Write a Compiler

David Beazley
http://www.dabeaz.com

Deep Thought

Programming

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

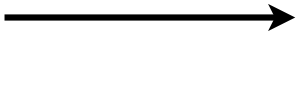
"Metal"

How does it all work????

Metal

Machine Code (bits)

```
01011111111111101110100000011111
11000111110000001101110111011111
01111101111110111101111111111111
11011111110000000000000000000000
11000000000000010000000000000000
00000000000000010000000000000000
01111100011110011001010110010000
10000000110000001111111111111111
11000000000000000010110110101010
01000000000000000000000001000001
01000000101000000000000000000000
```



"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
```

Machine Code

```
01011111111111101110100000111
11000111110000011011101111011
01111101111110111101111111111
11011111100000000000000000000
11000000000000100000000000000
00000000000000100000000000000
01111100011110011001010110010
10000000110000001111111111111
11000000000000000010110110101
01000000000000000000000100000
01000000101000000000000000000
```

"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 → .exe → run

Executable

Compiler: A tool that translates a high-level program into bits that interpret the program

Demo: C Compiler

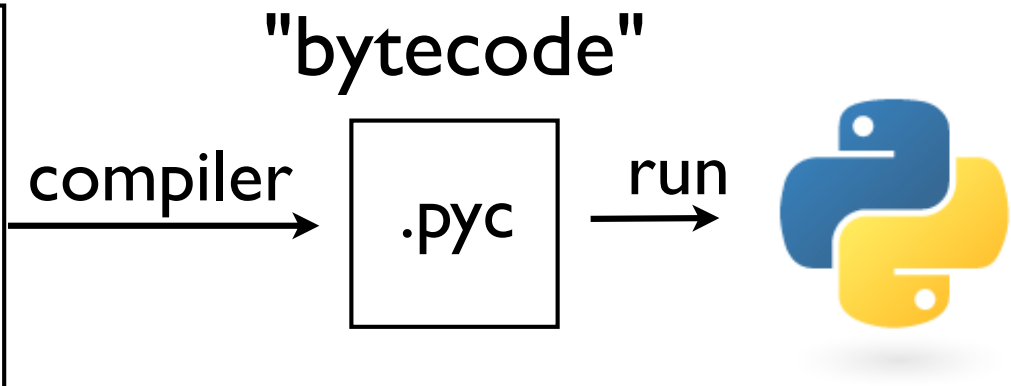
```
#include <stdio.h>

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;
}
```

```
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;
```



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
```

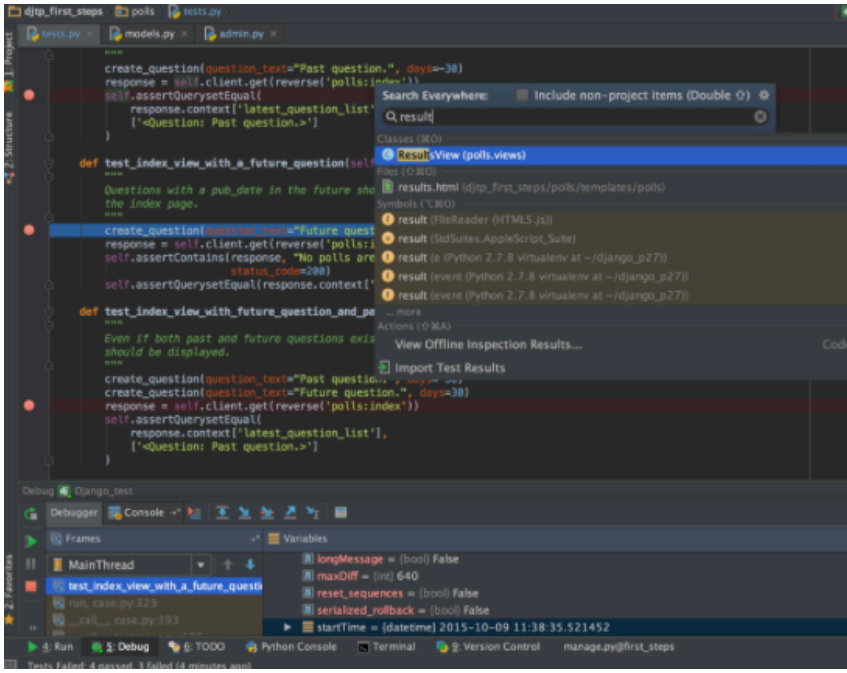
- Translation to a different language
- Example: Compilation to Javascript, C, etc.

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.

