

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)

```
010111111111111110111010000001111
11000111110000001101110111011111
01111101111111011110111111111111
11011111110000000000000000000000
11000000000000001000000000000000
00000000000000000100000000000000
01111100011110011001010110010000
10000000110000001111111111111111
110000000000000000001011011010101
010000000000000000000000001000001
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
```

Machine Code

01011111111111111011101000000111
1100011111000000110111011101111
0111110111111101111011111111111
1101111111100000000000000000000
1100000000000000100000000000000
0000000000000000001000000000000
0111110001111001100101011001000
1000000011000000111111111111111
1100000000000000000001011011010
0100000000000000000000000010000
0100000001010000000000000000000



"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 bits that interpret the program

Demo: C Compiler

```
#include <stdio.h>

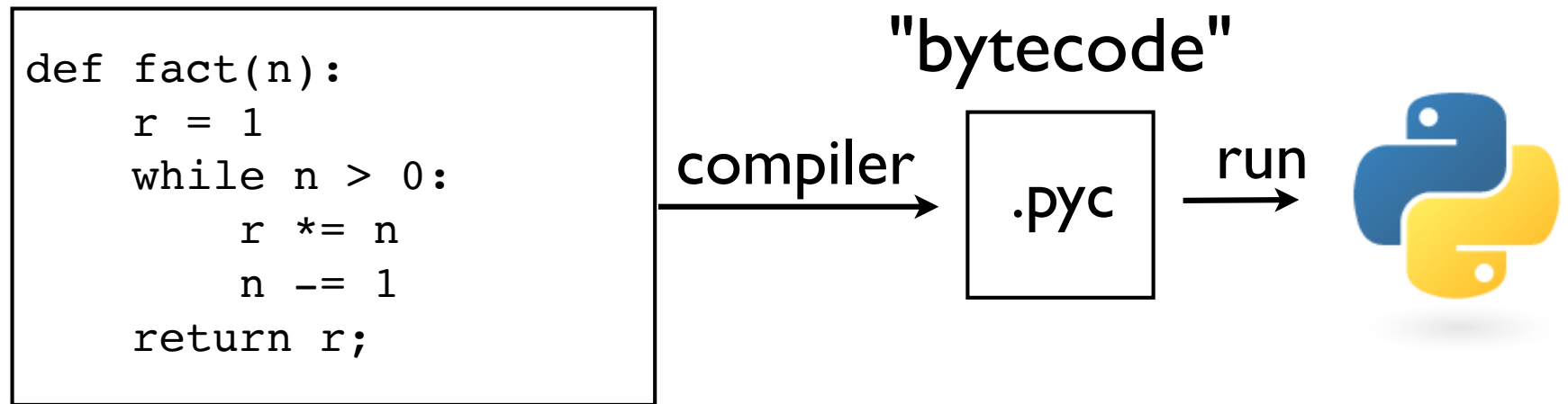
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}

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



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)  
...
```

Tip: Compiler Explorer

<https://godbolt.org>

The screenshot displays the Compiler Explorer interface in a web browser. The left pane shows the C++ source code for a function named `square`. The right pane shows the corresponding x86-64 assembly code generated by gcc 12.2. The bottom status bar indicates the compilation was successful and provides performance metrics.

Source Code (C++):

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

Assembly Code (x86-64 gcc 12.2):

```
1 square(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul    eax, eax
7     pop     rbp
8     ret
```

