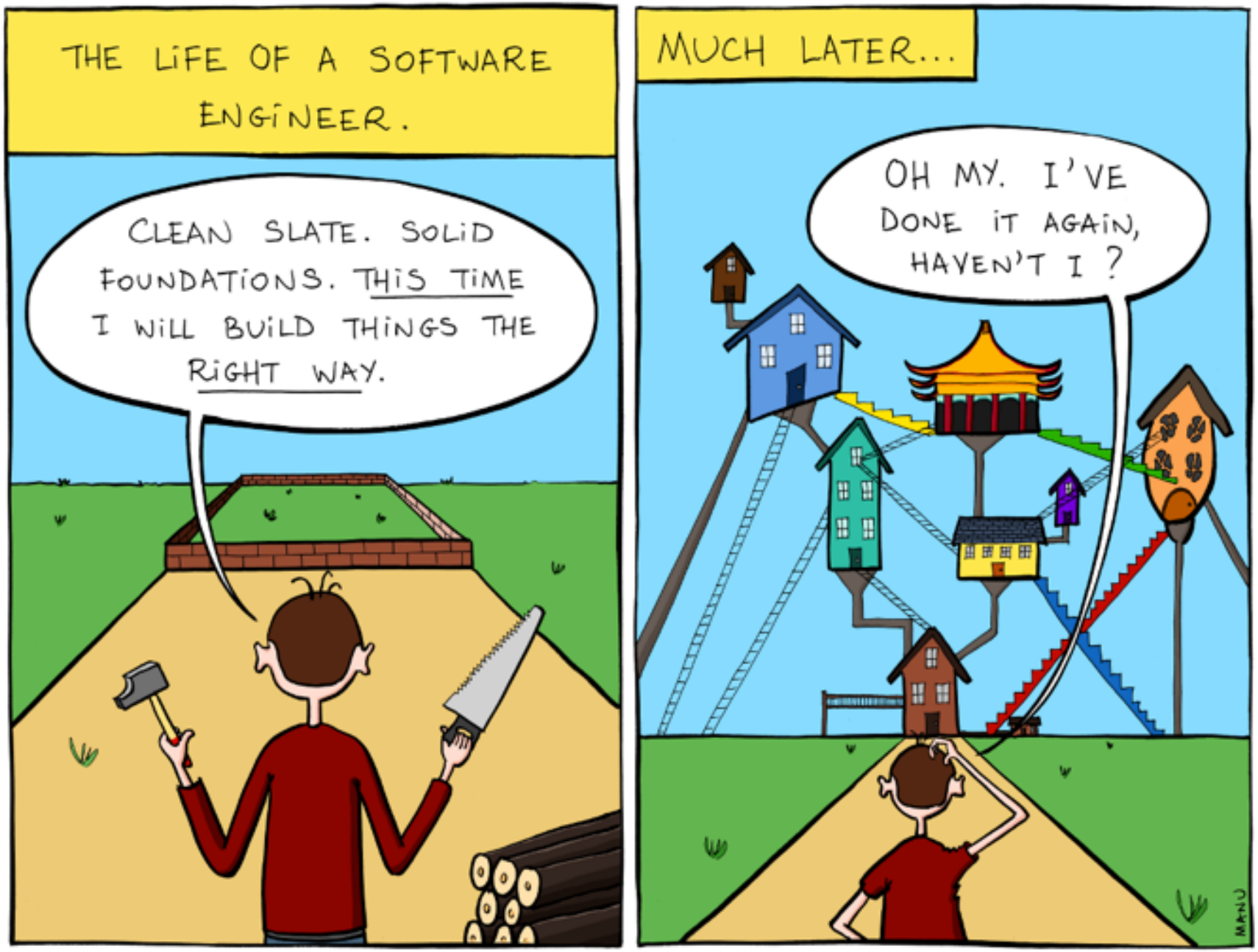
- Compilers are one of the most studied topics in computer science
- Huge amount of mathematical theory
- Interesting algorithms
- Programming language design/semantics
- The nature of computation itself

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

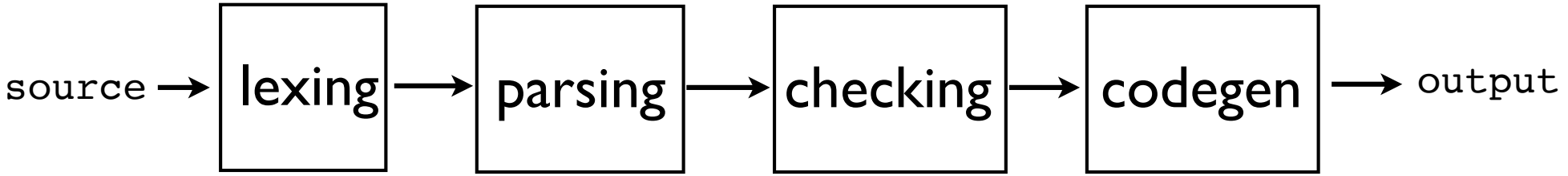
- It's one of the most messy programming projects you will ever undertake
- Many layers of abstraction and tooling
- Involves just about every topic in computer science (algorithms, hardware, etc.)
- Difficult software design (lots of parts)
- Lots of "hacks"



http://www.bonkersworld.net

Behind the Scenes

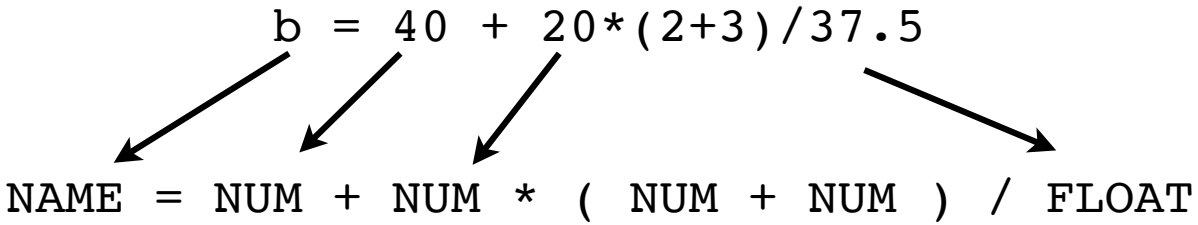
- Compilers are built as a pipeline/workflow



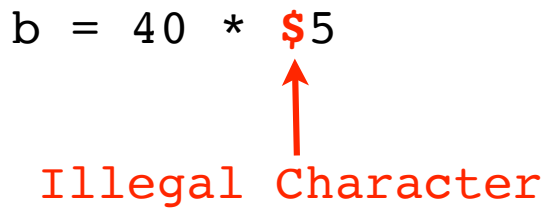
- Each, responsible for a different aspect.

Lexing

- Splits input text into tokens



- Detects illegal symbols



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

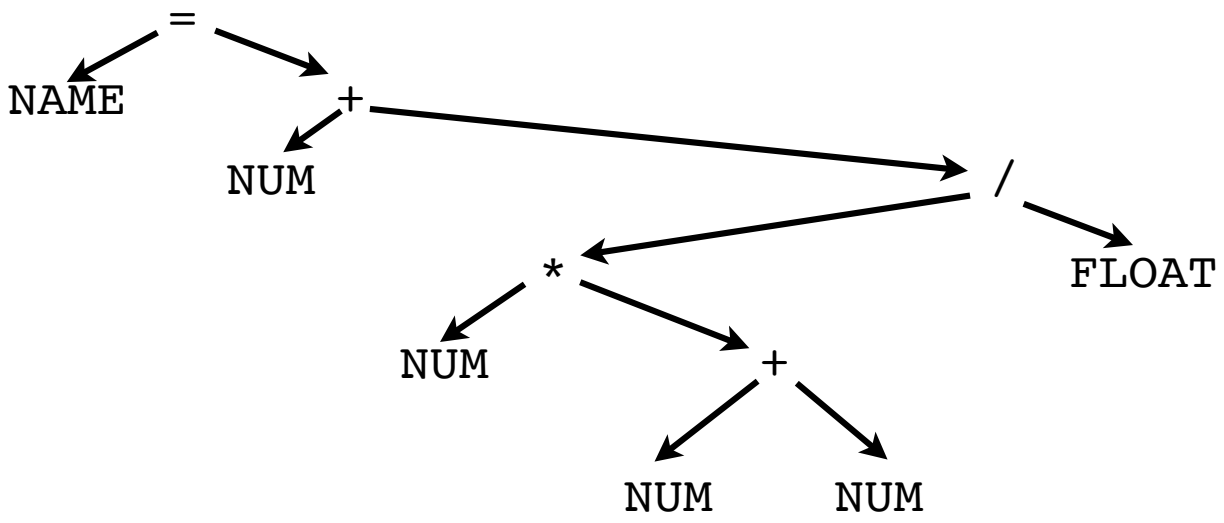
Parsing

- Checks that input is structurally correct

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

- Builds a data structure representing the code

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

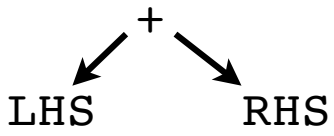


Type Checking

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

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

- Example: + operator



1. LHS and RHS must be the same type
2. The type must implement +

Code Generation

- Generation of "output code":

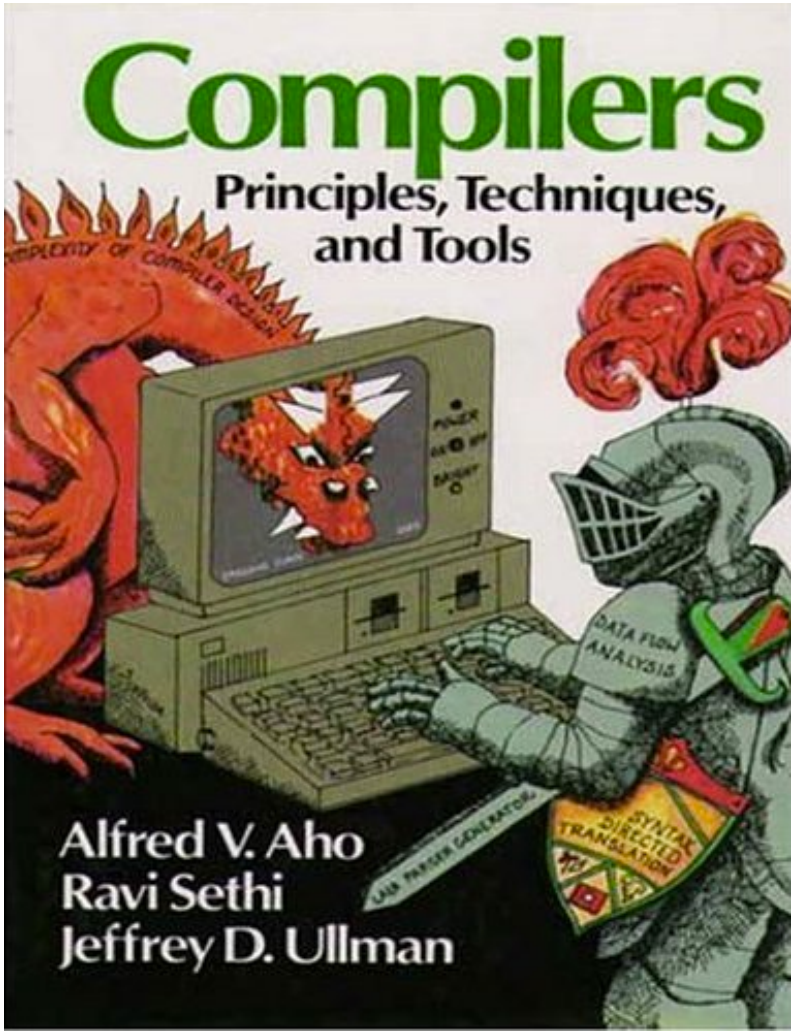
```
b = 40 + 20*(2+3)/37.5
```

```
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”
```

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
- Confidence : If you can write a compiler, chances are you can code just about anything (few tasks are ever *that* complex)

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
    - 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: Will follow the "Star Wars" narrative

Now

understanding the problem

{

program checking — Day 1/2

- Modeling
- Validation
- Unit testing

Day 3

program checking	Code generation

- Wasm, LLVM, etc.

lexing	parsing	program checking	Code generation

Day 4

Project Demo

Tips

- We are going to be writing a lot of code.
- I will be guiding you and giving you code fragments that point you in the right direction
- 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)

