Write a Compiler

David Beazley
http://www.dabeaz.com

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 (bits)

```
01011111111111111011101000001111
11000111110000001101110111011111
01111101111110111101111111111111
11011111110000000000000000000000
11000000000000010000000000000000
00000000000000010000000000000000
01111100011110011001010110010000
10000001100000011111111111111111
11000000000000000101101101010101
01000000000000000000000001000001
01000000010100000000000000000000
```

"Metal"

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
```

$\longrightarrow$

Machine Code

```
010111111111111110111010000000111
110001111100000011011101110111
011111011111101111011111111111
110111111000000000000000000000
110000000000001000000000000000
000000000000001000000000000000
011111000111100110010101100100
100000011000000111111111111111
110000000000000000101101101010
010000000000000000000000100000
010000001010000000000000000000
```

"Human" readable
machine code

High Level Programming

Source Code

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

"Human understandable"
programming

```
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
```

Compilers

Source Code

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

compiler →

Executable

.exe

run →

Compiler: A tool that translates a high-level program into a lower-level program

Demo: C Compiler

```c
#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;
}
```
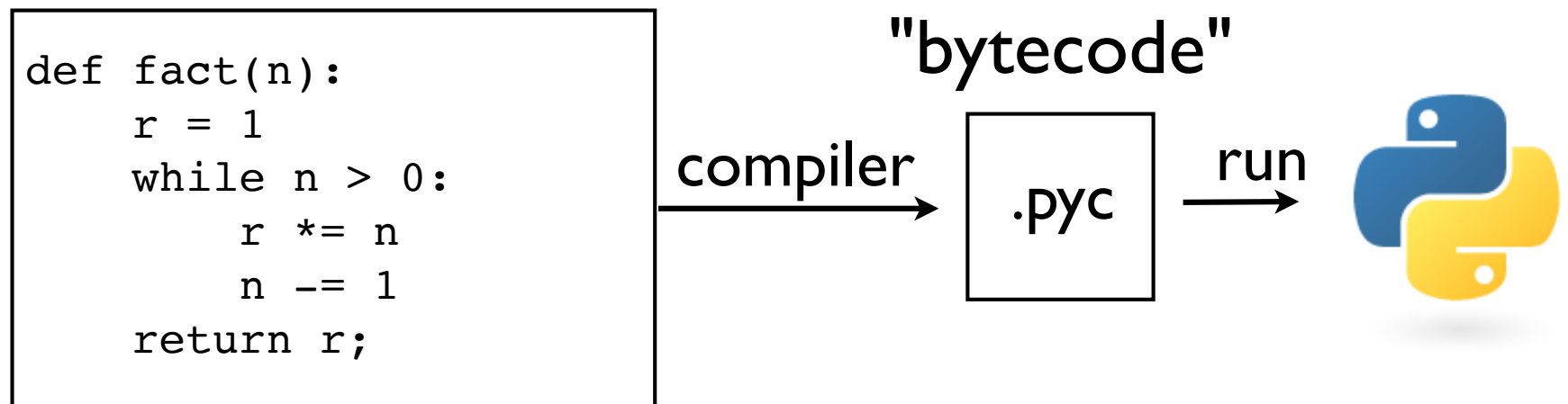
```
shell % cc fact.c
shell % ./a.out
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
shell %
```

Virtual Machines

Source Code

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

"bytecode"

compiler → .pyc → run

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;
}
```