Background

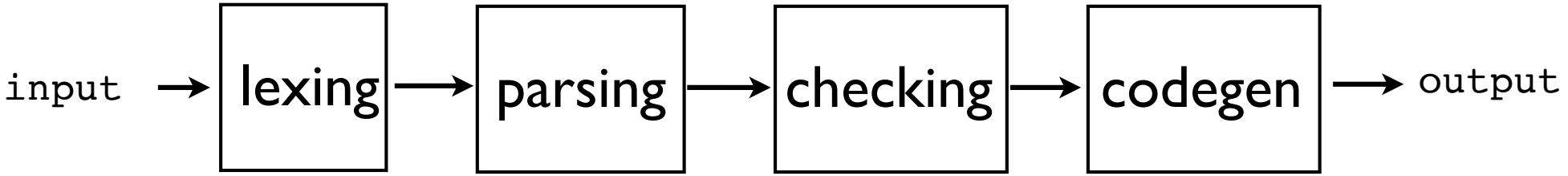
- Compilers are one of the most studied core topics in computer science
- Mathematical theory
- Interesting algorithms
- Programming language design/semantics
- The nature of computation itself

Compiler Writing is Fun!

- It's a complex programming project
- Many layers of abstraction (and often tooling)
- Involves just about every topic in computer science (algorithms, hardware, testing, etc.)
- "Hey, I wrote a compiler!!!!"

Behind the Scenes

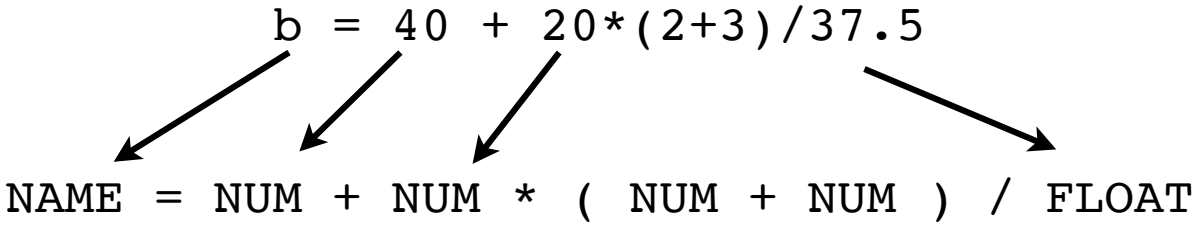
- Classic compiler architecture is a workflow



- Constructed as a series of translation stages

Lexing

- Splits input text into words called tokens



- Identifies valid words, detects illegal input

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

Parsing

"A ship shipping ship shipping shipping ships"

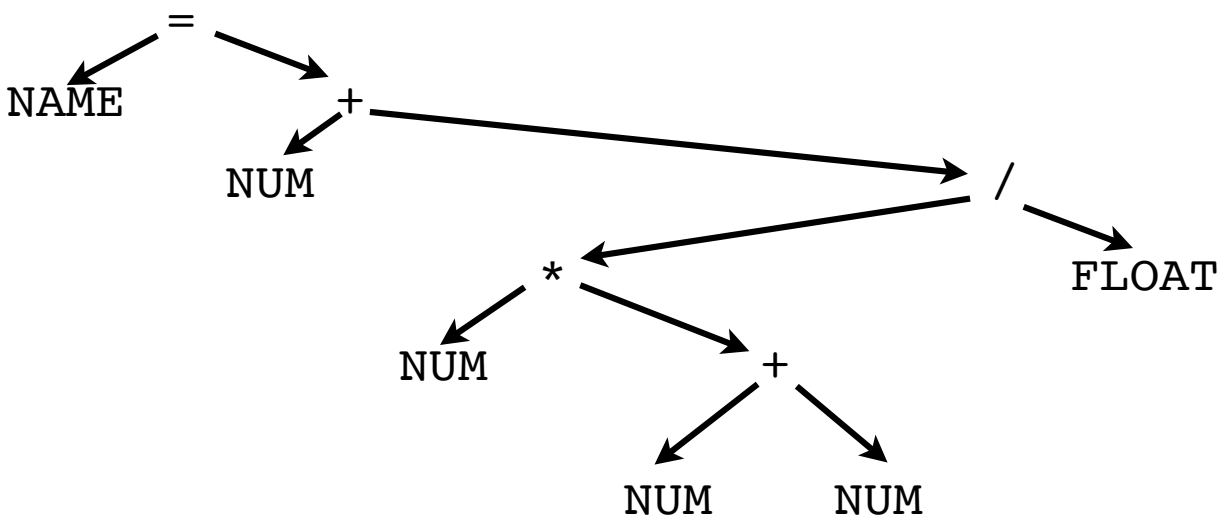
Parsing

- 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
```

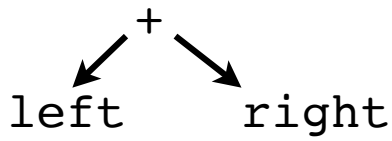


Type Checking

- 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



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
```

```
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”
```