Caution

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

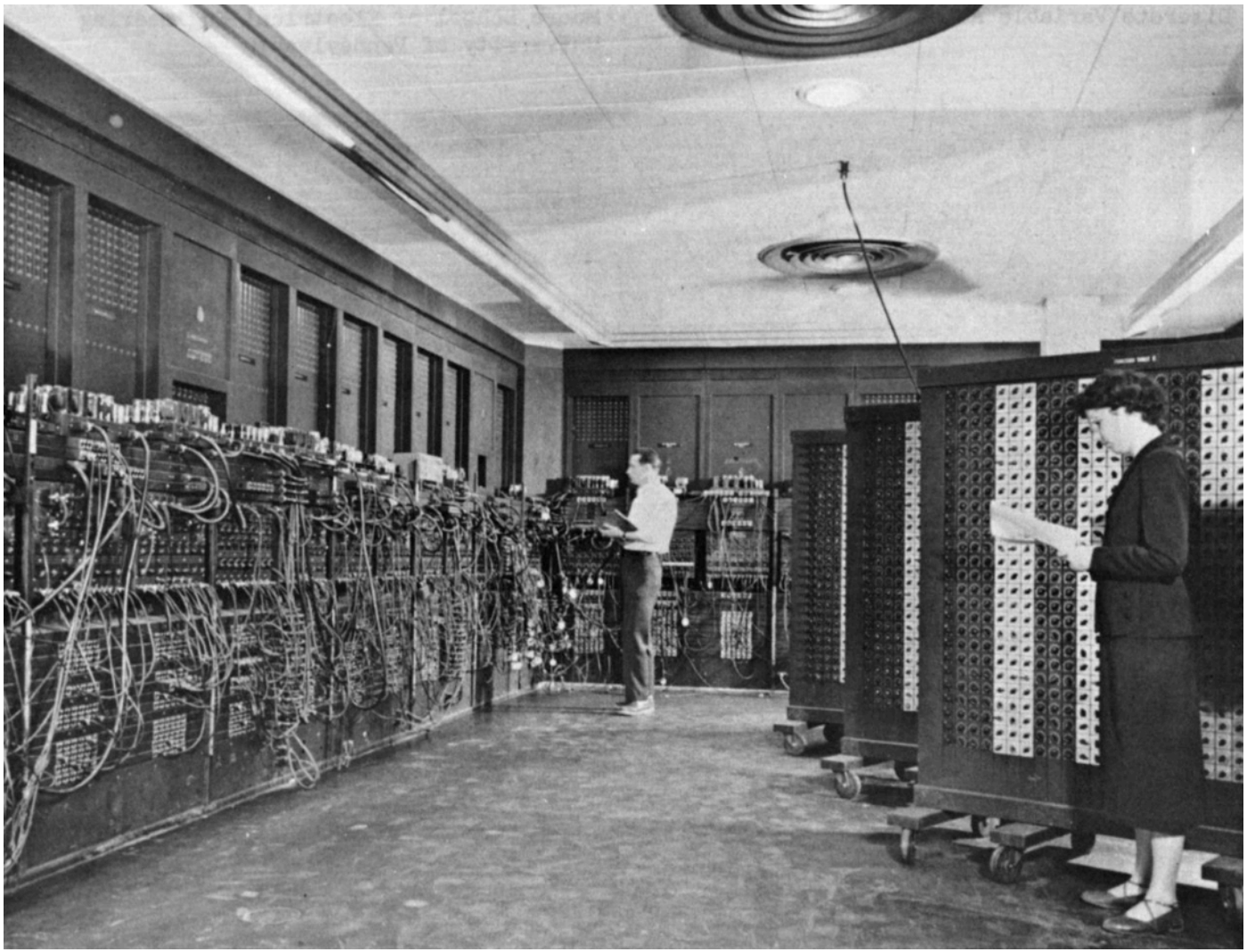
Pace Yourself

- Take breaks, don't overdo it on food, etc.
- Get sleep (you'll need it by day 5)

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?

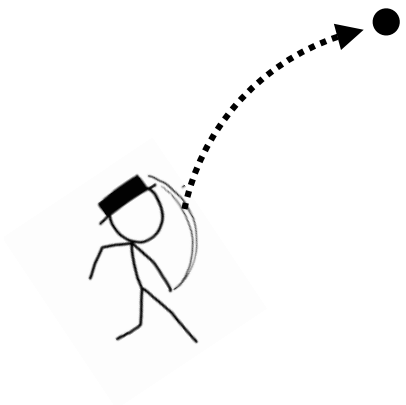
"Computer Science is no more about computers than astronomy is about telescopes."

What does it actually mean to "compute" something?

What can be computed?

Math vs. Computation

- Math: Expressing relationships/properties



```
height = -16*t**2 + V*t
```

- Computation: A process/procedure

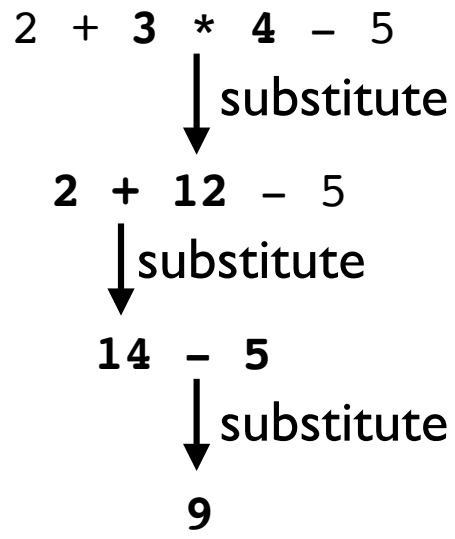
```
Compute: 5!     -> 5 * 4 * 3 * 2 * 1

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

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 substitutions possible

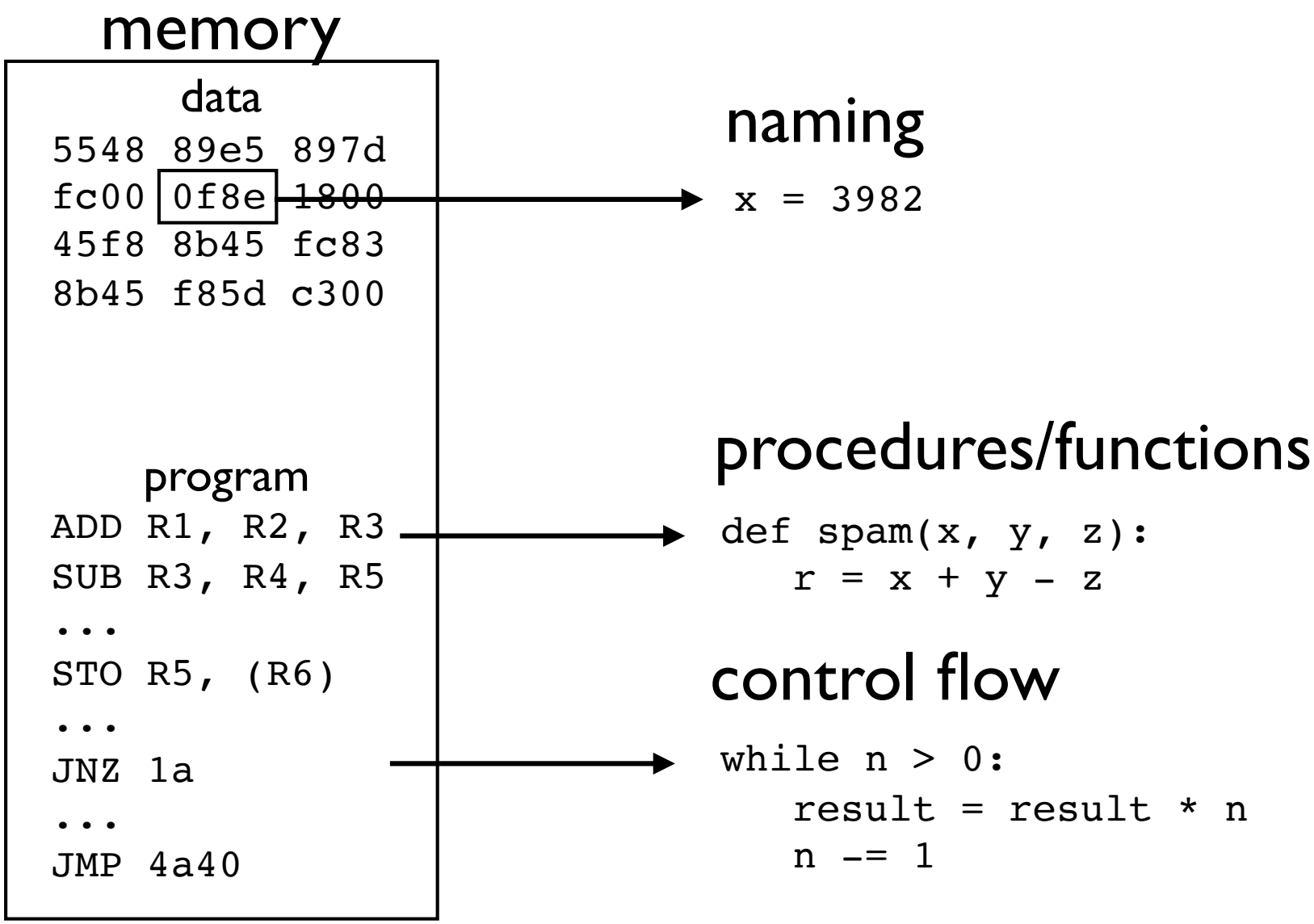
Substitution

- 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 keys on a keyboard??
- Key feature: Abstraction

Programming is Abstraction



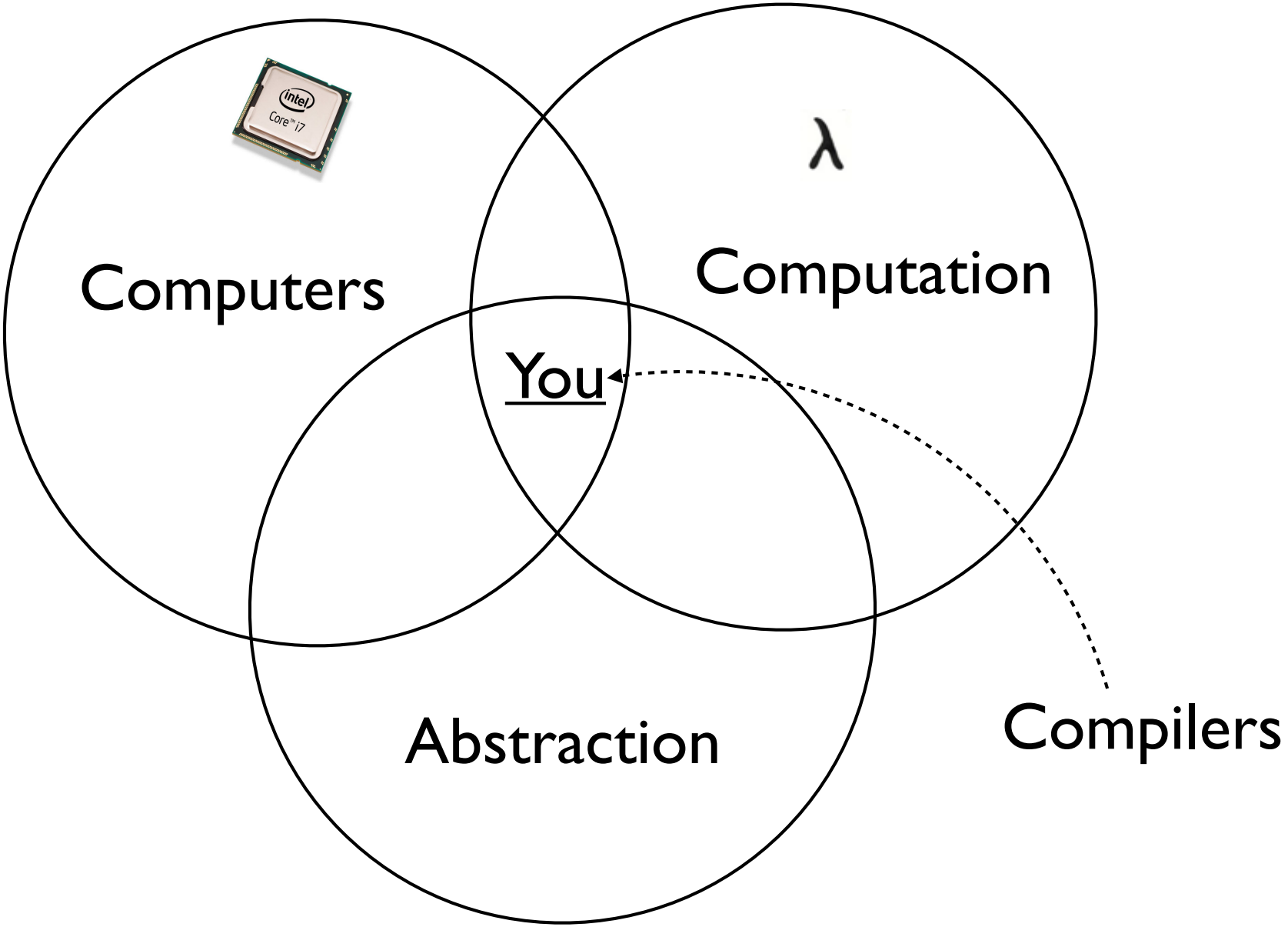
Programming is Abstraction

C Python Go Perl Java Lisp Rust Haskell Scala Javascript

Abstraction

"Metal"

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 the parts of a program?
- 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 simple things
- Indivisible
- The foundation of everything else

Types

- There is usually an underlying type system

```
int
float
char
bool
```

- Values always have a "type"
- Partly it's required to map operations onto actual hardware (e.g., integer operations vs floating point operations).
- Also for code verification/checking

Type Abstraction

- There may be more complex types

```
int [10]          // Arrays

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

- Arrays, pointers, tuples, structures, classes,

Names

- You can name things

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

- Don't hardcode values, use a name

Declarations

- Names might 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;
```

- Note: Python doesn't require this.
- Compiled languages like C usually require it.

Environments/Scopes

- Names are stored in environments

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;
}
```

- Global/local scope

Expressions

- There are operators and expressions

```
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)
```

- Think about math class in school

Locations

- Computers have memory
- Load/store operations

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

- The concept of a "location" is complex

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

- Locations can appear on either side of =

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

Functions/Procedures

- Defining a function

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

- Applying a function (produces a value)

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

- Function parameters vs arguments.

