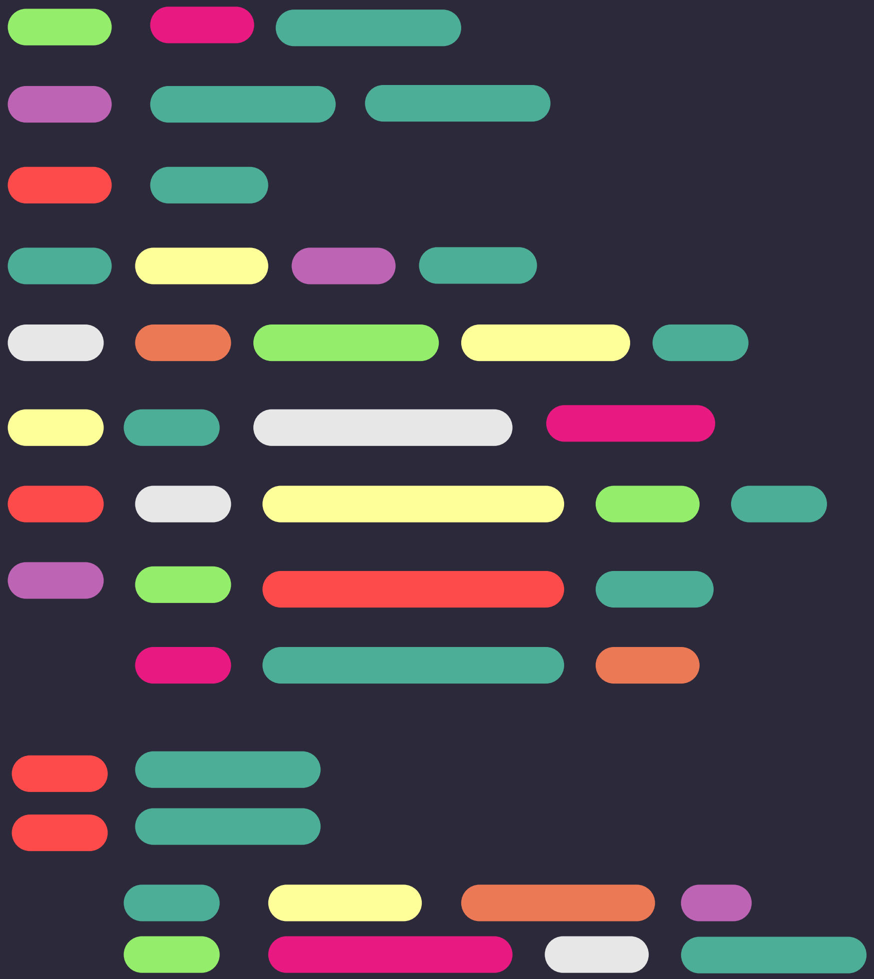
Anna-OOP

< AJ Scollo, Anthony Nguyen, Kenneth Acevedo, Creed Leichtle >

01

Introduction

Our language is an imperative, strongly-typed programming language that uses type inference. We wanted to combine the safety of strict type enforcement with the flexibility of inferred types, like languages such as Ruby and Python.



Design Choices

- Type System
    - Our goal was to combine the developer-friendly simplicity of dynamically typed languages like Python, while maintaining strong typing to easily catch possible errors.
        - Furthermore, we wanted to differentiate it from our target language, Java.
- Syntax
    - We designed our syntax to mirror the simplicity and readability of languages like Ruby and Python. However, we introduced a unique syntax to differentiate it from those languages.

```
Our language:
result = add(3,1)!     (result will be an integer)

Java:
int result = 3 + 1;    (result must be an integer)

Python:
result = 3 + 1         (result will be an integer)
```

Development Process

- We began our development process by defining the key features we wanted
  - Type inference, strongly-typed, imperative
- From there, we broke our language down into individual components
  - AJ – Input and Output, Anthony – Control Structures, Creed – Arithmetic and Logical Operations, Ken – Variable Assignment
- To maintain code organization, we decided to separate our language's functionalities into distinct statement classes.
  - AddStatement, IfStatement, FuncCallStatement, etc.
  - Ultimately, this made everything easier to manage, test, and extend
- Altered our parsing methodology from Regular Expression patterns to Tokenization

Executable Statements

- The ExecutableStatement interface is the blueprint for all statements in our language that can be executed
  - This interface allows us to handle various types of statements, from simple arithmetic to complex control structures

- The run method executes the statement using the given namespace that contains variable bindings, and returns an Object which represents the result of the statement execution
  - Ultimately, when we parse a string like `add(1,2)` we instantiate the numbers `1` and `2` as ValueStatements, and the entire expression as an AddStatement.
    - We end up with this: `new AddStatement(ValueStatement(1), ValueStatement(2))`
    - When we run this statement with `run` we return an Object with the value 3

Challenges

- Initially, we discussed using Regular Expression patterns to parse input strings and perform arithmetic operations.

```
Private static Pattern pattern = Pattern.compile("^add\\((.+),(.+)\\)$");
```

- This method proved problematic for handling nested operations like: add(mult(2,3),mult(2,3))
    - This is due to the fact that Regular Languages cannot parse Context-Free-ness #CSC-473

- This issue was resolved by adopting a “tokenization” methodology where input strings are broken down into a list of tokens, which can then be parsed and translated into our target language, Java, which we will break down in the following slides.

