

Part 2

The Evaluation of Programs

Structure vs. Evaluation

- The data model describes the structure

`23 + 4.5` \longrightarrow `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 actually work
- At least I hope so...
- We'll start with that
- Will dig deeper later

Literal Values

- Literals

2
2.3
'c'

- Literals just "are"
- They don't do anything other than exist.

Expressions

- Example: A binary operator (+)

left + right

- Evaluate each side first, then add results

The diagram illustrates the evaluation of the expression $(2*3) + (4*5)$ through four steps, connected by downward arrows:

- Step 1: $(2*3) + (4*5)$
- Step 2: $6 + (4*5)$ (The $2*3$ part is evaluated to 6)
- Step 3: $6 + 20$ (The $4*5$ part is evaluated to 20)
- Step 4: 26 (The final result)

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

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 (includes I/O).
- "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)
```

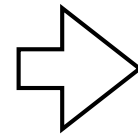
```
sum_squares(5, 9)
```

```
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

Scoping

- Environments are nested

```
const pi = 3.14159;  
  
func area(r float) float {  
    a = pi * r * r;  
    return a;  
}  
  
print area(4.0);
```

Functions can only see
variables defined in the
enclosing definition context.

(lexical scoping)

Globals

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

Locals (area)

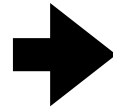
```
{  
  'r': 4.0,  
  'a': 50.26544  
}
```



Structures

- A structure is a container of data

```
struct Point {  
    x int;  
    y int;  
}  
  
p = Point(2, 3);
```



```
{  
    'x': 2,  
    'y': 3  
}
```

- An instance holds values for all fields
- They're all together in one object
- Implementation details vary (concept is same)

Moving Beyond Intuition

- Yes, we have intuitions about how things "work" when we 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 executes it directly.
- 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)



(Interpreter)

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


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

(E-IFTRUE) `if true then t2 else t3 \mapsto t2`

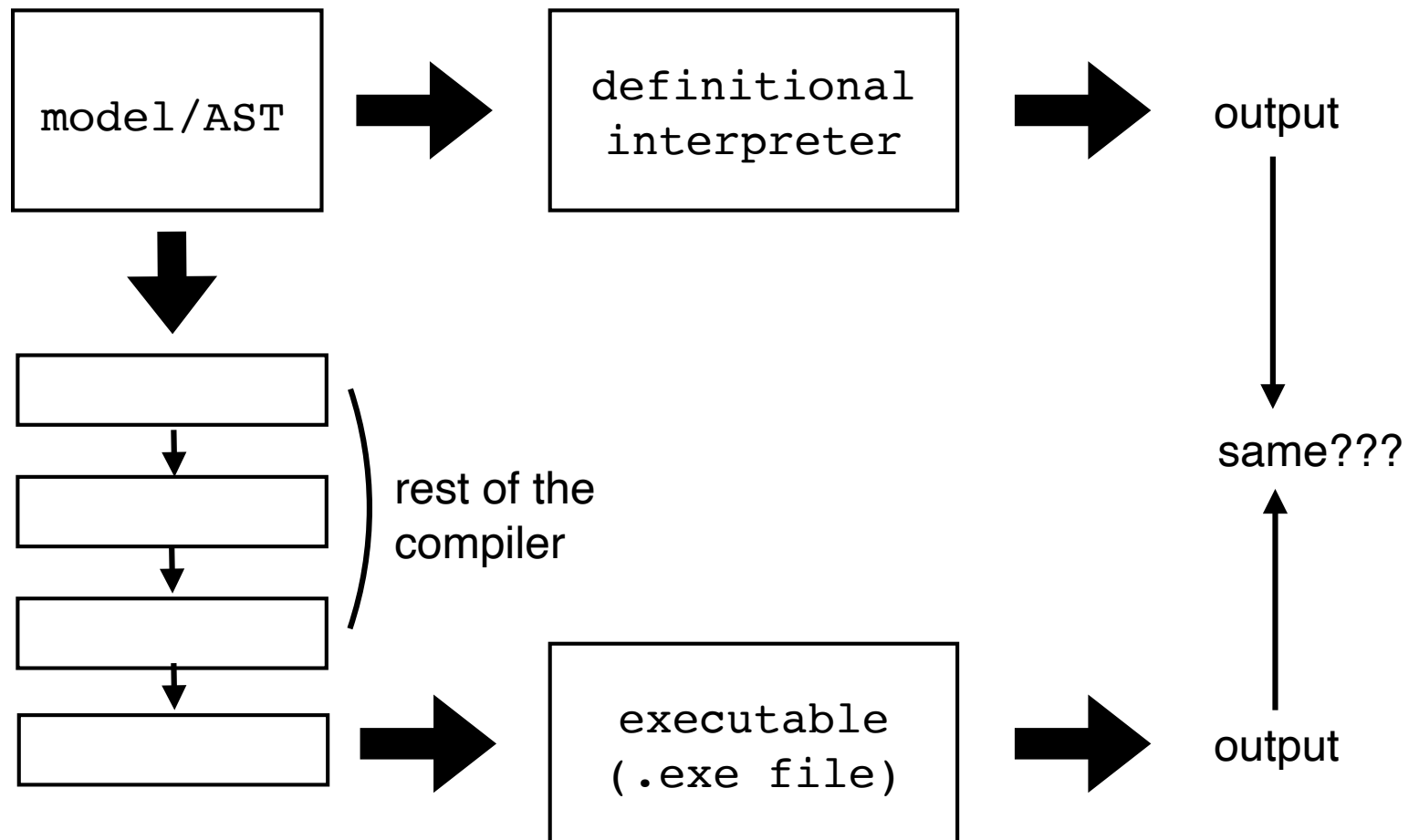
(E-IFFALSE) `if false then t2 else t3 \mapsto t3`

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

- 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?