Status Bar: Output (0/0) x86-64 gcc 12.2 - 119ms (2811B) ~170 lines filtered

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
```

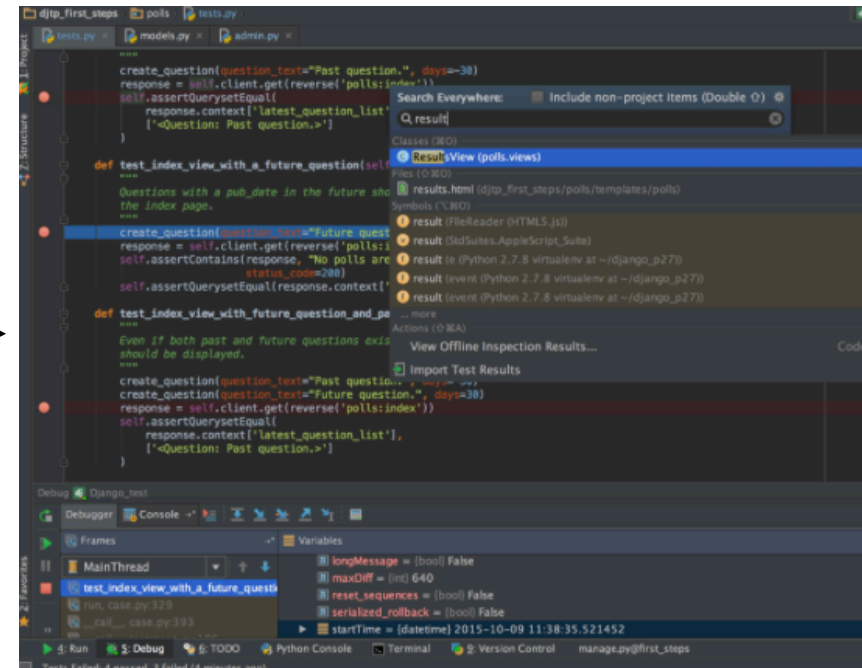
- Translation to a different language
- Example: Compilation to javascript, C, etc.

Other Tooling

Source Code

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

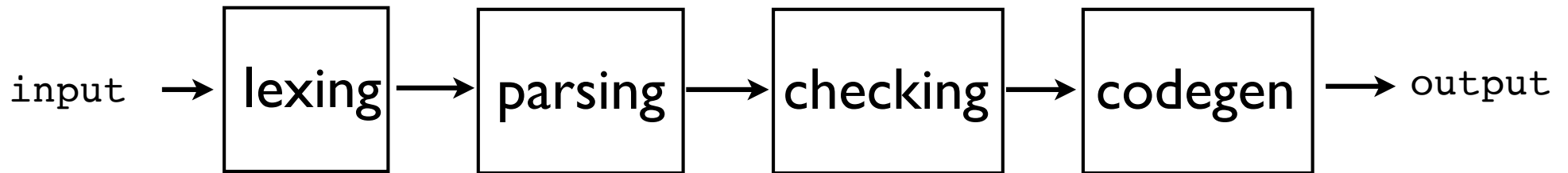
checking/
analysis



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