{

Tokenizing

- We step through each character in an input string (ignoring whitespace) to determine its TokenType (VARIABLE, NUMBER, OPERATION, etc.)
    - We do this until we reach our statement terminator `!`
- Each Token is appended to a List of Tokens which will ultimately be sent to the Parser to fully translate and execute the statement

}

Parsing

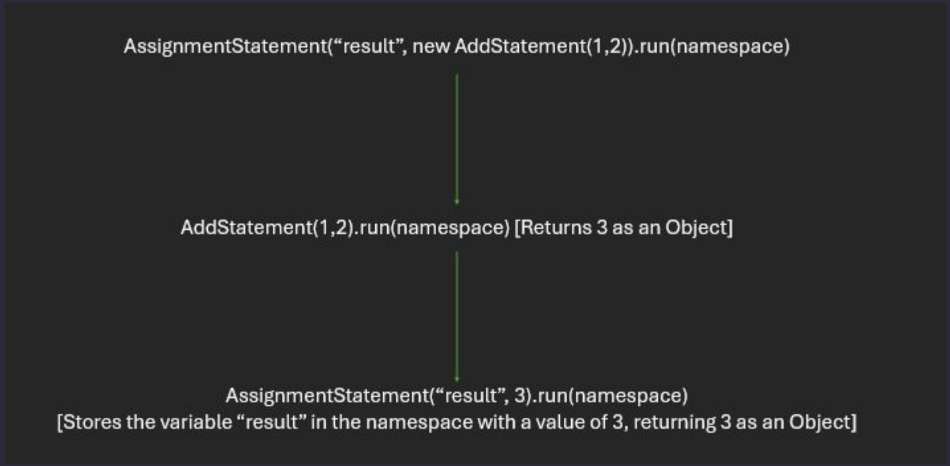
- Our Parser takes the list of Tokens and translates them into a list of `ExecutableStatements`
- We iterate over the list of Tokens instantiating the necessary `ExecutableStatements` as we go.

<u>Token List</u>
[VARIABLE: “result”]
[ASSIGN: “=”]
[OPERATION: “add”]
[LPAREN: “(“]
[NUMBER: 1]
[COMMA: “,”]
[NUMBER: 2]
[RPAREN: “)”]
[END: “!”]

<u>ExecutableStatement</u>
new AssignmentStatement(“result”, new AddStatement(1,2))