translate →

Source Code

```
def fact(n):
    r = 1
    while n > 0:
        r *= n
        n -= 1
    return r
```
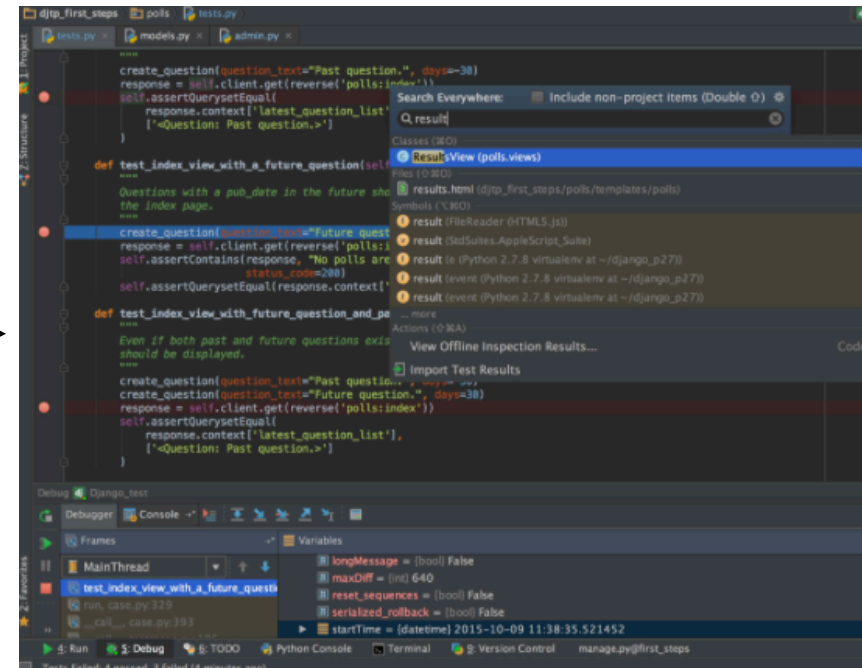
- Translation to a different language

- 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;
}
```
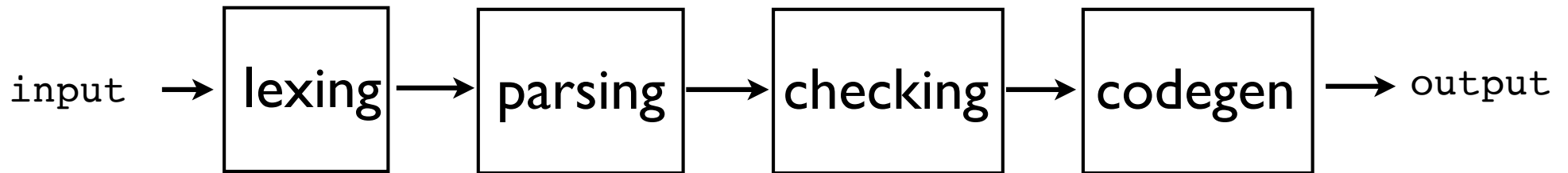
checking/
analysis

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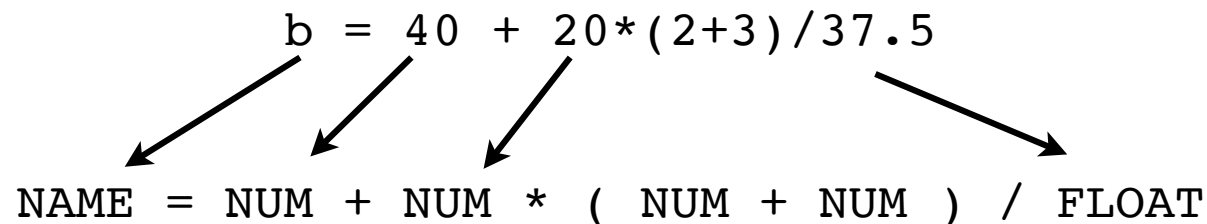
- Refactoring, IDE tool-tips, etc.

Behind the Scenes

- Classic compiler architecture is a workflow

input → | lexing | → | parsing | → | checking | → | codegen | → output

- Each stage is responsible for a different problem.

Lexing

- Splits input text into words called tokens

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

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

- Identifies valid words, detects illegal input