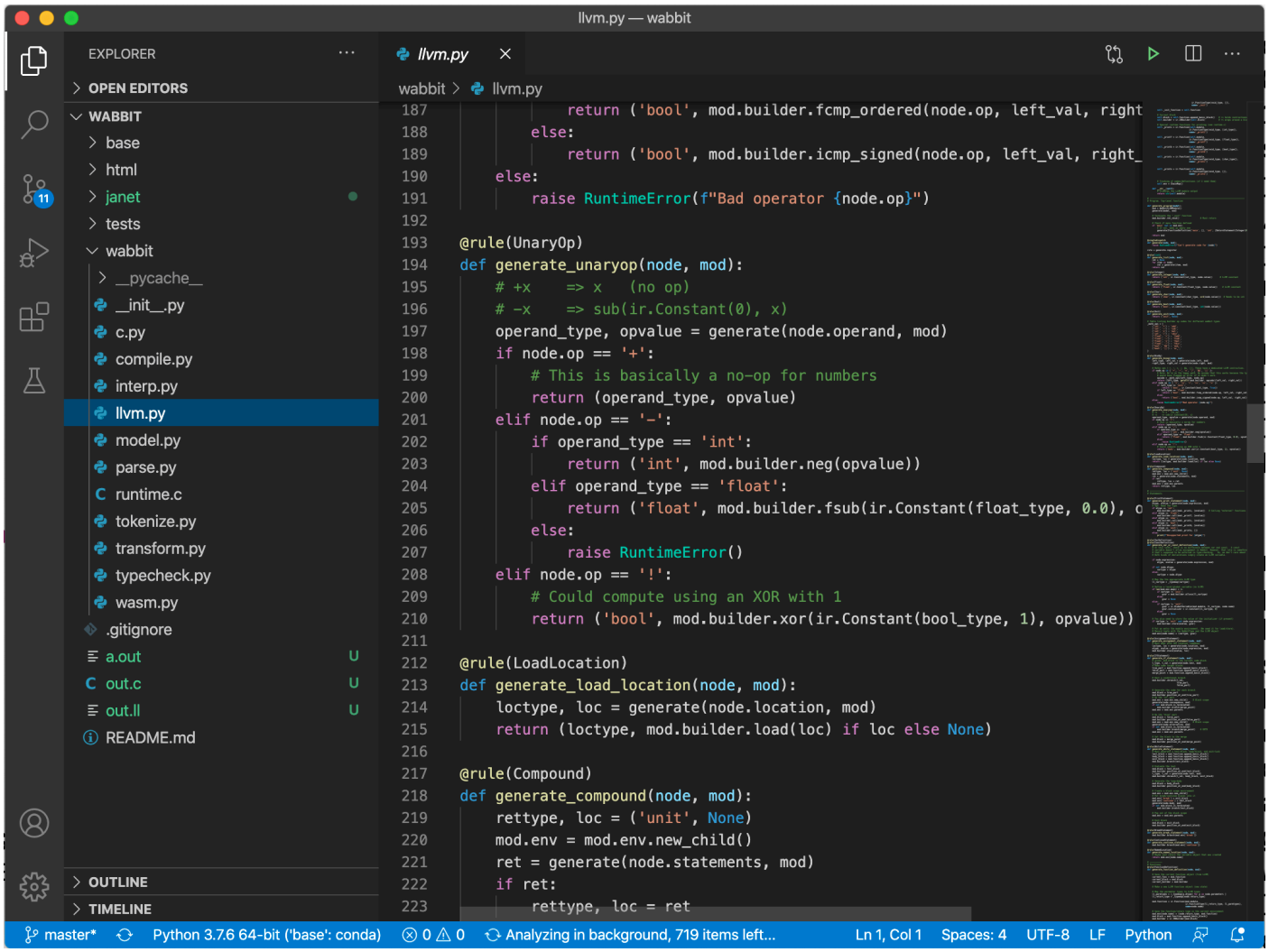
- Many possibilities.

In Reality...

- Compilers for modern programming languages might involve a significant number of stages
- Example: Scala has 25 separate steps

Modern Compilers

- Compiler architecture is evolving with IDEs



- Example: Language Server Protocol (LSP)

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 NOT going to follow that path
- Instead: The "Star Wars" narrative

"WHAT is happening?!?!?"