Behind the Scenes

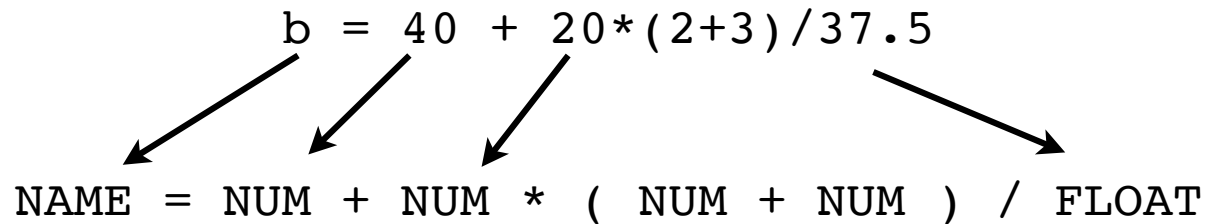
- Classic compiler architecture is a workflow



- A series of code translation stages

Lexing

- Splits source text into words called tokens



- 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"

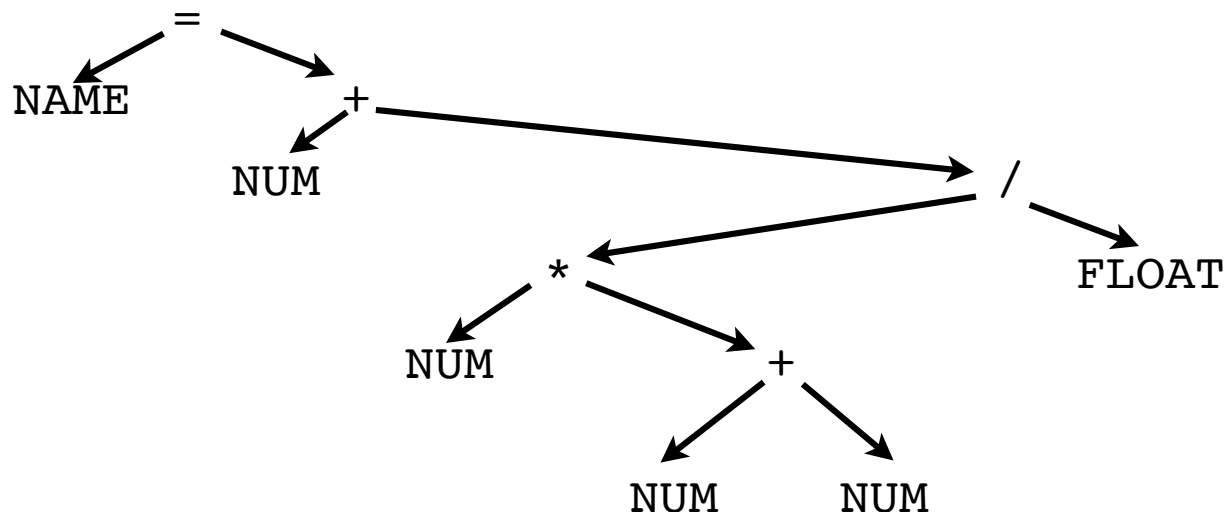
Parsing

- 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`

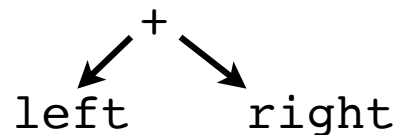


Type Checking

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