Control Flow

- You can make decisions

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

- And repeat operations

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

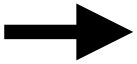
Programs as Data

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

```
23
```

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

Programs as Data

- Example

```
x = 23 + 42;

Assignment(
  GlobalVariable('x', 'int'),
  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.

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
...
```

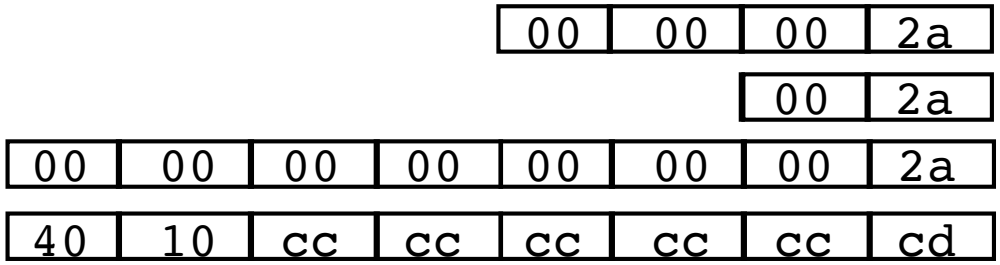
- Each type has different capabilities

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

What is a Type?

- Partly relates to representation of objects

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

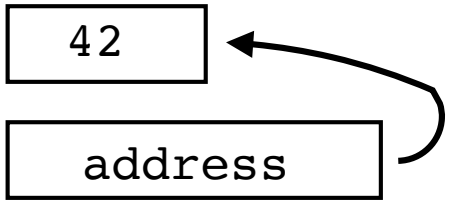


- It directly relates to how raw data gets handled during computation
- Low-level operations on hardware.
- CPUs have limited capabilities

Derived Types

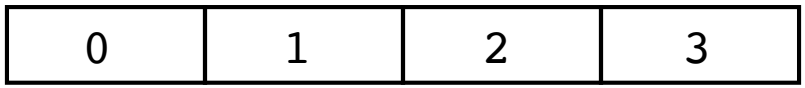
- Types also encode data abstractions
- Pointers/References

```
int a = 42;

int *b = &a;
```



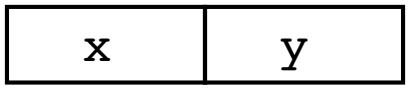
- Arrays

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



- Structures

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



Type Checking

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

Dynamic Typing

- Rules are enforced at run-time
- All objects carry type-information in 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 "discarded" during execution (not needed)

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

The Holy Grail

- How much correctness can a compiler enforce?
- A formal mathematical proof of correctness?
- Algebraic types: Composition of types (primitives, data structures, vectors, etc.)
- Dependent types : types depending on values (e.g., tuple (x, y) where y > x)
- Much of this is very mathematical, very complicated, and probably undecidable.

Commentary

- Specification of formal semantics can be highly mathematical and formal

$$\frac{C, \text{labels}\,[t^?] \vdash instr_1^* : [] \to [t^?] \qquad C, \text{labels}\,[t^?] \vdash instr_2^* : [] \to [t^?]}{C \vdash \text{if}\ [t^?]\ instr_1^*\ \text{else}\ instr_2^*\ \text{end} : [\text{i32}] \to [t^?]}$$

- We are NOT going to do that
- That is covered in a "Type Theory" course
- Will take a more pragmatic/practical approach

Building a Type System

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

Type Specification

- Types are "labels" that get attached to values

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

- Types have names (there is syntax for typing it)
- Keep in mind that the "type" could be much more complex (pointers, arrays, structs, etc.).

Type Specification

- Types must be comparable

```
int != float
```

- A major part of checking is finding type-mismatches in the code (comparing types)
- Might be a simple name comparison
- Probably more complex

Type Specification

- Types have different capabilities

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

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

- Checker will consult when validating

Type Propagation

- Information propagates within the data model

```
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's known.

Enforcing The Rules

- You write checking rules for each kind of object in the data model

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

def check_UnaryOp(node):
    ...

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

- These are basically contracts
- Is the program correct or not?

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')
```

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

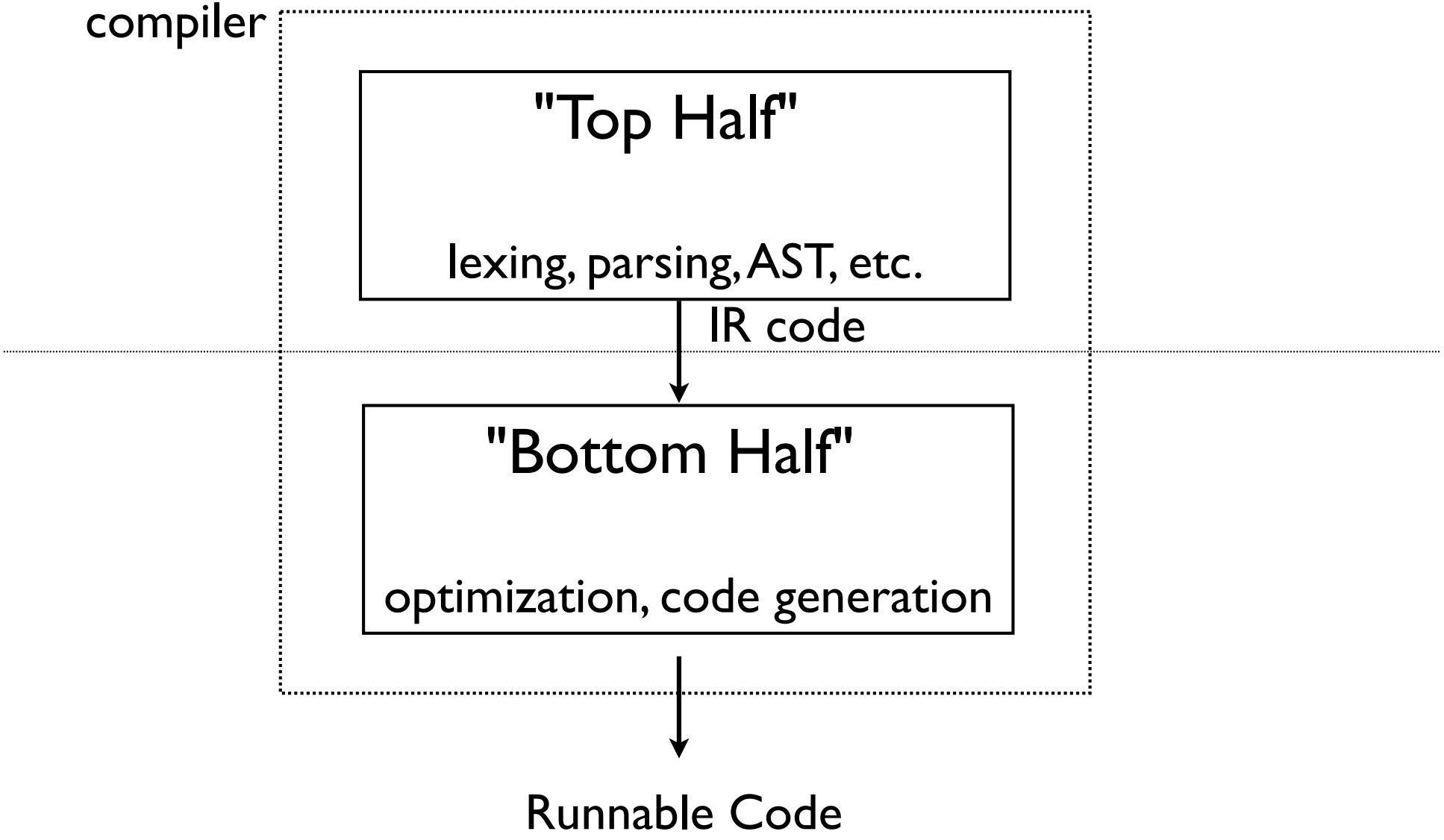
Backing up....

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

Intermediate Code

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

Compiler Design



Three-Address Code

- A common IR where most instructions are just tuples (opcode,src1,src2,target)

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

- Closely mimics machine code on CPUs
- The initial warmup exercise

Three-Address Code

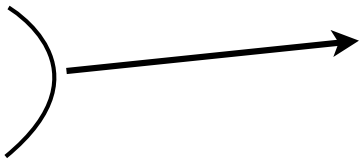
- Example of three-address code IR

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

    t1 = 2              ('CONST',2,'t1')
    t2 = a              ('LOAD','a','t2')
    t3 = t1 * t2        ('MUL','t1','t2','t3')
    t4 = b              ('LOAD','b','t4')
    t5 = t3 + t4        ('ADD','t3','t4','t5')
    t6 = 10             ('CONST',10,'t6')
    t7 = t5 - t6        ('SUB','t5','t6','t7')
    c = t7              ('STORE','t7','c')
```

Optimization

- Many compiler optimization techniques involve analysis of 3AC IR
- Example: peephole optimization

```
t1 = 2
t2 = a
t3 = t1 * t2
t4 = b
t5 = t3 + t4
t6 = 10
t7 = t5 - t6
c = t7
t8 = 10
t9 = c
t10 = t8 * t9
d = t10
```

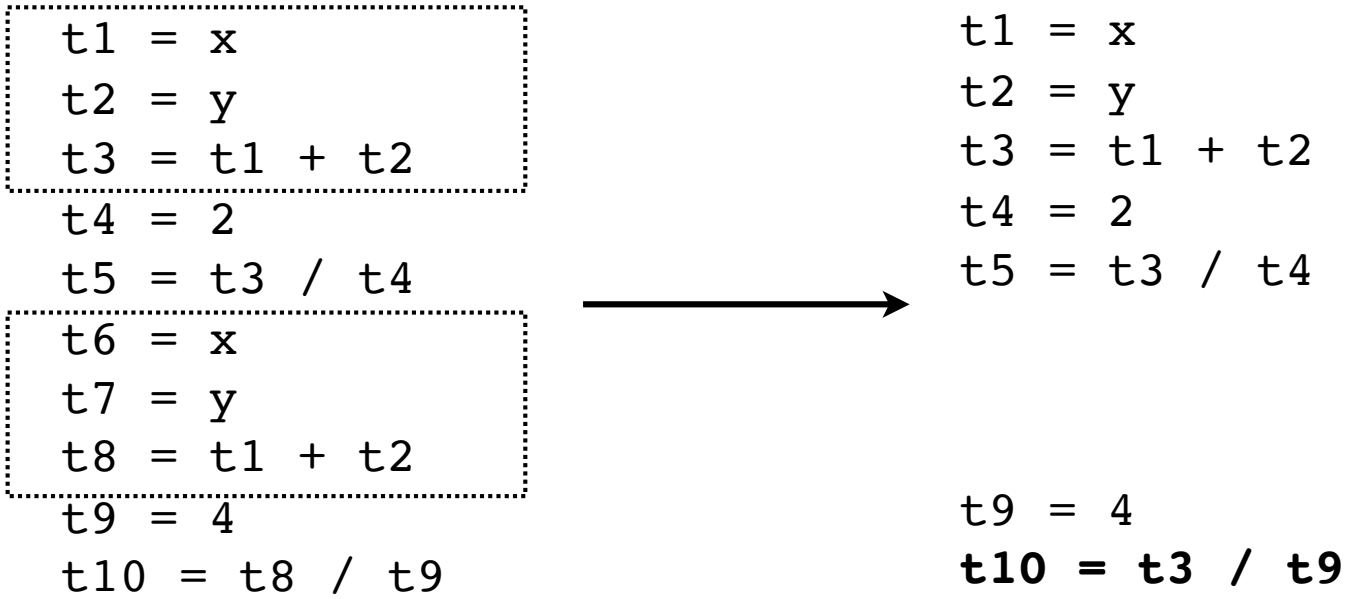


```
t1 = 2
t2 = a
t3 = t1 * t2
t4 = b
t5 = t3 + t4
t6 = 10
t7 = t5 - t6
c = t7
t10 = t6 * t7
d = t10
```

Optimization

- Example: Subexpression elimination

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



SSA Code

- Single Static Assignment
- A variant of 3-address code
  - Can never assign variables more than once
  - Assignments always go to new vars
- Imagine a register machine with infinite registers
- Registers never get overwritten (1-time use)
- This is basis of systems such as LLVM

Example : Stack Machine

- Many intermediate codes are based on a stack architecture
- General idea
  - Operands get pushed onto stack
  - Operators consume stack items
- Like RPN HP Calculators

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
  - Web Assembly

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:

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

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

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

def emit_BinOp(node, code):
    emit(node.left)
    emit(node.right)
    opcode = { '+': 'ADD', '-': 'SUB' }[node.op]
    code.append((opcode,))
```

