CSCI 4020
Simple Compiler

Levi Willms, Matthew Witvoet,
Russell Ngo

Tech Stack

- ANTLR Grammar
  - Lexer
  - Parser
- Kotlin Backend Interpreter
- Environment
  - Jupyter Notebook

Dev Environment

Windows Subsystem for Linux (WSL)

- Ubuntu

Antlr4

- Antlr dependencies

Jupyter Notebook

- Kotlin Kernel

Makefile

- Extended makefile from Assignment 3 to use make build to compile our language.

Grammar Overview

Grammar: Parser and Lexer

- PL.g4 grammar file

Backend: Kotlin Interpreter

- data.kt
- expr.kt
- runtime.kt

Interpreter

- Runtime class
  - Methods...
- Data class
  - Implementations of Data
- Expr class
  - Implementations of Expr
- Type checker (in Expr class)
- Aggregate Data Values
- List Functionality

Language Features

Integer and string data

Inherited from A3:

- Assignment and dereference of variables
- Arithmetics of integers
- Concatenation of strings and integers
- Iteration over integer ranges
- Function declaration
- Function invocation
- Recursion

New

- Simple static type checker
- Aggregate data values

Data Class

- Null data
- None data
- Integers
- Strings
- Booleans
- Functions

Expr Class

- Expr (expressions)
- None Expr
- Parenthesized Expr
- Assign
- Deref
- Invoke
- Block
- FunctionDef
- Loop
- Print
- IfElse

- Arithmetic
  - Addition (+)
  - Subtraction (-)
  - Multiplication (*)
  - Division (/)
- Cmp (Compare)
  - Less than (<)
  - Less than or equal to (<=)
  - Greater than (>)
  - Greater than or equal to (>=)
  - Equal (==)
- Concat
- Type (enum)
  - NUMBER
  - STRING
  - BOOL
  - FUNC
  - NONE

Aggregate Data Types

Aggregate Values: Total value of smaller sums together

```
val program1325 = """
function setval(x){
    2;
}
function recursiontest(n){
    if(n < 2) {
    setval(1);
  } else {
    n + recursiontest(n-1);
  }
}

print(recursiontest(10));
"""
```

```
execute(program1325)
```

```
56
```

```
val program1328 = """
sum = 0;
x = 1;
y = 2;
sum = sum + x + y;

print(sum);
"""
```

```
execute(program1328)
```

```
3
```

```
val program42 = """
x = 1;
y = 2;
z = 3;
function seven(a){
    result1 = x + (y * z);
    result1;
}
function subtract(b){
    result2 = seven(1) - 5;
    result2;
}
function sum(c){
    result3 = subtract(1) + seven(1);
    result3;
}
function testloop(d){
    result4 = sum(1);
    for(i in 1..3){
        result4 = result4 + y;
    }
    result4;
}

print(testloop(1));
"""
```

```
execute(program42)
```

```
15
```

Static Type Checker

- Extended Assignment 3 to add a static type checker
- Added abstract typeCheck function to the Expr class
- All implementations of Expr override typeCheck to set validation logic
- Extended grammar to include implicit type setting

Interpreter implementation of Invoke.

```
class Invoke(
    val funcname: String,
    val arguments: List<Expr>
) : Expr() {
    override fun eval(runtime:Runtime): Data {
        val f = runtime.symbolTable[funcname]
        if(f == null) {
            throw Exception("$funcname does not exist.")
        }
        if(f !is FunctionData) {
            throw Exception("$funcname is not a function.")
        }
        if(arguments.size != f.parameters.size) {
            throw Exception("$funcname expects ${f.parameters.size} arguments, but ${arguments.size} given.")
        }

        // evaluate each argument to a data
        val argumentData = arguments.map {
            it.eval(runtime)
        }

        // create a subscope and evaluate the body using the subscope
        return f.body.eval(runtime.subscope(
            f.parameters.zip(argumentData).toMap()
        ))
    }

    override fun typeCheck(runtime: Runtime): Type {
        val f = runtime.symbolTable[funcname]
        if(f == null) {
            throw Exception("$funcname does not exist.")
        }
        if(f !is FunctionData) {
            throw Exception("$funcname is not a function.")
        }
        if(arguments.size != f.parameters.size) {
            throw Exception("$funcname expects ${f.parameters.size} arguments, but ${arguments.size} given.")
        }

        val argumentTypes = arguments.map {
            it.typeCheck(runtime)
        }

        val expectedTypes = f.parameters.map { Type.STRING }
        if(argumentTypes != expectedTypes) {
            throw Exception("Argument types do not match. Expected: $expectedTypes, but found: $argumentTypes")
        }

        return f.returnType
    }
}
```

Static Type Checker Examples

```
val program88 = """
typecheck1 = 9 + 10;
print(typecheck1(1));
"""
```

```
execute(program88)
```

```
Error: java.lang.Exception: typecheck1 is not a function.
```

```
val program89 = """
print(typecheck2(1));
"""
```

```
execute(program89)
```

```
Error: java.lang.Exception: typecheck2 does not exist.
```

```
val program90 = """
a = 1;
b = 2;
function sum(x,y){
    result = x + y;
    result;
}
print(sum(a,b,3));
"""
```

```
execute(program90)
```

```
Error: java.lang.Exception: sum expects 2 arguments, but 3 given.
```

Static Type Checker Examples Cont'd

```
val program91 = """
sum = true + 2;
print(sum);
"""
```

```
execute(program91)
```

```
Error: java.lang.RuntimeException: Non-integer operands for arithmetic operation
```

```
val program92 = """
function typecheck(a){
    x = "2";
    y = 1;
    result = y ++ x;
    result;
}
print(typecheck(1));
"""
```

```
execute(program92)
```

```
Error: java.lang.IllegalArgumentException: Type mismatch: at least one operand of a CONCAT operation must be a string
```

List Functionality

- Basic:
  - Initialization
  - Indexing
  - Appending
  - Printing

```
val program6 = """
List myList = [1,2,3];
print(Append myList [4,5,6]);
print(Index myList@2);
"""
execute(program6)
[1, 2, 3, 4, 5, 6]
3
```

- Advanced:
  - List Comprehension
  - Slicing

```
val program7 = """
function setOne(_val) {
  1;
}
List myList = [1,2,3];
[setOne(val) for val in myList];
"""
execute(program7)
[1, 1, 1]
```

```
val program8 = """
List myList = [1,2,3,4,5];
print(Slice myList[1:4]);
"""
execute(program8)
[2, 3, 4]
```

How can this be extended?

- Can extend this language to implement more dynamic type checking.

- Extend functions to have specific return types.

- Use explicit type definition for functions and variables (like C rather than python)

- Implement a simple class interface