<code>b = 40 + 20*(2+3)/37.5</code>	(OK, Maybe?)
<code>c = 3 + "hello"</code>	(TYPE ERROR)
<code>d[4.5] = 4</code>	(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.

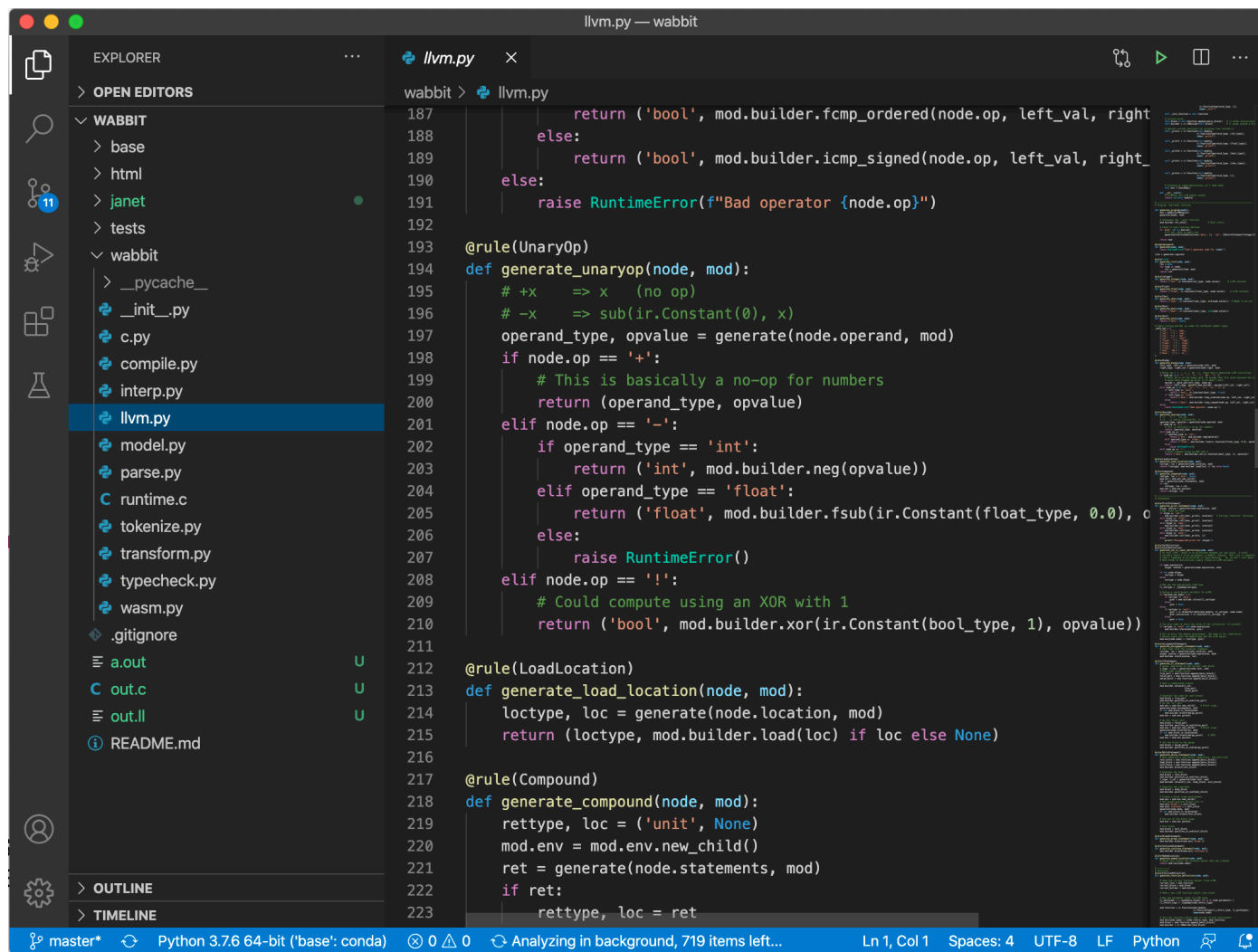
In Reality...

- Compilers for modern programming languages might involve a significant number of phases
- Example: Scala has more than 90 phases

`(scalac -Xshow-phases)`

Modern Compilers

- Compiler architecture is evolving with IDEs