Project 3

Reading: `docs/ircode.html`

Find the file `wabbit/ircode.py`

Follow the instructions inside

Part 4

Metal

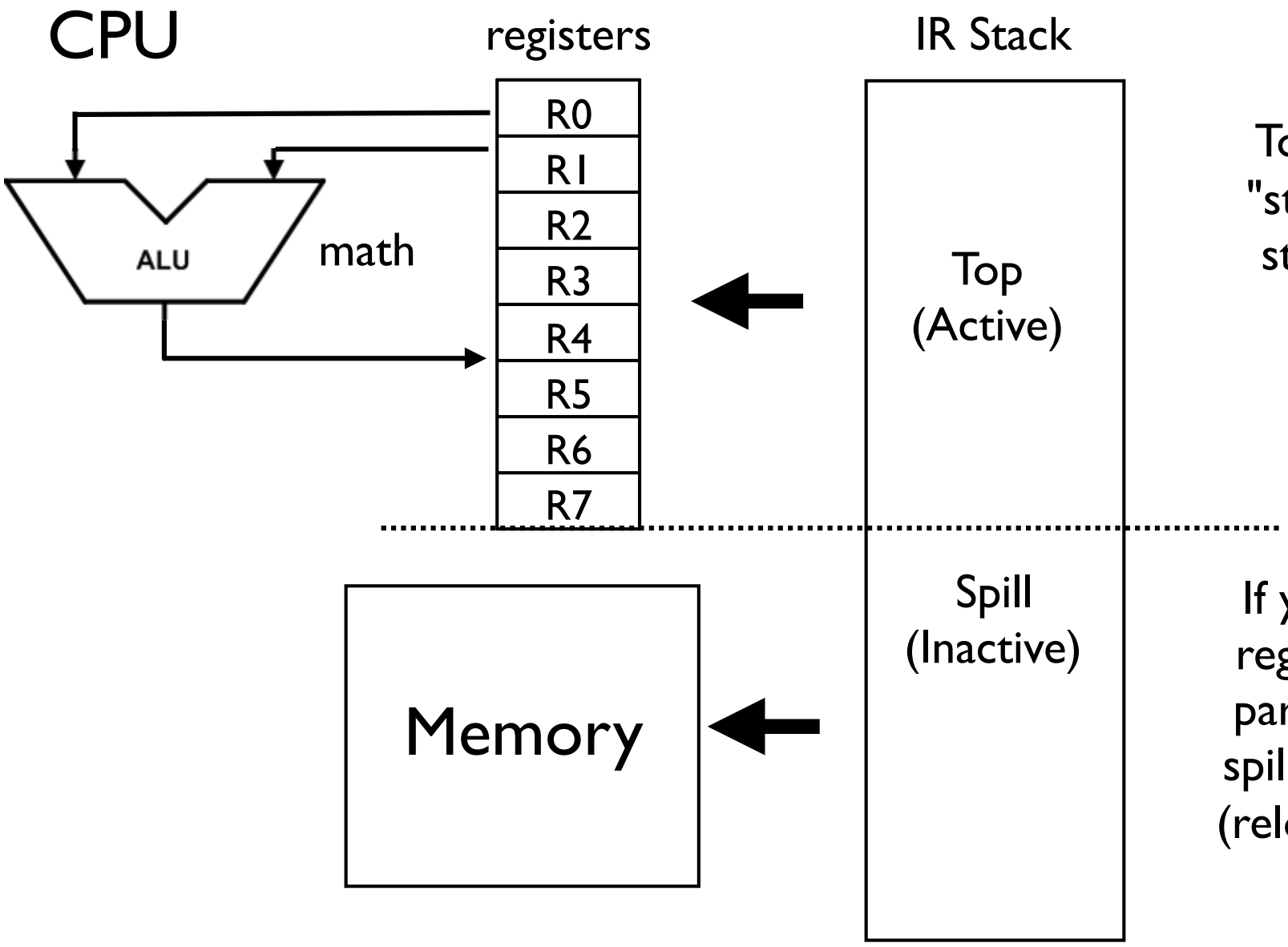
Mapping to Hardware

- 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

Stacks -> Registers

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

Mapping to Hardware



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

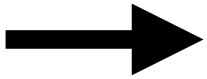
If you run out of registers, bottom part of stack gets spilled to memory (reload when stack shrinks)

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
- Topic: Compiler optimization

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 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
- Operators result different type than operands

```
a = 2
b = 3

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

- What is a truth value?

```
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

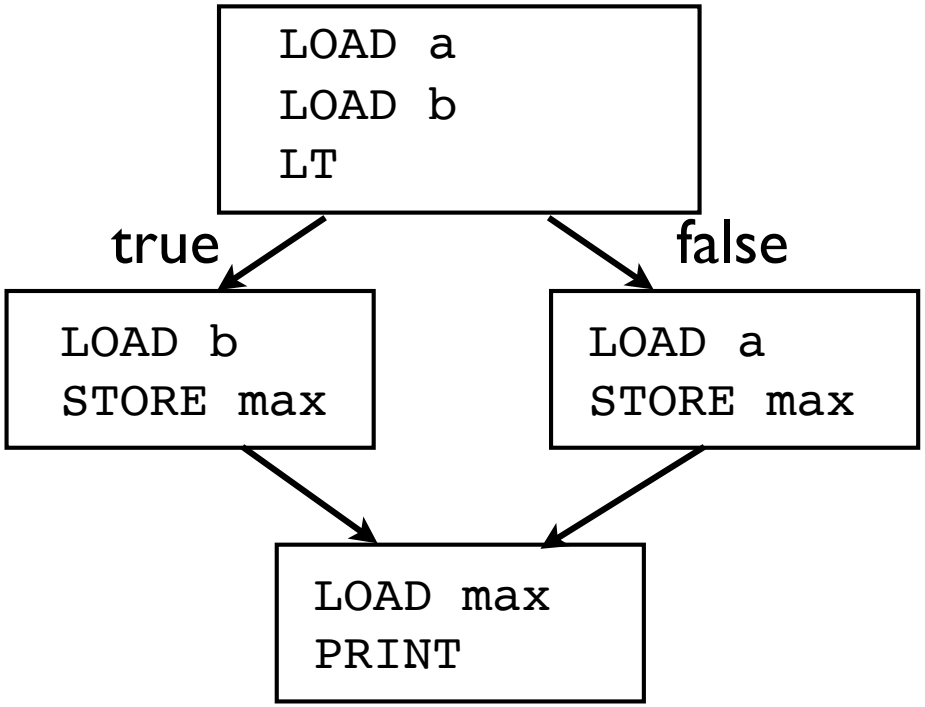
```
var a int = 2;
var b int = 3;
var c int = a + b;
print(2*c);
...
```

- 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

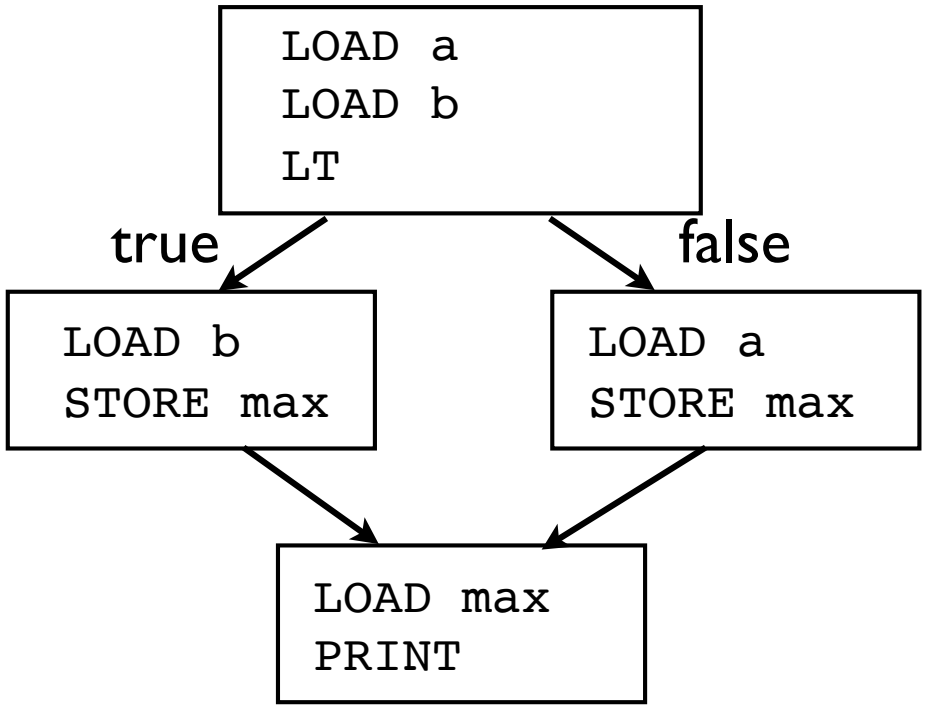
```
var a int = 2;
var b int = 3;
var max int;

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

print max;
```



- Control flow graph

Problem

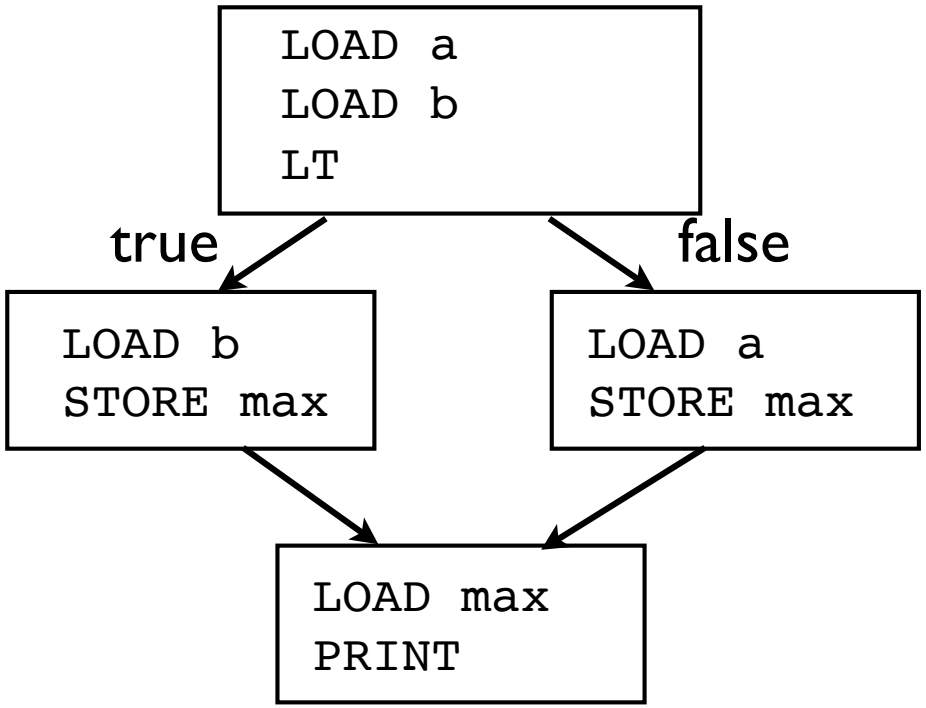
- How do you encode the control-flow graph into intermediate code?



- How is control-flow expressed?

One Approach: Gotos

- Label each block and emit jump/gotos




```
b1: LOAD a
    LOAD b
    LT
    JMP_TRUE b2, 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
}
```

current block

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

Emit jump

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

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 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 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 b3

LABEL b2
statements1
JMP b4

LABEL b3
statements2
JMP b4

LABEL b4
...
```

Commentary

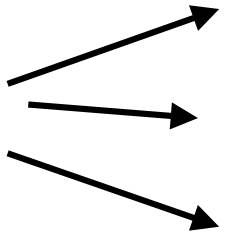
- Emitting jumps and branch statements is how low-level CPUs work
- Can result in a kind of low-level "spaghetti code" that is very hard for humans to follow
- When was last time you used "goto"??

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



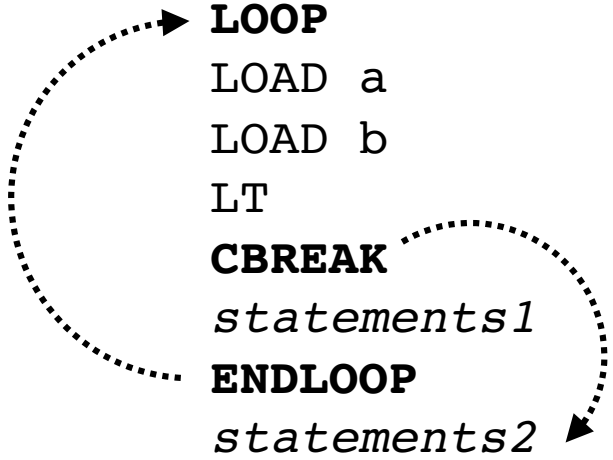
current block

```
...
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 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

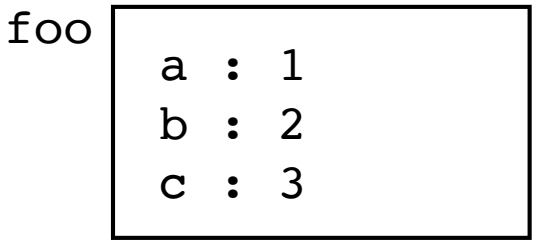
- Each invocation of a function creates a new environment of local variables
- Known as an activation frame (or record)
- Activation frames make up the call stack

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)
```

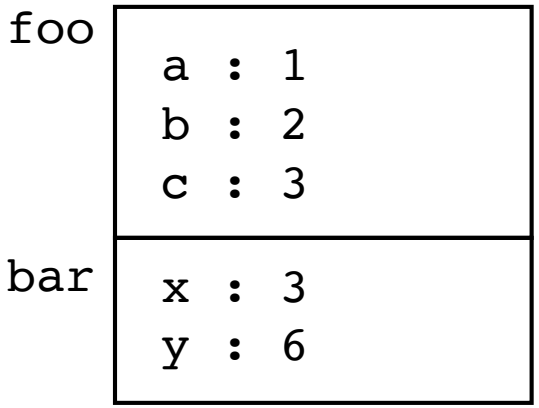
Activation Frames

```
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)
```



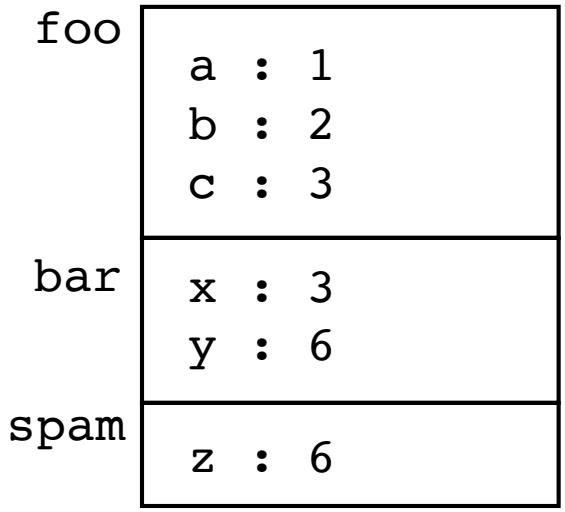
Activation Frames