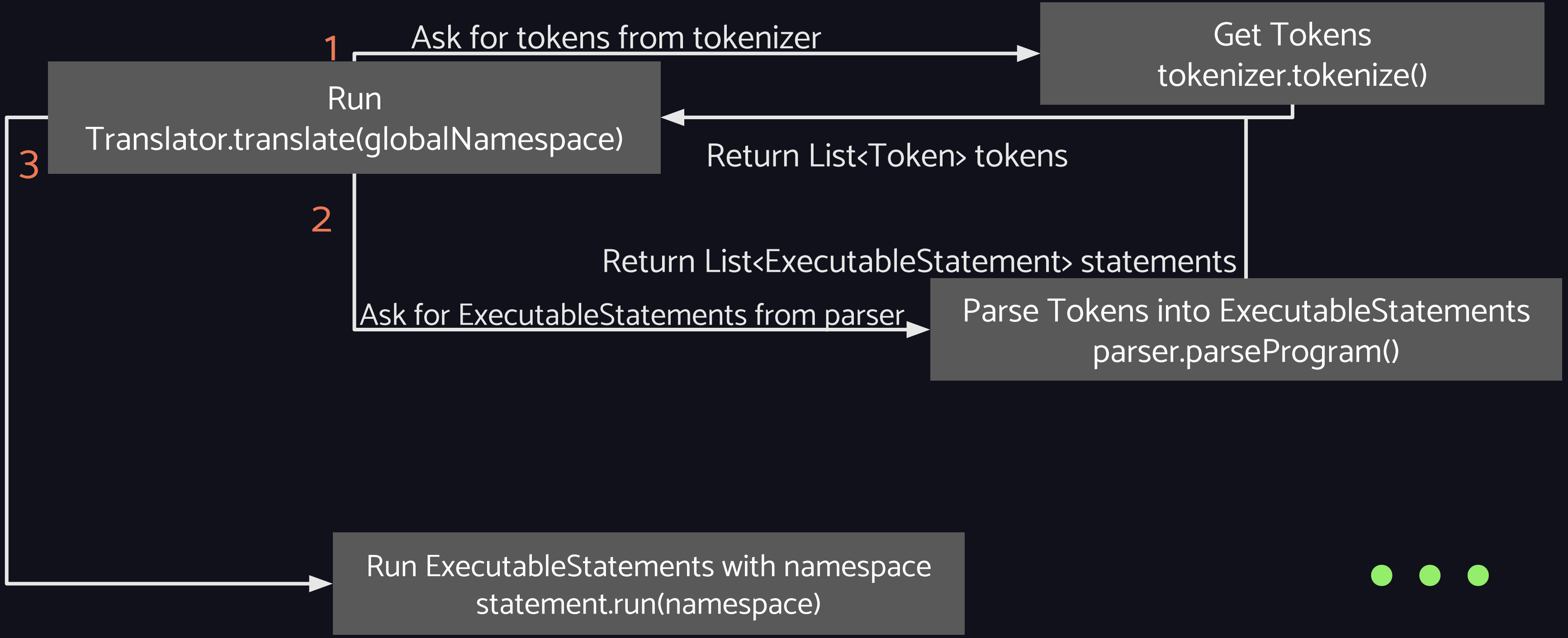
Executing

- After our Parser has created a list of ExecutableStatements, our Translator simply iterates over the list calling the `run()` method for each statement



Overview

- Here is a diagram of the process



02 { ..

Language Tutorial

< Variables, Arithmetic, Comparisons, IO,
Control Structures, and More >

} ..

Language Tutorial

Variable Assignment

```
x = 5
y = false
z = “some string”
```

Arithmetic

```
add(1,2)
add(add(1,2),3)
sub(2,1)
mult(2,1)
div(6,3)
mod(5,2)
```

Comparisons

```
equalTo(4,4)
greaterThan(5,4)
lessThan(4,5)
```

IO

```
readi()
readb()
reads()
```

{ Control Structures

Loops and If-Else Statements

```
loop <i=0> <add(i,1)>
* print(i)
done <equalTo(i,5)>!
```

```
if <bool> then
* print(“t”)
else
* print(“f”)
done!
```

}

{ Functional Programming

```
function sum(x,y,z)
* return add(x,add(y,z))
done!

function diff(x,y,z)
* return sub(x,sub(y,z))
done!

function functional(func, x, y, z)
* printn(func(x,y,z))
done!

functional(sum, 10, 3, 1)!
functional(diff, 10, 3, 1)!
```

}

03

Sample Programs

Sample Problem Showcase (Exceptions)

```
x = false
x = false!

add(5, false)!
add(5, 4.5)!

println(Hello)!
printn(“Hello”)!

str1 = “Hello ”! str2 = “World!”!
print(add(str1, str2))!
print(str1)! print(str2)!
```

```
if <true> then
print(“Hello World!”)!
if <true> then
* print(“Hello World!”)
else print(“”) done!

for <i=5> <add(i,1)> then
* if <i==4> then
** print(“Hello World!)!
loop <i=5> <add(i,1)> then
* if <equalTo(i,4)> then
* print(“Hello World!)
* else print(“”) done
done <equalTo(i,5)>!
```

Sample Problem Showcase (Program2.txt)

Written by Group 7 to accomplish the tasks laid out in Program 2

```
% Read in variables
a = readi()!
b = readi()!
m = readi()!

% Start of loop
loop <i=a> <add(i,1)>
* loop <j=0> <add(j,1)> * print("*") done
<equalTo(j,i)>
done <equalTo(i,add(b,1))>!

% Multiples Sum
sum = 0!
loop <x=1> <add(x,1)>
* if <equalTo(mod(x,a),0)> then
*   sum = add(sum,x)
* else if <equalTo(mod(x,b),0)> then
*   sum = add(sum,x)
*   else print("") done done
done <equalTo(x,m)>!

printn(sum)!
```

(Hi-Lo) Program Showcase

Too High

This is a standard game of Hi-Lo, where you guess a number.

You Win!

While the number is not random, it is still able to take user input.

Too Low

Shows a function + call, loop, if, boolean operators, and IO.

```
function hilo()
* number = 420
* loop <i=0> <add(i,1)>
     * input = readi()
     * if <equalTo(input, number)> then
          * printn("You win!")
          * return 0
     * else if <greaterThan(input, number)> then
          * printn("Too high") else
          * printn("Too low") done
     * done
* done <equalTo(i,10)>
done !

hilo()!
```

PA5 Question 5: Smallest Number with Factors up to 15

```
function smallest_positive_with(factors_up_to)
*   loop <i=factors_up_to> <add(i,1)>
    *   if <divisible_up_to(factors_up_to, i)> then
        *   printn(i)
        *   return i
    *   else print("")
    *   done
*   done <equalTo(i,100000000)>
done!
```

For numbers from 15 to 100 million, check if it has all 15 factors

```
function divisible_up_to(max, num)
*   loop <i=2> <add(i,1)>
    *   if <not(has_factor(num, i))> then
        *   return false
    *   else print("")
    *   done
*   done <equalTo(i,max)>
*   return true
done!
```

For numbers from 1 to max check if number is divisible by it

```
function has_factor(num, factor)
*   return equalTo(mod(num, factor), 0)
done!
```

Check if the number is divisible by the factor

```
smallest_positive_with(15)!
```

The program runs in a very short amount of time, and prints an output of 360,360 successfully

Additional Features

- Functions (Already mentioned, so how they can be passed as variables)
  - Recursion
- Interactive System
- Type-Checking System
- Error Handling
- Tokenization and Parsing
- Scoping System