```
187         return ('bool', mod.builder.fcmp_ordered(node.op, left_val, right_val))
188     else:
189         return ('bool', mod.builder.icmp_signed(node.op, left_val, right_val))
190     else:
191         raise RuntimeError(f"Bad operator {node.op}")
192
193 @rule(UnaryOp)
194 def generate_unaryop(node, mod):
195     # +x => x (no op)
196     # -x => sub(ir.Constant(0), x)
197     operand_type, opvalue = generate(node.operand, mod)
198     if node.op == '+':
199         # This is basically a no-op for numbers
200         return (operand_type, opvalue)
201     elif node.op == '-':
202         if operand_type == 'int':
203             return ('int', mod.builder.neg(opvalue))
204         elif operand_type == 'float':
205             return ('float', mod.builder.fsub(ir.Constant(float_type, 0.0), opvalue))
206         else:
207             raise RuntimeError()
208     elif node.op == '!':
209         # Could compute using an XOR with 1
210         return ('bool', mod.builder.xor(ir.Constant(bool_type, 1), opvalue))
211
212 @rule(LoadLocation)
213 def generate_load_location(node, mod):
214     loc_type, loc = generate(node.location, mod)
215     return (loc_type, mod.builder.load(loc) if loc else None)
216
217 @rule(Compound)
218 def generate_compound(node, mod):
219     ret_type, loc = ('unit', None)
220     mod.env = mod.env.new_child()
221     ret = generate(node.statements, mod)
222     if ret:
223         ret_type, loc = ret
```

- Example: Language Server Protocol (LSP)

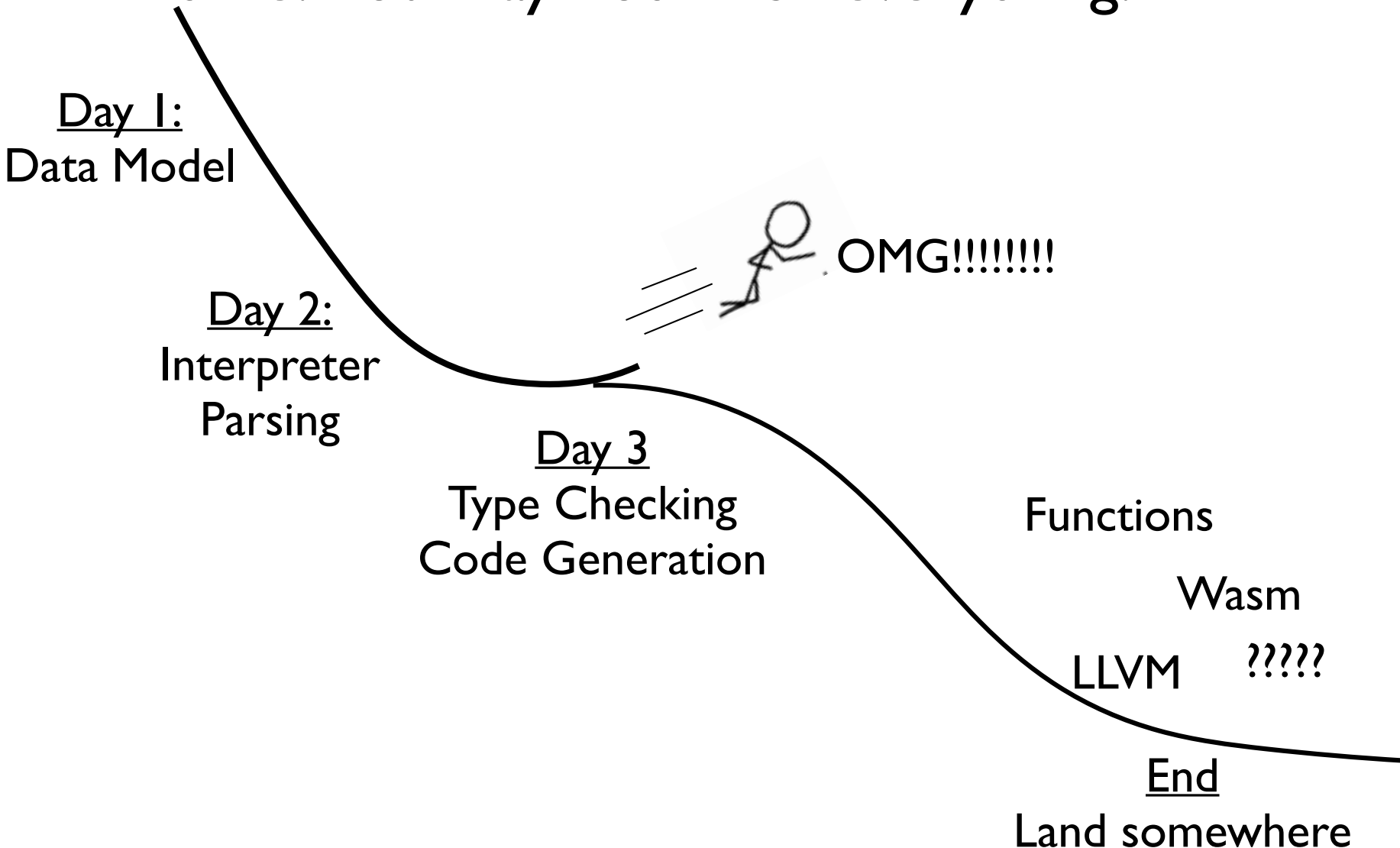
Project Demo

Overview

- We will write a basic compiler
- This project focuses on concepts
- Everything is written from scratch
- Goal: Gain a better understanding
- It's really just a starting point

A Final Note

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



Let's Write a Compiler ...