Design Document Programming Assignment 11

(Revised for Programming Assignment 12)

Anna Sedlackova | Jason Shin

We have five classes: TypeSynthesis, ExpressionTypes, VariableTypes, Type, and Expression. The user would call TypeSynthesis.evaluateRootType with a set of predefined expression types, variable types, and the root of the parsed tree in order to evaluate the type of the root.

Pseudocode

public final class TypeSynthesis

This class contains the main algorithm for analyzing the inputted tree based on the rules of how the types are evaluated.

Fields:
private ~~final~~ VariableTypes variableTypes
private ~~final~~ ExpressionTypes typeConversionRules

Constructor:
TypeSynthesis(Optional<Node> parsedTree, VariableTypes variableTypes, ExpressionTypes typeConversionRules)

Methods:
```
//only public method that evaluates the result of the expression tree
public Type evaluateRootType(Optional<Node> parsedTree, VariableTypes variableTypes,
ExpressionTypes typeConversionRules):
        variableTypes ← variableTypes
        typeConversionRules ← typeConversionRules

        if validateTree(parsedTree):
                return traverseTree (parsedTree)
        else:
                return Type.undefined

private boolean validateTree (Node parsedTree):
        Input: A parse tree given by the user
        Output: True or False on whether the tree is valid
// checks that all nodes contain at least an operator and a internalnode (in that order)
// minus, InternalNode
// minus, Variable
// Variable or InternalNode, operator, Variable or InternalNode, with last two repeated an arbitrary
amount of times
// Also checks that each open parenthesis has a closed parenthesis in an InternalNode
```

```
private Type traverseTree (Node parsedTree):
        Input: A parse tree given by the user
        Output: Type of the root node of the parse tree

        Expression exp ← new Expression with Type.empty, Type.empty, and Connector.empty

        for each child in parsedTree's children do
                exp ← addChildToExpression(exp, child)
                exp ← evalExpressionAndSetLeft(exp)
                If (exp's leftExpressionType is Type.undefined) then
                        return Type.undefined
        return exp's leftExpressionType

//checks if node contains a parenthesis
private boolean isParenthesis (Node n)

//checks that all inputs are not null
This method serves as a mini barricade that ensures along with validateTree, that the input is not null
and correct.
private boolean isInputNotNull

private Expression addChildToExpression(Node child, Expression exp)
        Type childType ← null
        If (child is an InternalNode) then
                childType ← traverseTree(child)
        else If (child is an operator) then
                set exp's expressionSymbol to child's token (Connector)
                return exp
        else if (child is not a parenthesis) then
                childType ← look up child's Variable in variableTypes
        else
                return exp
        return addTypeToExpression(exp, childType)

private Expression addTypeToExpression(Expression exp, Type t)
        If (exp's expressionSymbol is not Connector.empty) then
                set exp's rightExpressionType to t
        else
                set exp's leftExpressionType to t
        return exp

private Expression evalExpressionAndSetRight(Expression exp)
        If exp's rightExpressionType is Type.empty then
                return exp
        typeOfExpression ← look up exp in typeConversionRules to get type of expression, if it does not
                           exist in typeConversionRules, Type.undefined
```

set exp’s leftExpressionType **to** typeOfExpression
set exp’s expressionSymbol **to** Type.empty
set exp’s rightExpressionType **to** Type.empty
return exp

final class Expression

This class stores left and right expression and also a connector. The expressions can be Type.empty, and the issue with unaries is addressed in the main algorithm.

Fields:
private Type leftExpressionType
private Type rightExpressionType
private Connector expressionSymbol

Constructor:
Expression(Type leftExpressionType, Type rightExpressionType, Connector expressionSymbol)
leftExpressionType ⟵ leftExpressionType
rightExpressionType ⟵ rightExpressionType
expressionSymbol ⟵ expressionSymbol

Methods:
@Override
boolean equals(Object obj)
Type getLeftExpression()
Type getRightExpression()
Connecter getExpressionSymbol()
void setLeftExpression(Type leftExpressionType)
void setRightExpression(Type rightExpressionType)
void setExpressionSymbol(Connector expressionSymbol)

final class Type (immutable)

This class stores the types that user inputs as strings. It has two special types: empty and undefined. Undefined is returned if a rule or type variable mapping is missing.

Field:
private final String variableType
static final Type empty = new Type(“”)
static final Type undefined = new Type(“NA”)
Constructor:
Type (String type)
Methods:
String getType()
@Override
public String toString()

final class ExpressionTypes (cannot be extended)

This class holds a set of conversion rules that are inputted by the user (in the format Expression → Type).