```
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

- 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_FuncDecl
  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_IfStatem
  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)

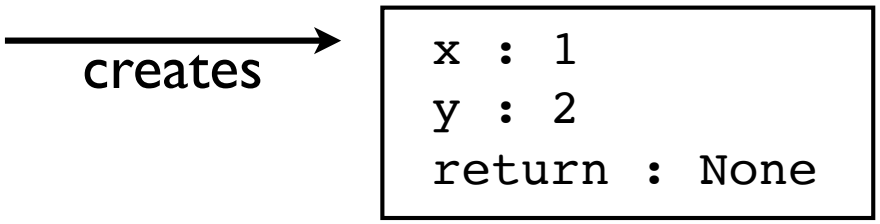
```
def foo(x,y):
    z = x + y
    return z
```

(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 is responsible for creating 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)

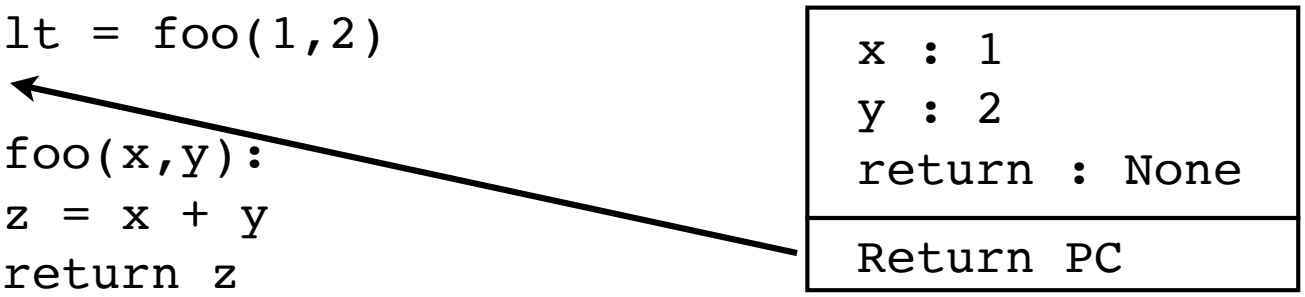
Pointers to the arguments? (Pass by reference)

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
```

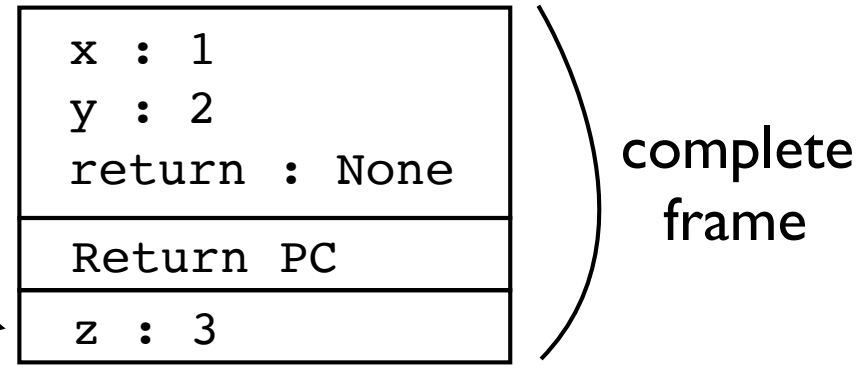


Return address (PC) recorded in the frame (so system knows how to get back to the caller upon return)

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
```

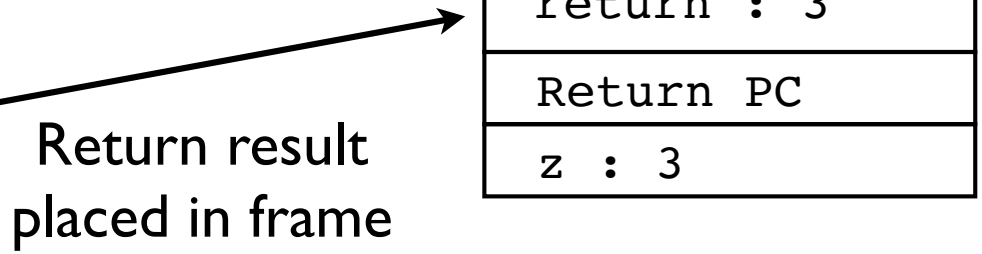


Local variables get added to the frame by the callee

Frame Management

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

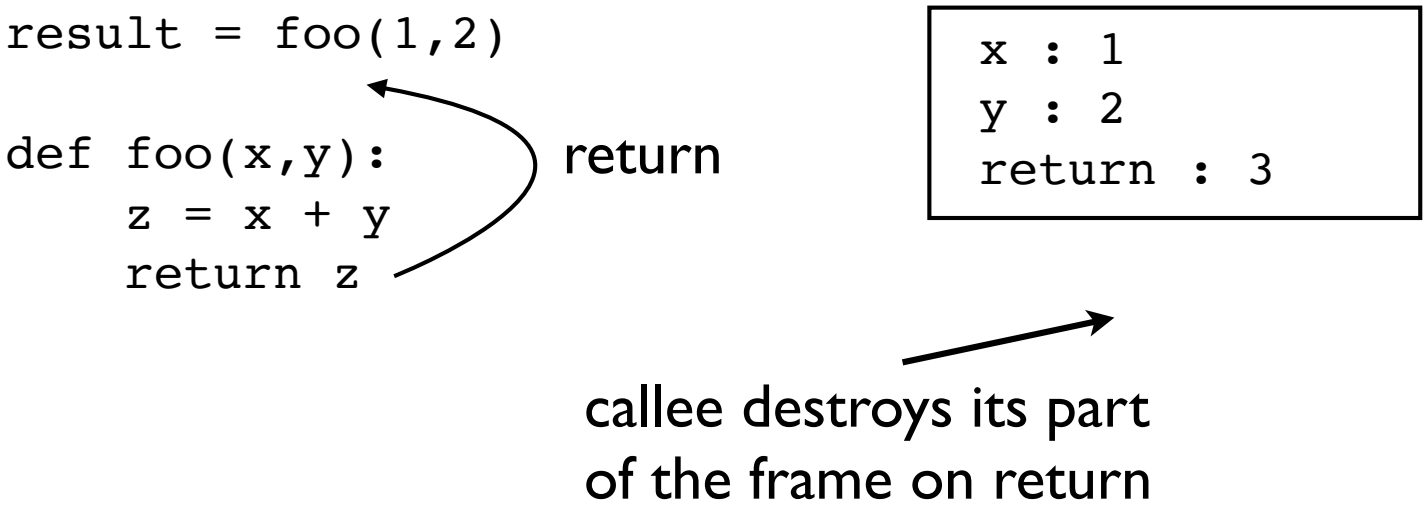
```
result = foo(1,2)

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

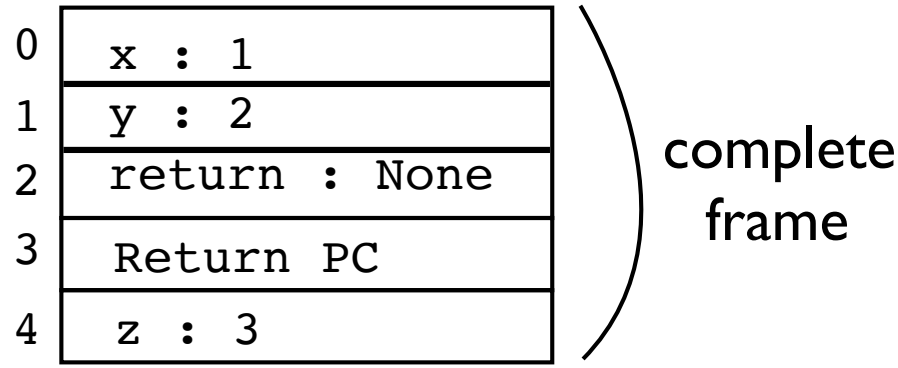
```
result = foo(1,2)

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
```

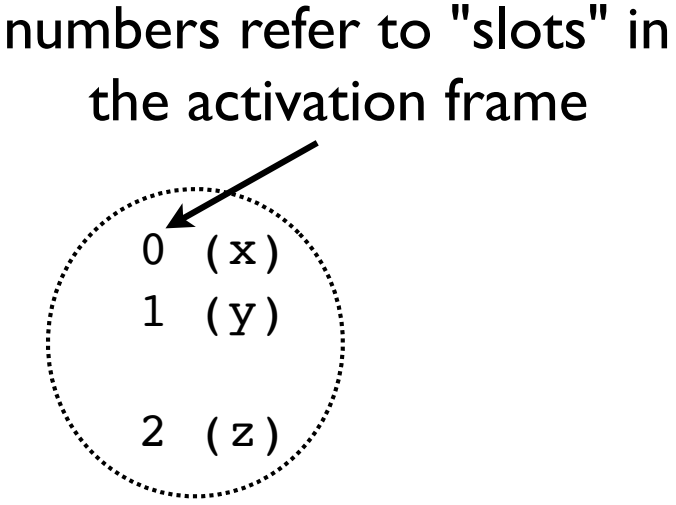


- Slot numbers used in low-level instructions
- Determined at compile-time

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
>>>
```