Now

Day 1

understanding the problem

{

Programming

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

Now

Day 1

understanding the problem

Programming

- Data Model
- Evaluation
- Semantics

parsing ⟵ rest of course ⟶ code generation

Caution

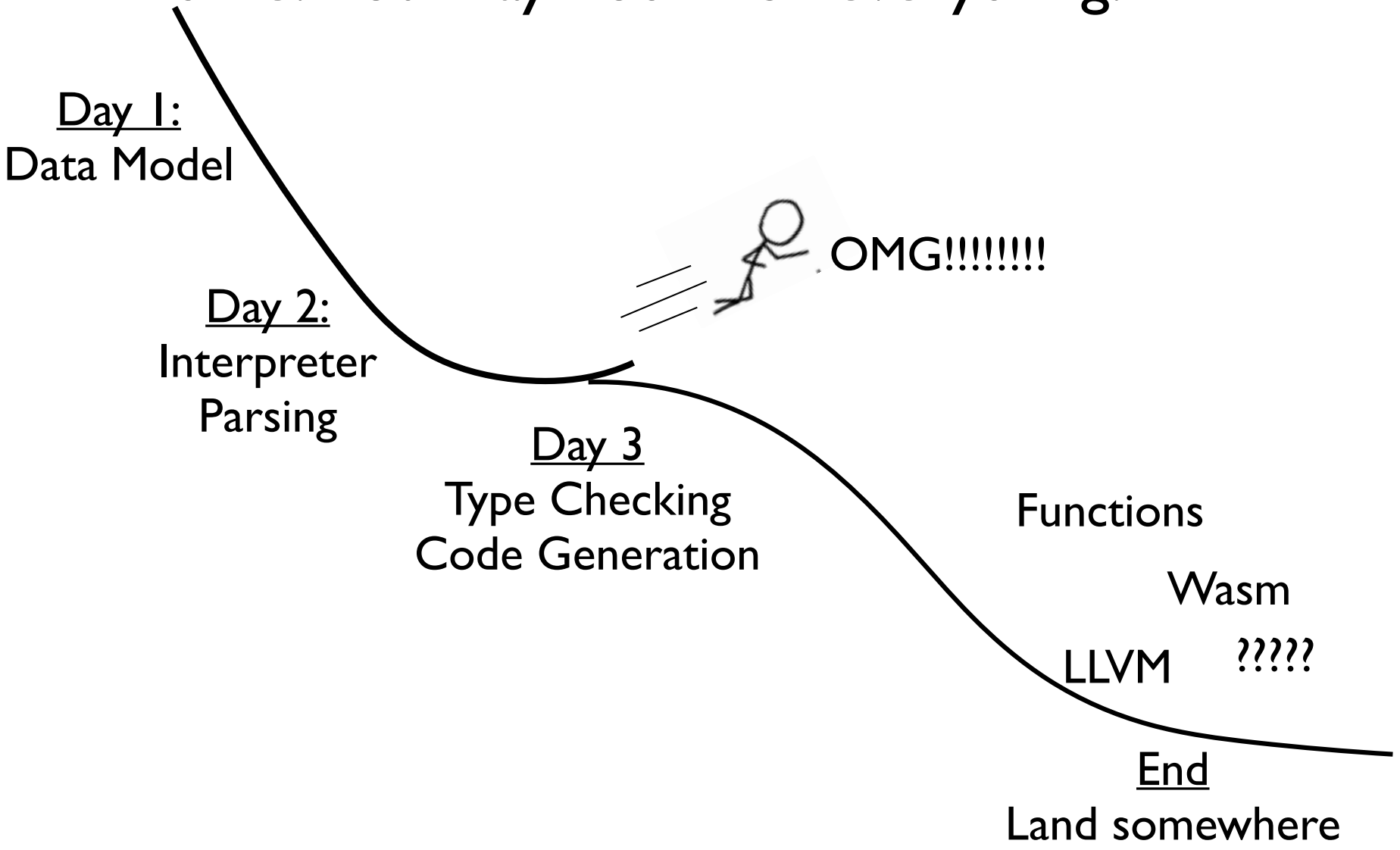
- For success, you need as few distractions as possible (work, world cup, child birth, global pandemics, etc.)

Concepts vs. Tools

- The project focuses on concepts
- Almost everything is written from scratch
- Yes, we could use tools to help, but plugging things into a tool isn't as satisfying.

A Final Note

- The project is designed to keep you busy the entire time. You may not finish everything.



Let's Write a Compiler ...