```
b = 40 * $5
```

Illegal Character

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

Parsing

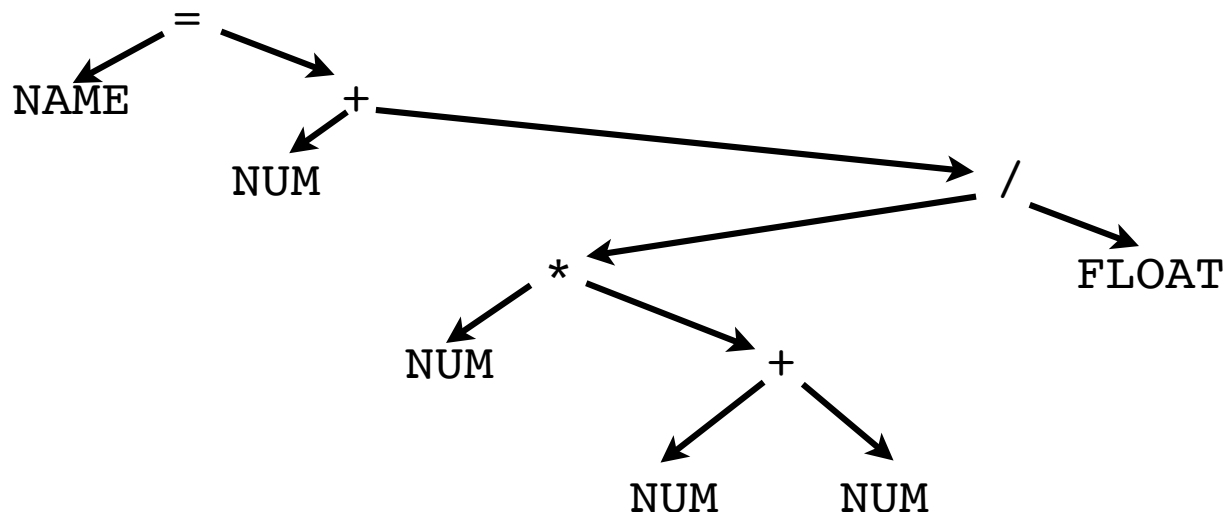"A ship shipping ship shipping shipping ships"

Parsing

- Verifies that input is grammatically correct

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

- Builds a data structure representing the input

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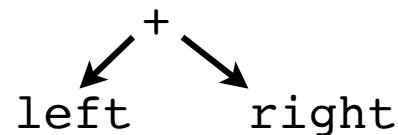
Type Checking

- ## Enforces 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

```
                    +
                ↙       ↘
            left        right
```

```
1. Left and right must be compatible types
2. The type must implement +
3. The result type is the same as both operands
```

Code Generation

- Generation of "output code":

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

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"
```

- Many possibilities.

Project Demo

Heresy!

- Many compiler courses are taught in a narrative that follows the workflow of a compiler

- Lexing -> Parsing -> Checking -> CodeGen

- Each stage builds upon the previous stage

- I am <u>NOT</u> going to follow that path

- Instead: The "Star Wars" narrative

"WHAT is happening?!?!?"

understanding the
problem

{

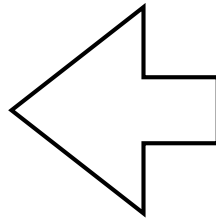| Programming |

– `Data Model`
– `Evaluation`
– `Semantics`

understanding the
problem
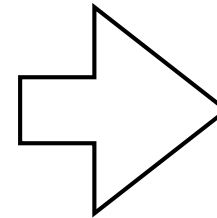
{

Programming

- `Data Model`
- `Evaluation`
- `Semantics`

parsing ⟸ **rest of course** ⟹ code generation

Making Progress

- Parts of the project are tricky

- It's not always necessary to solve all problems at once

- You're not being graded on "coding style."

- 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, global pandemics, etc.)

Common Project Fails

- <u>Testing</u>: Think about tests. Test what you can. Do <u>NOT</u> write a massive test framework.

- <u>DRY</u>: There is a lot of repetition. It may be faster to just repeat code than to figure out how to not.

- <u>Clever Code</u>: Yes, you could use metaclasses and decorators. Or you could write a compiler.

