Part 2

The Evaluation of Programs

Structure vs. Evaluation

- The data model describes the structure

```
23 + 4.5  ──→  BinOp('+', Integer(23), Float(4.5))
```

- Evaluation is about how a program executes
- Semantics

Your Intuitions

- You, as a programmer, have certain intuitions about how programs work
- At least I hope so...
- Let's start with that.

Literal Values

- Literals

```
2
2.3
'c'
```

- Literals just "are"
- They don't do anything other than exist
- But, they have a value and a type

Expressions

- Example: A binary operator (+)

```
left + right
```

- Evaluate each side first, then add results

```
(2*3) + (4*5)
  ↘
    6 + (4*5)
          ↙
    6 + 20
      ↓
     26
```

- It's a recursive process ("show your work!")
- The final result is a <u>value</u>.

Names/Variables

- Names refer to objects in an "environment"

```
const pi = 3.14159;
var r float = 2.0;
var a = pi * r * r;
```

```
{
    'pi': 3.14159,
    'r': 2.0,
    'a': 12.56636
}
```

- An environment is a place to store things
- Two operations: load/store

```
a = 2.0 * pi;       // Load from "pi". Store to "a"
```

Statements

- Statements execute one after another

```
result = result * n;
n = n - 1;
print result;
```

- Each statement usually causes some kind of change in the environment (variables, I/O, etc.)
- "Imperative programming"

Conditionals

- if-statement presents two evaluation routes

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

- You evaluate the test first (a < b)
- Then, only one branch executes

Loops

- Repeated evaluation of statements

```
while n > 0 {
   result = result * n;
   n = n - 1;
}
```

- You evaluate the test first (n > 0)
- If true, evaluate the body and repeat.

Functions

- Consider a function

```
func sum_squares(x int, y int) int {
    return x*x + y*y;
}
```

- You evaluate arguments. Then the body

```
sum_squares(2+3, 4+5)

sum_squares(5, 4+5)

sum_squares(5, 9)

          5*5 + 9*9

           25 + 9*9

           25 + 81

             106
```

Note: This is not the only way to do it, but most "normal" programming languages work like this.

"Applicative Order"

- Terminology: "Function Application"

Operation Sequencing

- Digression: The order in which operations occur and how values propagate is a major topic of interest in programming languages

```
sum_squares(2+3, 4+5)

sum_squares(5, 4+5)

sum_squares(5, 9)

        5*5 + 9*9

         25 + 9*9

         25 + 81

           106
```

Steps

```
1. r1 = 2 + 3
2. r2 = 4 + 5
3. r3 = r1 * r1
4. r4 = r2 * r2
5. r5 = r3 + r4
```

- There is a direct mapping of the steps to machine instructions

Environment Model

- Understanding the environment is critical
- Programming languages have scoping rules
- These rules dictate the visibility of names

Environments/Scopes

- Definitions are part of environments (scopes)

globals

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

locals

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

- Scopes are nested (e.g., notion of "locality").

Scope Implementation

- Can implement as chained environments.

```
const pi = 3.14159;

func area(r float) float {
    a = pi * r * r;
    return a;
}

print area(4.0);
```

Locals (area)

```
{
  'r': 4.0,
  'a': 50.26544
}
```

Globals

```
{
  'pi': 3.14159,
  'area': <function>
}
```

Environment Creation

- Dynamically created during execution
- Each function call creates an environment
- Exists to store local variables
- Destroyed when the function exits
- These environments make up "stack traces"

Moving Beyond Intuition

- Yes, you have intuitions about how things "work" when you write programs
- Question: How do you turn this into a more formal specification?
- To write a compiler, you need a precise definition of how everything actually works.
- At a fine level of detail (i.e., language lawyer).

Formalizing Semantics

- One approach : Write an interpreter
- Example: Write a program that takes the data model and directly executes it.
- Sole focus: "What does the program do?"
- Sometimes known as a "definitional interpreter."

