EECS 293 Homework 11: “Type Synthesizer: Additional functionality on top of parsers”

**People**: Jessica Kwok, Phan Trinh Ha

**Overview**: We are adding a functionality that infers type of our parsed tree, given a variable declaration and type rules as our input. Our approach to this project is reading from two text files containing our given types, variables, and rules. We are building on top of the old parser code to implement the type synthesis portion of this project. The old code will be used to validate the input parse tree. Then by using depth first search, we will decipher the type of each subexpression to find the overall type of the tree.

**Problem**: Given our parser, we can now parse from an input to a complete (simplified) parse tree. However, the parser does not support the function such as type validation, type checking for the expression, especially for cases such as “String” + “int” should not be defined as something valid in Java, and that “int” + “double” should be defined as a double, or based on the given rule.

**Goals**: Adding a type synthesis/validation to the parser project

**Project**:

We will approach this problem with the given inputs: parse tree, type rules, and type assignment to each variable

Part 1: Input/Barricade

Part 2: Processes

Part 3: Error handling



**Part 1: Input/Barricade (Converting and filtering data)**

**Class:** Driver

This class is the top layer class that will take in the user’s input.

* Method(s):
  + parseInput(parseTree, varAssignments, convRules) - parses on the arguments into the hidden program components

**Class:** Barricade

This class serves as the error handling for all of the input parameters.

* Method(s):
  + validateInput(parseTree, varAssignments, convRules) - makes sure the parse tree is correctly simplified and the varAssignments and convRules text files are properly formatted

**Part 2: Processes (The inner processing of the type validation)**

**Class:** Synthesizer

This class is where all of the main components are happening such as taking in the list of rules and variable assignments and using them to find the type of the root node of the input parse tree.

* Attribute(s)
  + Rule rule
  + List of variables
* Method(s):
  + createRules(convRules) - takes in the text file and iterates line by line to create and set the private rule field
  + assignVars(varAssignments) - takes in the text file and iterates line by line to set the variables to their corresponding types
  + findTypeOf(parseTree) - will go through the parse tree and find the type of the root node.

1. Type resultType = null;
2. DFS through parseTree
3. Store visited nodes and compare to the rule set at each iteration
4. If a matching rule found
5. Delete stored nodes and insert rule type
6. Continue search
7. Return resultType

**Class:** Variable (given from the parser)

This class represents

* Additional Attribute(s)
  + Type type
* Addition Method(s)
  + setType() - gets the representation field value
  + getType() - gets the type field value

**Class:** Rule

This class represents the set of type conversion rules that the user inputs. Rules such as “Type A + Type B” will map to Type A

* Attribute(s)
  + Map<Equation, Type> map
* Method(s)
  + addRule(Equation eq, Type type) - maps the equation to its corresponding type
  + infer(Equation eq) - gets the value to the given key

**Class:** Type

This class is a data wrapper class to represent the type of a variable or to aid in giving the result type in a type conversion rule.

* Attribute(s)
  + String type
* Method(s)
  + Type(String type) - private constructor
  + build(String type) - implements the private constructor
  + getType() - gets the type field value

**Class:** Equation

This class represents the equation part of a conversion rule, we are enforcing the equation to be

* Attribute(s)
  + List<Type> types: Required to be immutable, and ordered as this will be used to determine the type of a resulting type check
  + Connector connector
* Method(s)
  + Equation(List<Type> types, Connector connector) - private constructor
  + build(List<Type> types, Connector connector) - implements the private constructor

**Class:** Connector

This class is the same implementation as it was in projects 2-5

**Part 3: Error Handling**

* Synthesizer - Inner Processing
  + Custom InvalidTypingException - handle cases where a type conversion rule is not defined for the given type
  + Custom UndefinedException - handle cases where types/variables are previously defined in input
* Barricade - Input Processing
  + Standard errors
    - InvalidArgument - handle rule/type definitions

**Approach to Testing**

Bottom up testing where we test starting at the inner classes and move to the outward classes of the system. At the end, we would do a whole system test, by creating inputs and test the outputs on the top level Driver class, and stress testing. When testing individual methods, we will use assertions to validate preconditions. We want to use a logger to keep track of newly generated data.

**Expected inputs**

If I give in a + b and a is type integer and b is type double, then the program must evaluate a + b to type double

f(a) is represented as f + a, given that if f is a function of generic type Double, and a is an Integer, our output would be type Double

a+b/c is represented in the parse tree as [a + [b / c]]. Assume that our inputs will be:

Parse tree representation: [a + [b / c]]

var.txt:

a : “A”

b : “B”

c : “C”

rules.txt:

A, B, C

A + B : C

B / C : B

Output will be come

Type : C

F \* g will be a result in type of the function<T> if both are T