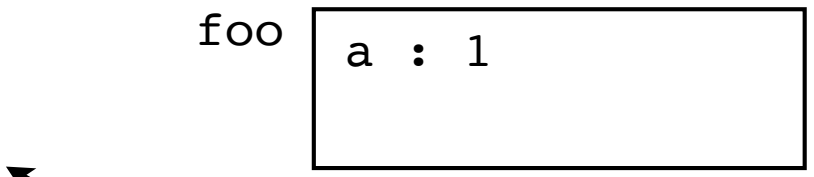
ABIs

- 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)
```

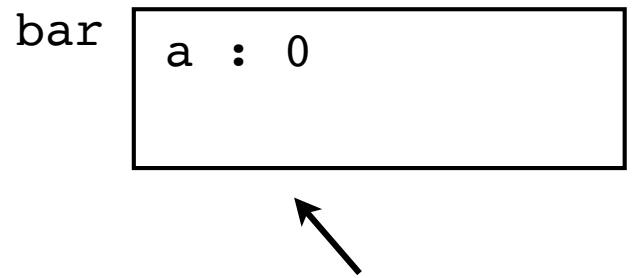


compiler detects that no more statements follow

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 <u>not</u> do this (although people often wish that it did)

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

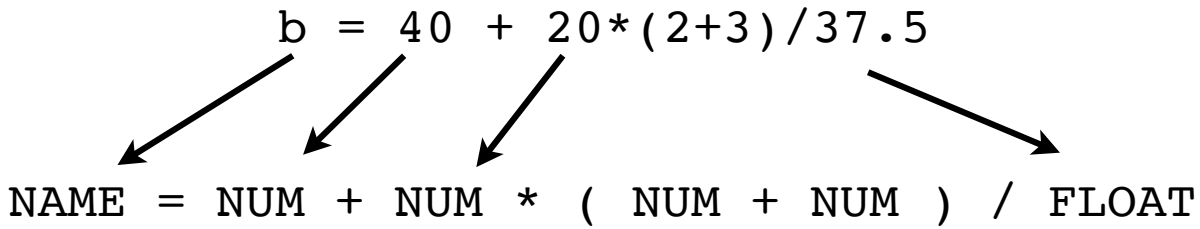
Coding: Modify IR-code and other parts of compiler to support function definitions and function call.

Part 7

Lexing

Lexing in a Nutshell

- Convert input text into a token stream



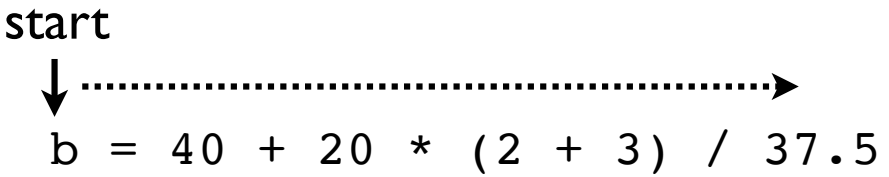
- Tokens have both a type and value

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

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

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

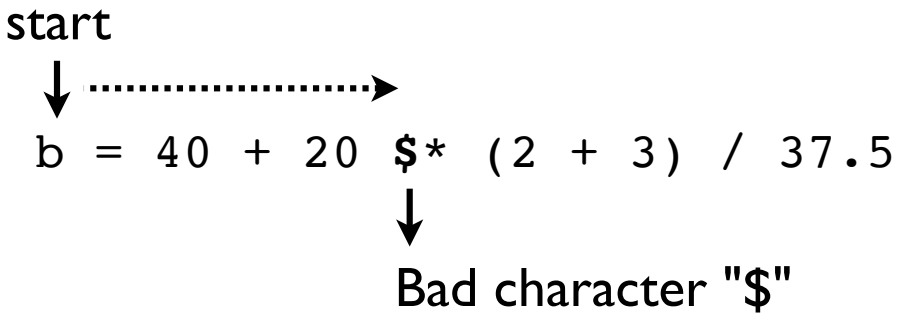
- Question: How to do it?

Text Scanning

- Must perform a linear text scan

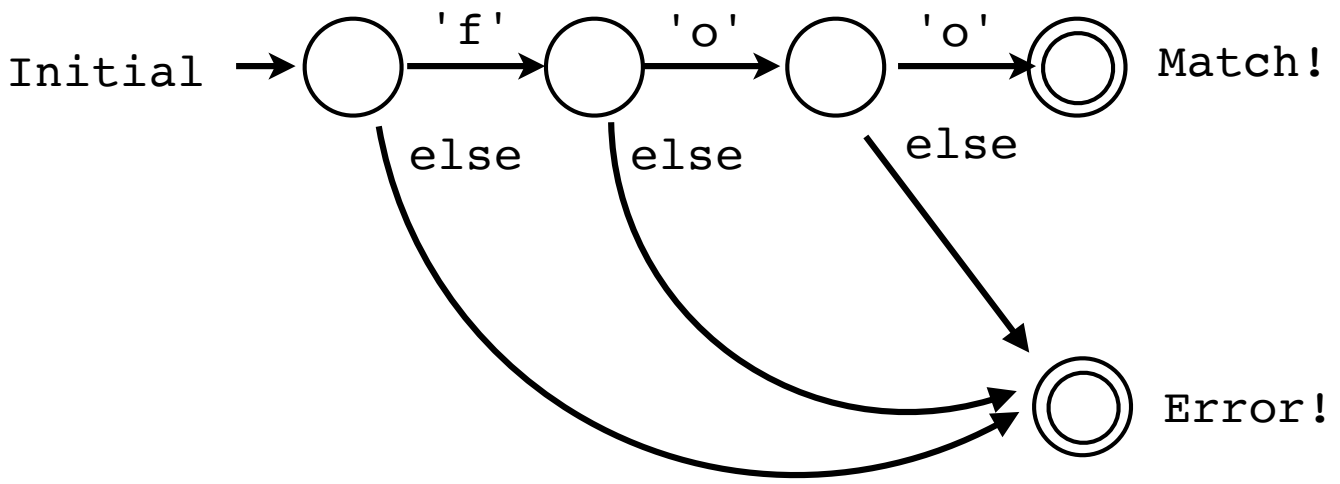


- ALL characters must be matched
- Otherwise error:



Text Matching

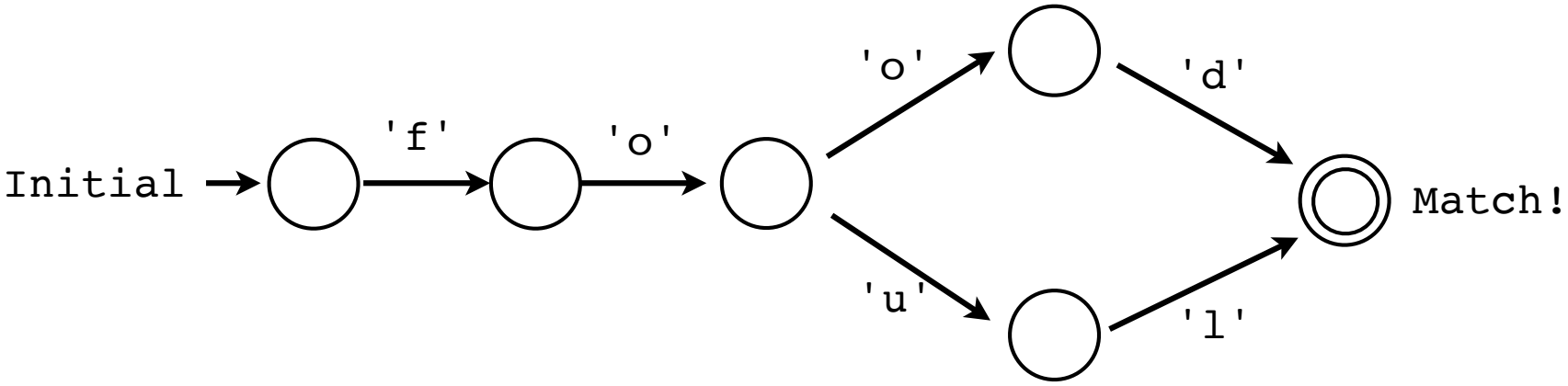
- Characters are processed using a state machine
- Example: Match the text "foo"



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

Matching Alternatives

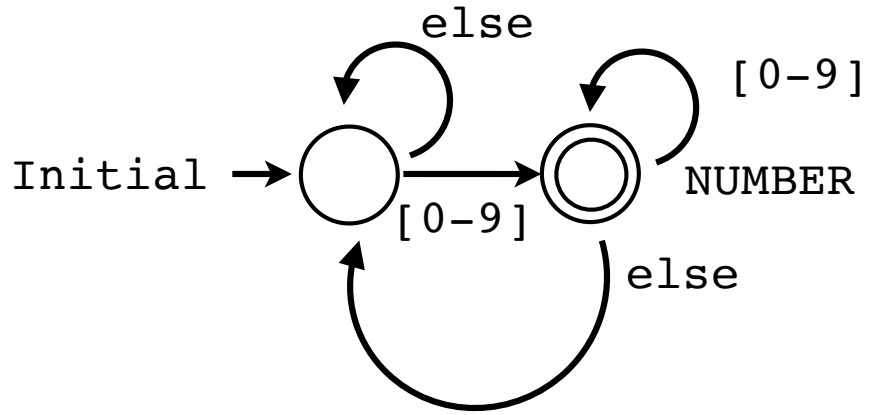
- Example: Match the text "food" or "foul



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]+

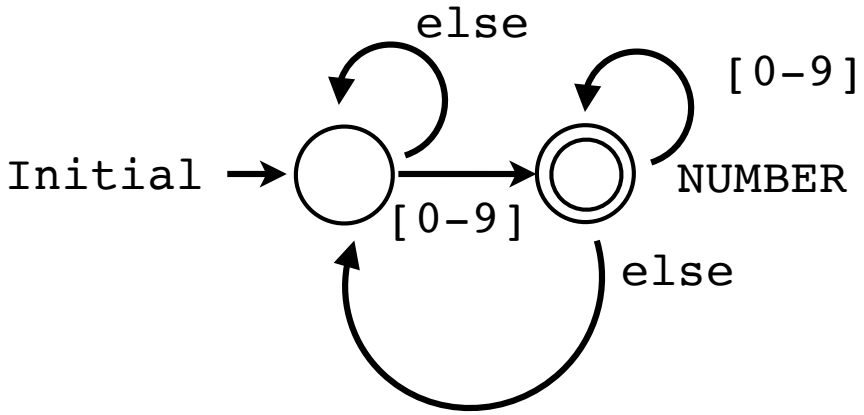


You stay in a matched state to collect additional characters

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

Coding Example

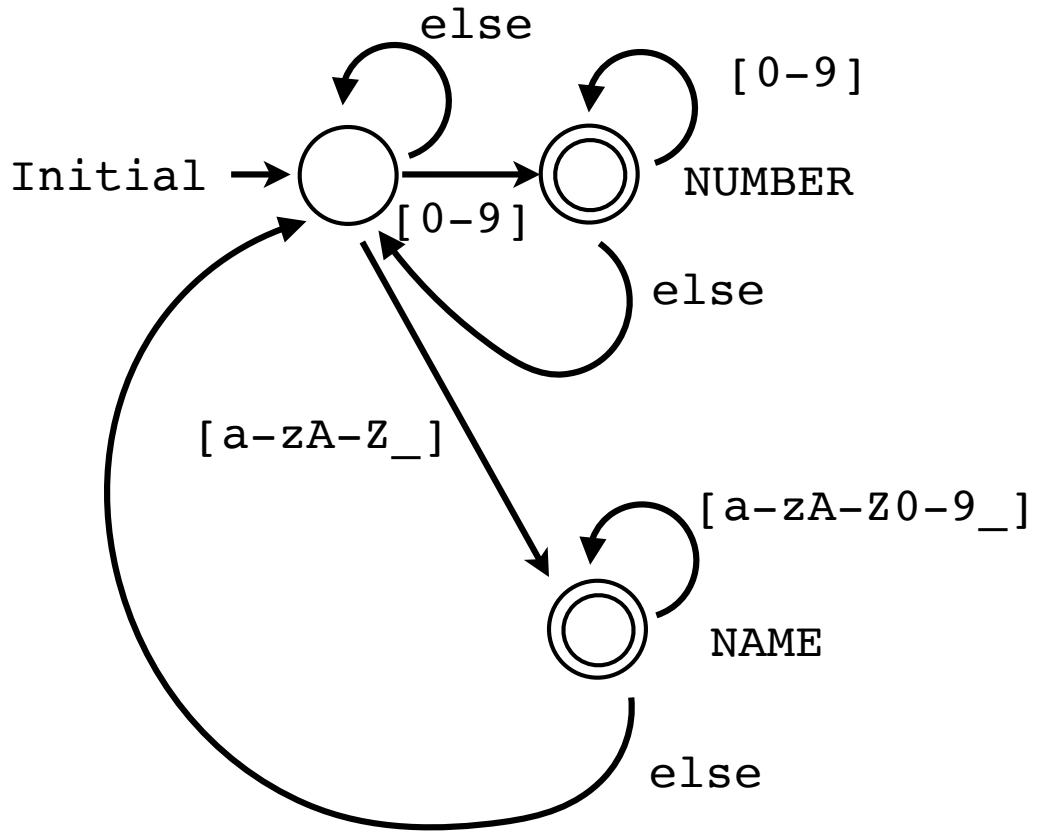
- Example: Match numbers [0-9]+



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

Coding Example

- Find all numbers [0-9]+
- 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

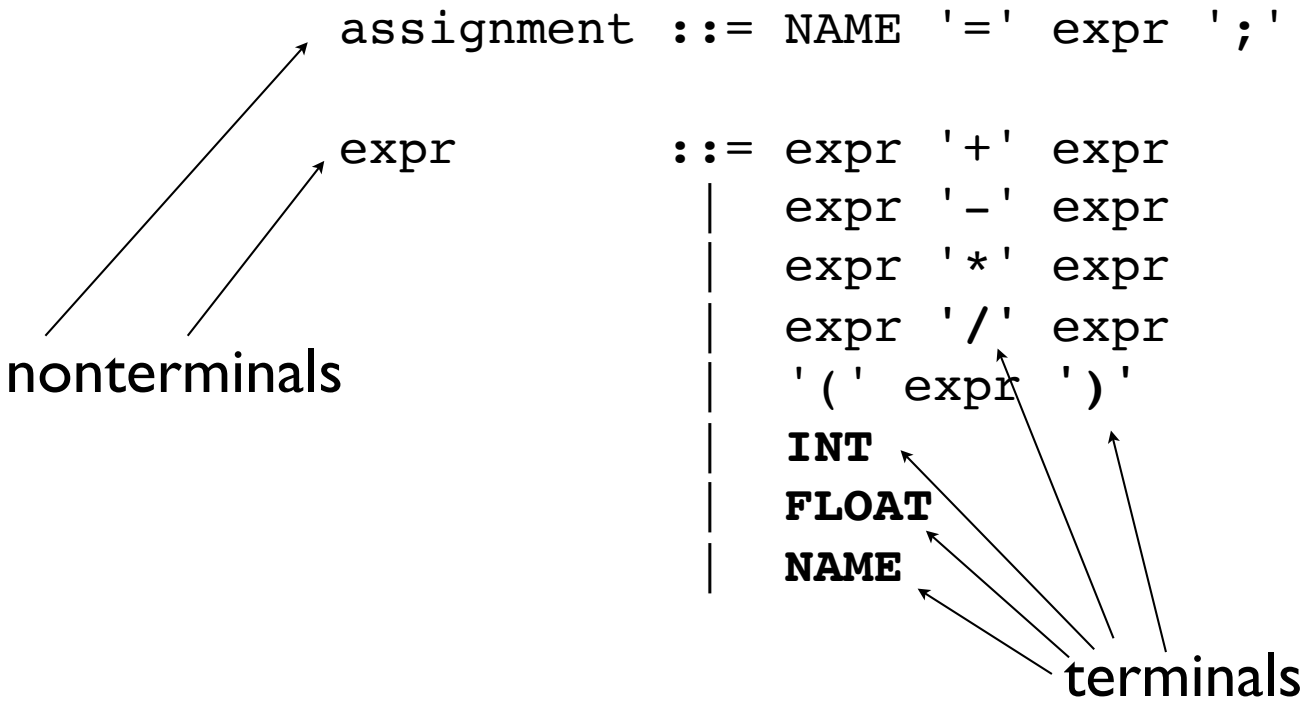
- Syntax is usually specified as a formal grammar

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

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

- Notation is in 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

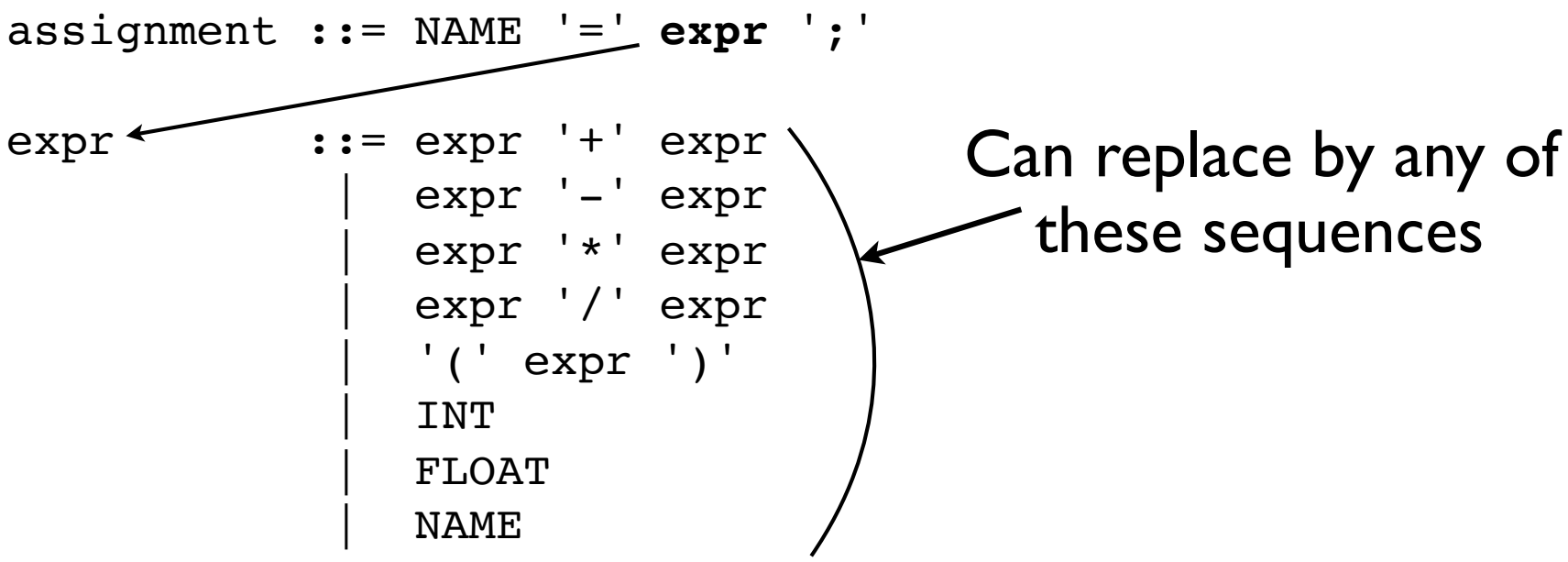
- A BNF specifies substitutions