Definitional Interpreter

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

(Model)

```
def interpret_binop(node, env):
    leftval = interpret(node.left, env)
    rightval = interpret(node.right, env)
    if typeof(leftval) != typeof(rightval):
        raise TypeError()
    if node.op == '+':
        return leftval + rightval
    elif node.op == '*':
        return leftval * rightval
    ...
```

(Interpreter)

Operational Semantics

- Writing a "definitional interpreter" is an approach taken by language designers and compiler writers in the real world
- They just don't use Python (not usually)
- There is also a mathematical notational that gets used for a similar purpose

Example:

- Semantics of a conditional

$$(\text{E-IFTRUE}) \quad \texttt{if true then } t_2 \texttt{ else } t_3 \longmapsto t_2$$

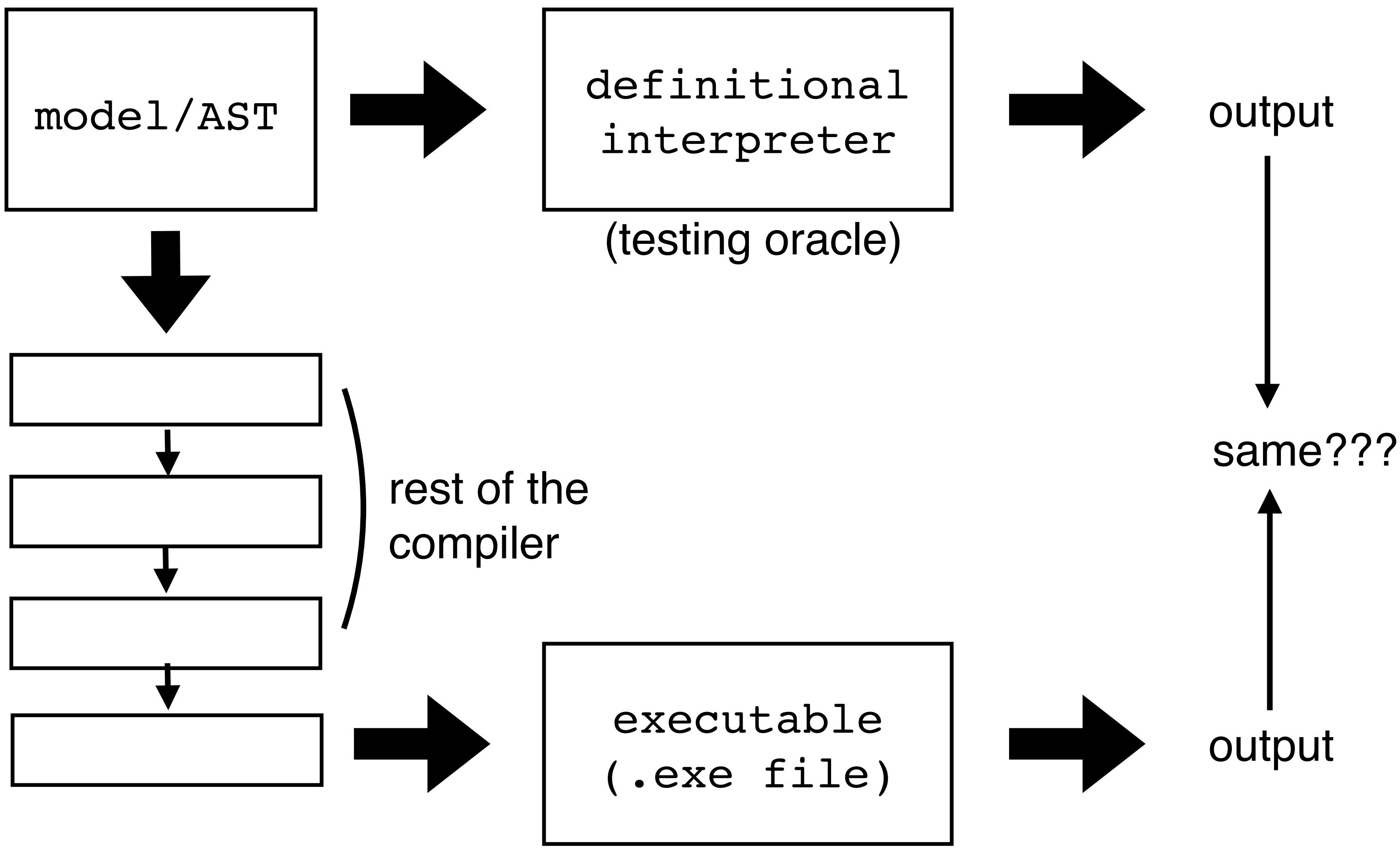
$$(\text{E-IFFALSE}) \quad \texttt{if false then } t_2 \texttt{ else } t_3 \longmapsto t_3$$

$$(\text{E-IF}) \quad \frac{t_1 \longmapsto t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longmapsto \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$$

- This is defining "small steps"
- Think of it as defining substitutions.

Big Picture: Correctness

- How do you know if a compiler works correctly?



Project

- Find the file `wabbit/interp.py`
- Follow instructions inside.
- Goal: Can we more precisely define/ understand the semantics of Wabbit by writing an interpreter that runs programs directly from the data model?