- <u>Overthinking</u>: It's easy to over-architect (i.e., OO design). Keep it simple. Refactor later.
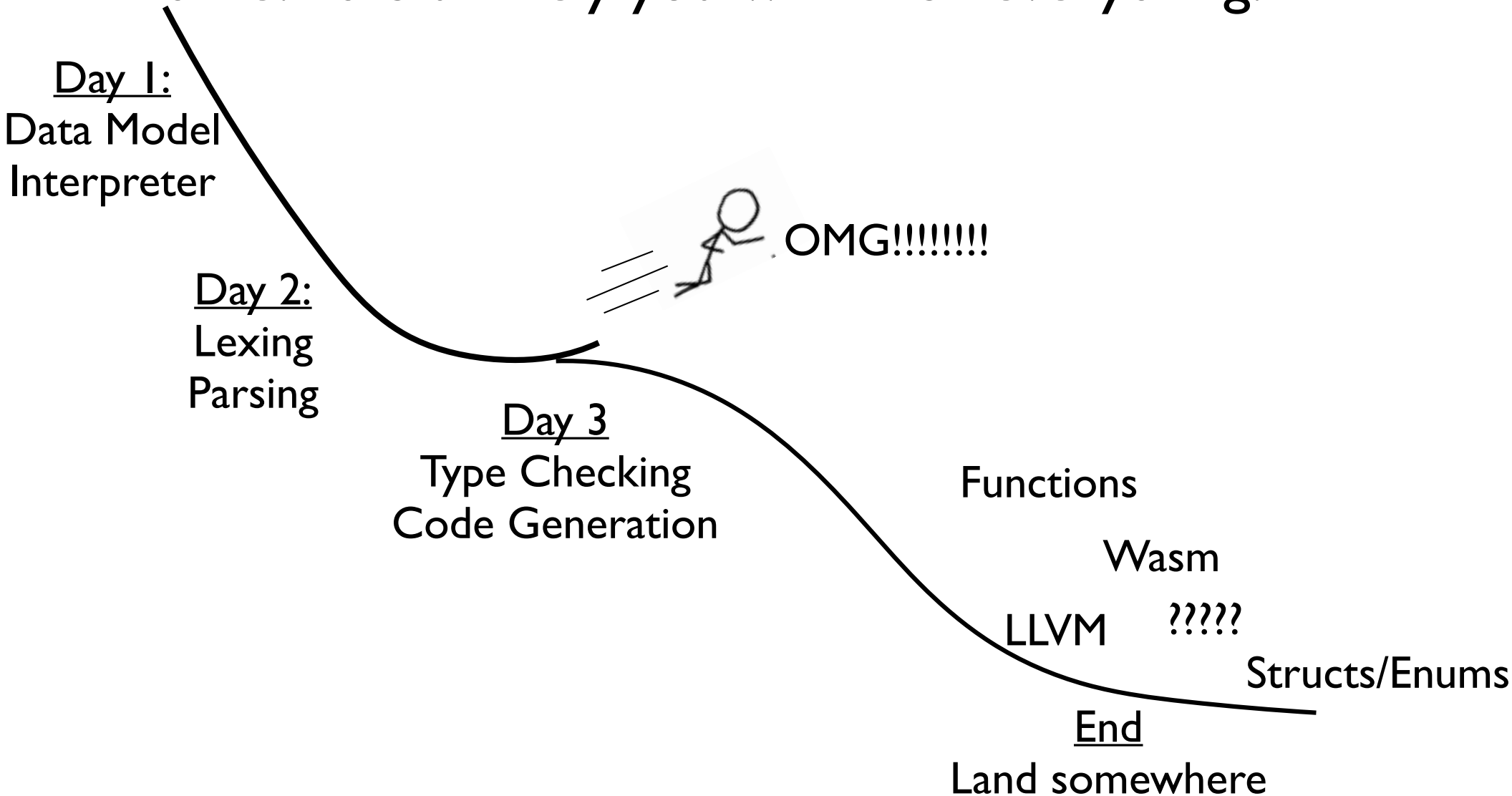
Common Project Fails

- 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.

Pick Your "Pain" Level

- The project has three levels of difficulty

    - "Script"

    - "Functions"

    - "Types"

- Plus an optional bonus level

    - Implement in a different language

A Final Note

- The project is designed to keep you busy the entire time. It is unlikely you will finish everything.

Day 1:
Data Model
Interpreter

Day 2:
Lexing
Parsing

OMG!!!!!!!!

Day 3
Type Checking
Code Generation

Functions

Wasm

LLVM   ?????

Structs/Enums

End
Land somewhere

Let's Write a Compiler ...