```
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 can be 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 it 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 matched 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

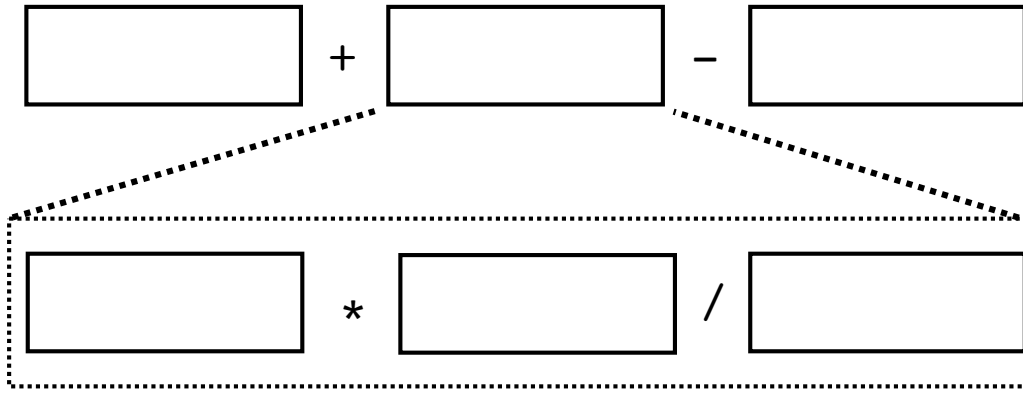
```
expr ::= expr + term
       |  expr - term
       |  term

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 )
```

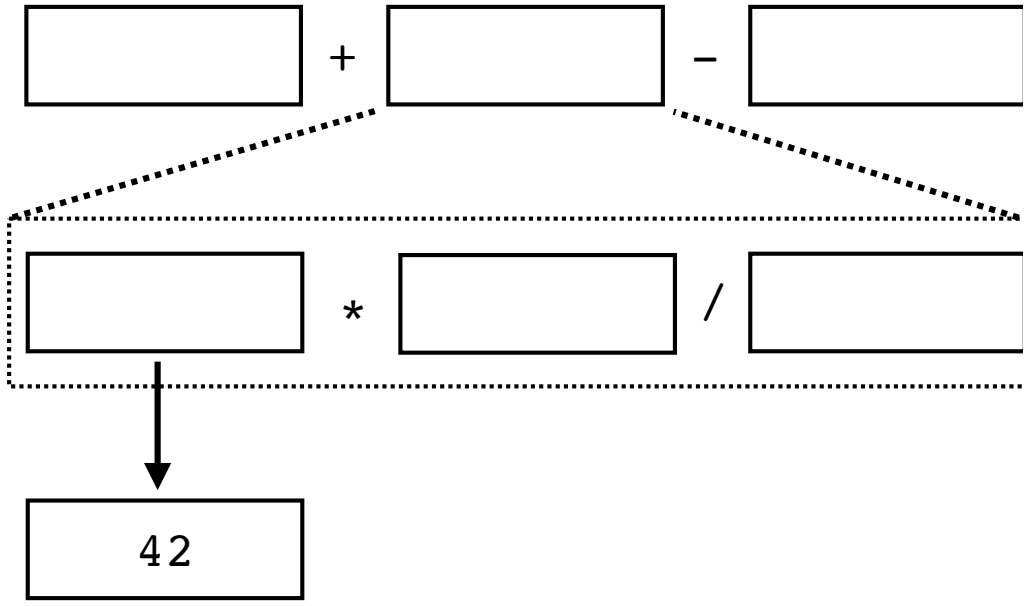


- Idea: Layering from low->high precedence

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

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
```

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
- Will 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

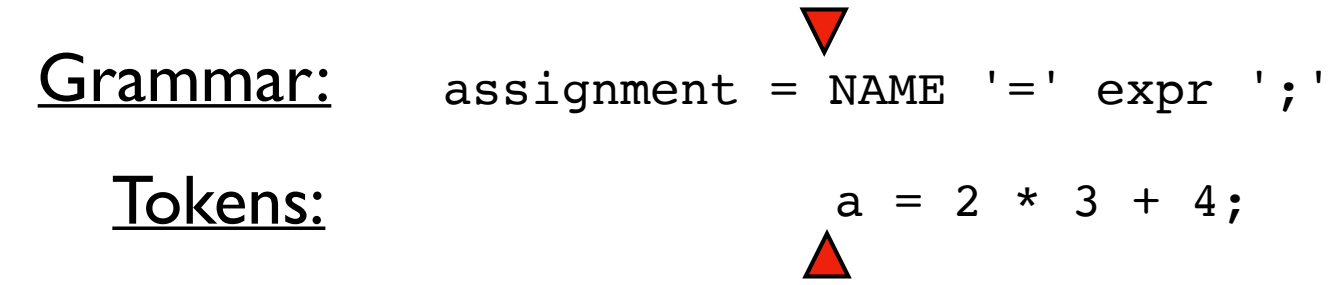
"Why did the parser cross the road?"

"To get to the other side."

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

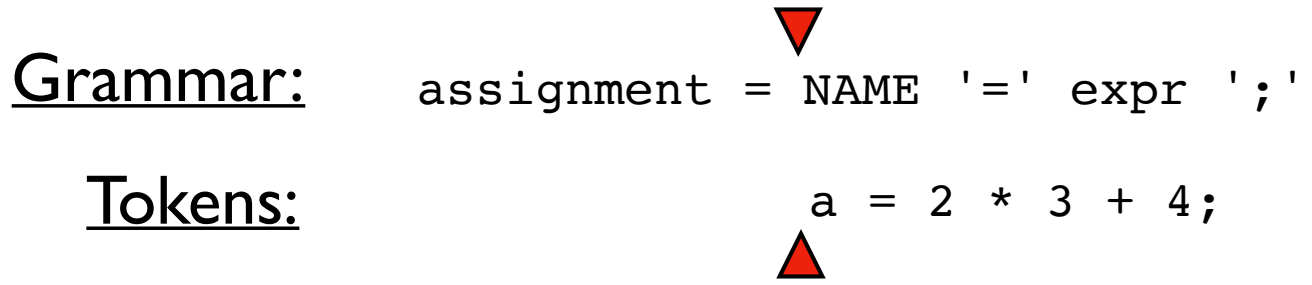
Parsing Algorithms

- In the beginning, you know nothing...

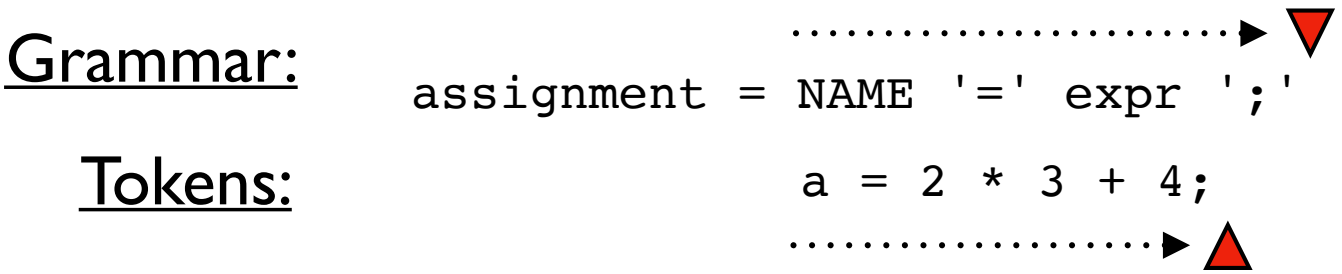


Parsing Algorithms

- In the beginning, you know nothing...

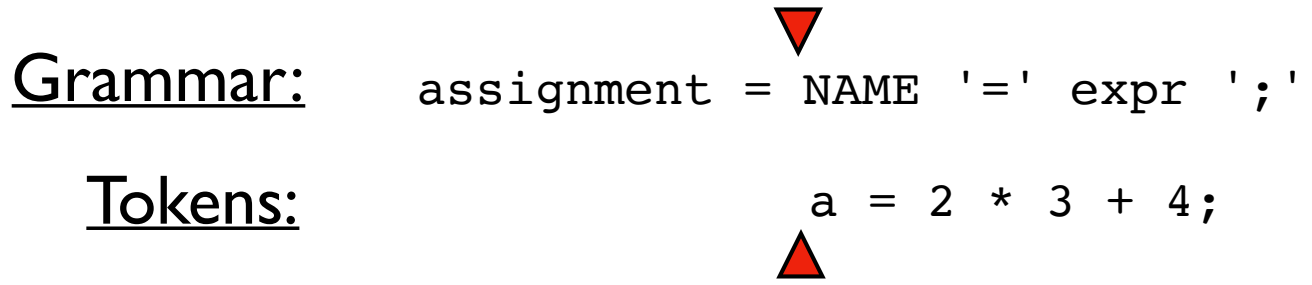


- The goal: move both markers to the other side

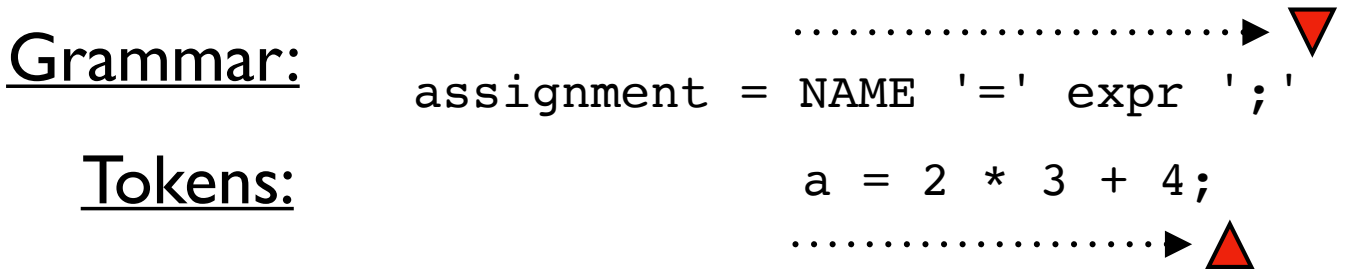


Parsing Algorithms

- In the beginning, you know nothing...



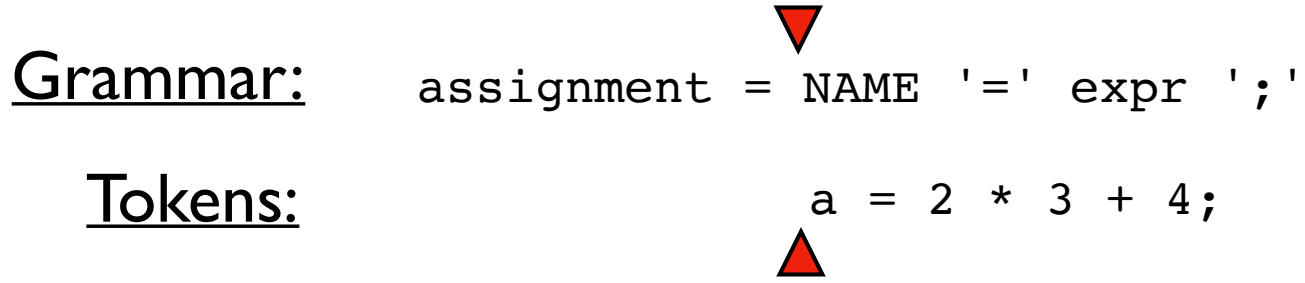
- The goal: move both markers to the other side



- But, you must follow the grammar rules

Parsing Illustrated

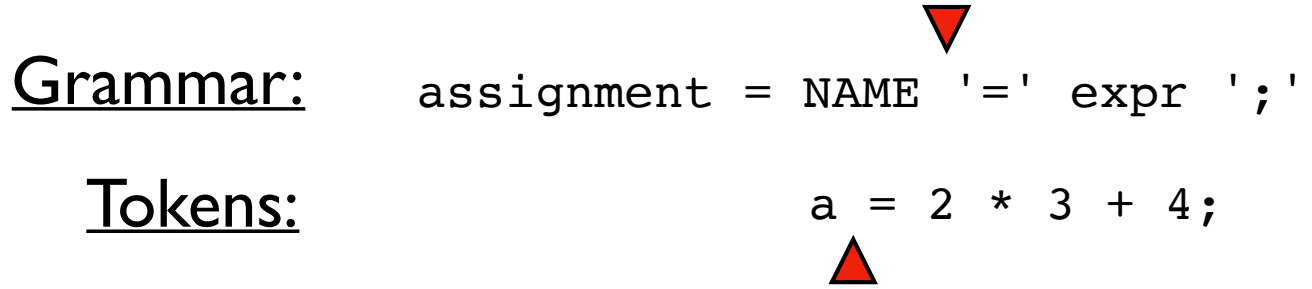
- Parsing involves stepping through tokens



- You try to match things up as you go

Parsing Illustrated

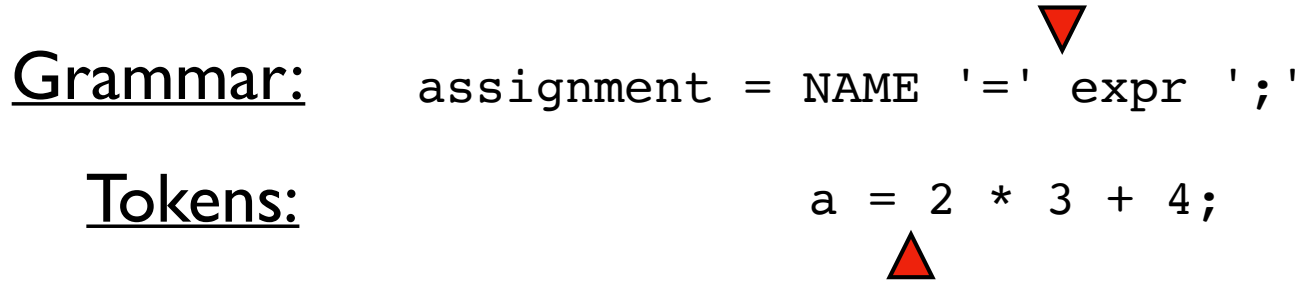
- Parsing involves stepping through tokens



- You try to match things up as you go

Parsing Illustrated

- Parsing involves stepping through tokens



- You try to match things up as you go

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `expr = term { '+'|'-' term }`

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

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

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `term = factor { '*'|'/' factor }`

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

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

Parsing Illustrated

- Parsing involves stepping through tokens

Grammar: `factor = INTEGER | FLOAT`

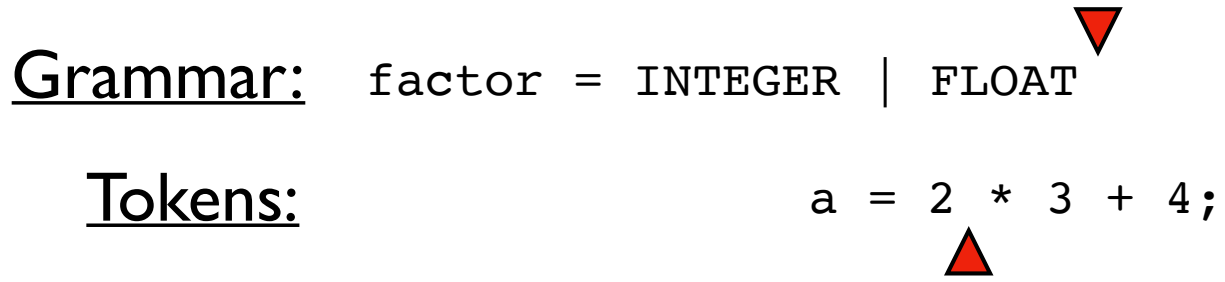
Tokens:

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

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

Parsing Illustrated

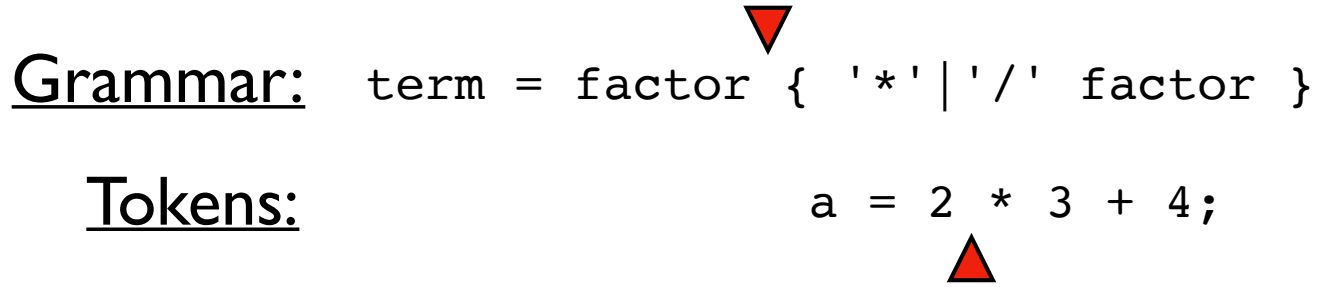
- Parsing involves stepping through tokens



- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress

Parsing Illustrated

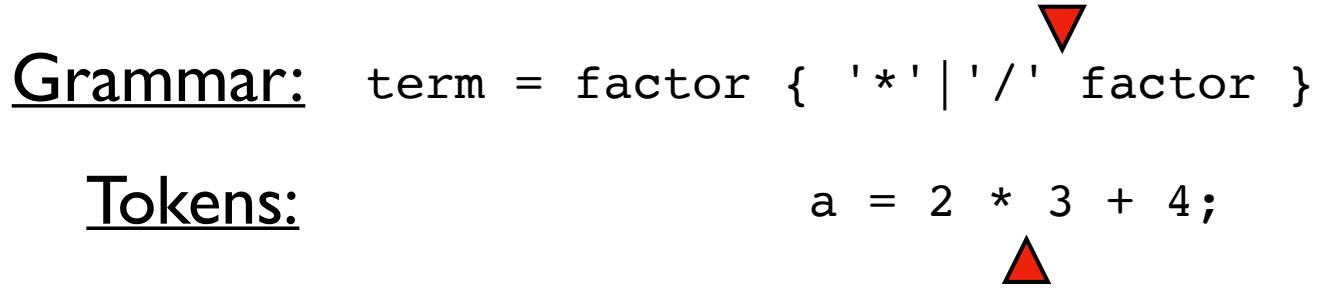
- Parsing involves stepping through tokens



- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress

Parsing Illustrated

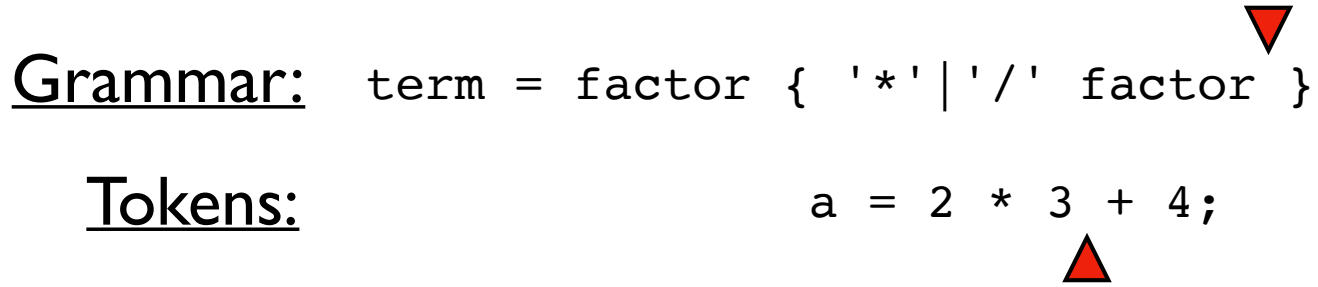
- Parsing involves stepping through tokens



- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress

Parsing Illustrated

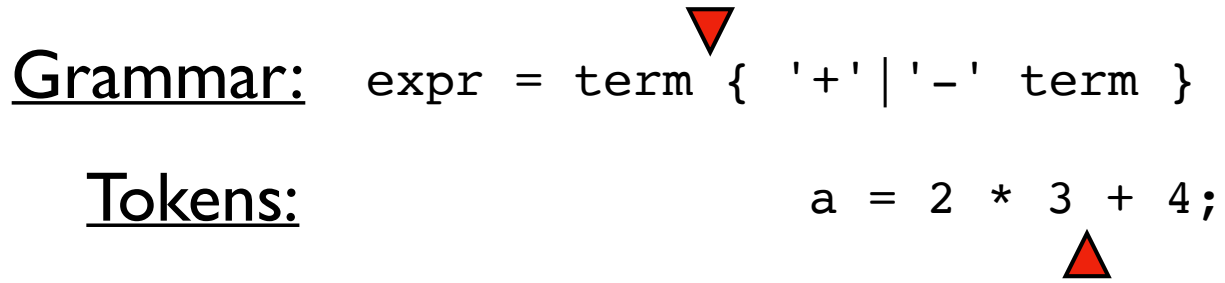
- Parsing involves stepping through tokens



- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress

Parsing Illustrated

- Parsing involves stepping through tokens



- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress

Parsing Illustrated

- Parsing involves stepping through tokens

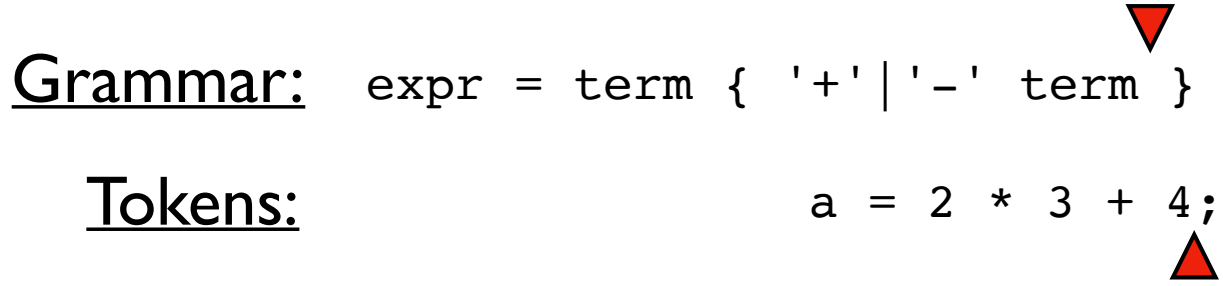
Grammar: `expr = term { '+'|'-' term }`

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

- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress

Parsing Illustrated

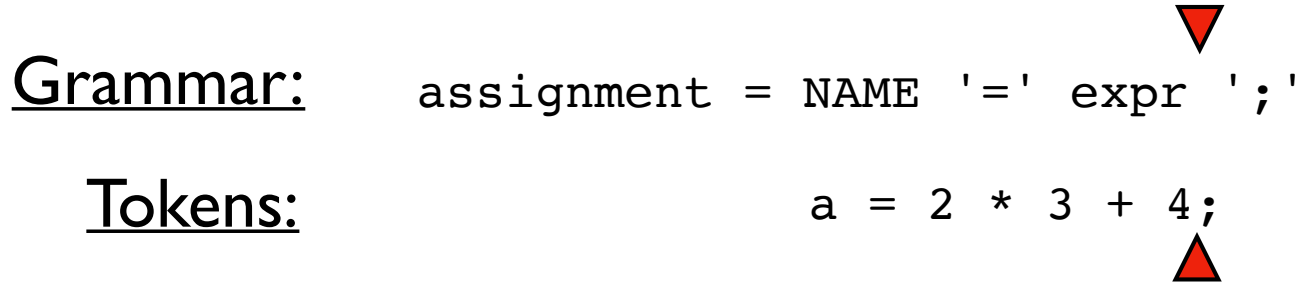
- Parsing involves stepping through tokens



- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress

Parsing Illustrated

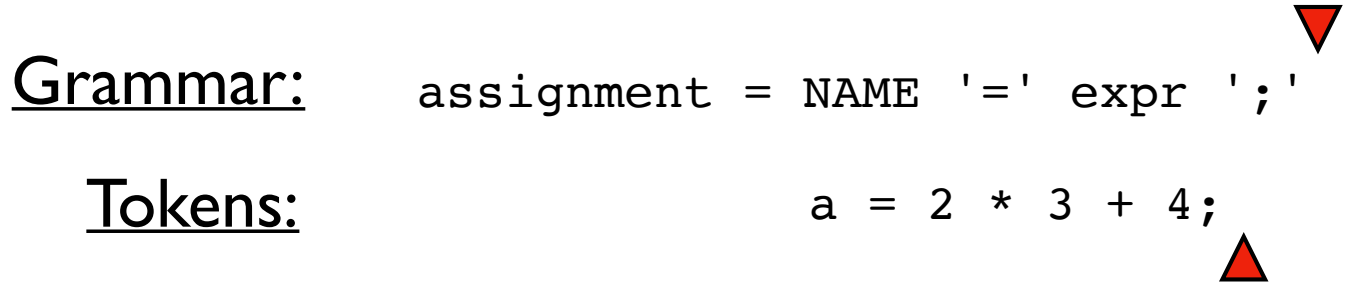
- Parsing involves stepping through tokens



- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress

Parsing Illustrated

- Parsing involves stepping through tokens



- You try to match things up as you go
- Matching descends into grammar rules
- The goal is to make forward progress
- You made it! A successful parse.

Commentary

- 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.

Project 8

Find the file `wabbit/parse.py`

Follow instructions inside.