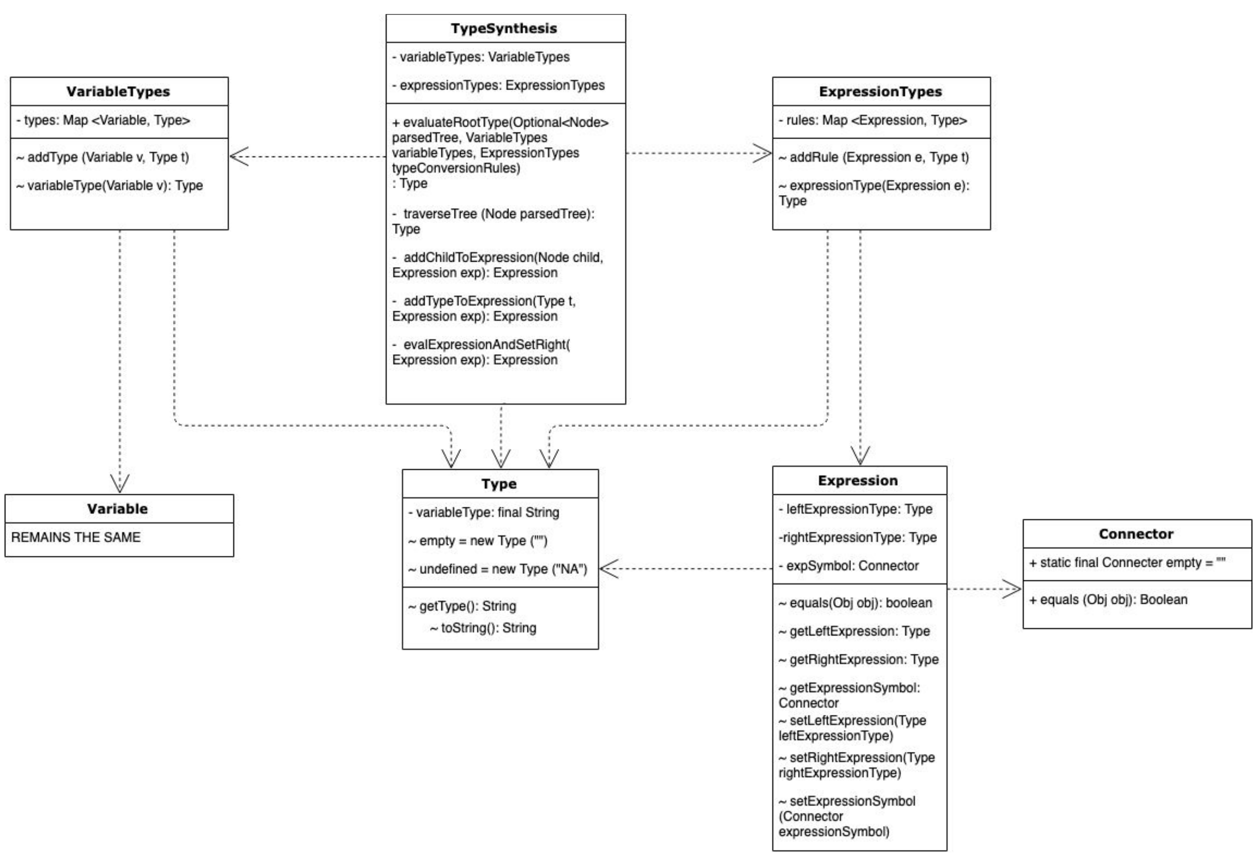
Field:
private Map<Expression, Type> rules
Methods:
void addRule(Expression e, Type t)
Type expressionType(Expression e)

final class VariableTypes (cannot be extended)

This class holds a set of variable types that are inputted by the user (in the format Variable → Type).

Field:
private Map<Variable, Type> types
Methods:
void addVariableType(Variable v, Type t)
Type variableType(Variable v)

Class Diagram



Error Handling Approach

Because our inputs are objects and not read from a file, we do not need to implement a barricade or complex error checking. However, we plan to utilize the following:

- **null checks** for each of the inputs of evaluateRootType (parsed tree, a map for variable types, a map for expression types) at the beginning of the method
- if we detect a **missing variable type** while going up the tree, we immediately return that the type is *undefined* (String NA)
- if a **missing rule** is detected, the result of the operation will be *undefined* (String NA) and immediately returned as such
- if the root node is empty, we will also return *undefined* (String NA)

Outlined Testing Approach

Unit Testing

- For private methods, we will create a TestHook for each of the classes in order to test them

Testing evaluateRootType method of TypeSynthesis

- To create the parse tree, we will use NonTerminalSymbol's parseInput method on a Token List that we will generate from an input String.
- To create VariableTypes, we will instantiate it and populate the Map it holds it using addVariableType with each Variable-Type pair.
- To create ExpressionTypes, we will instantiate it and populate the Map it holds using addRule with each corresponding Expression-Type pair.
- We will then input these to the TypeSynthesis method to see if the result we get is what we expect.
- We will try to achieve 100% branch and code coverage.

Example

Input: [a, +, [(, -, b,)]]

The tree will be built (and expected by our validating method) as follows:

```
Enter an expression:
 [a, +, (, -, b, )]
[        ]
 a+[    ]
   (-b)
```

Example VariableType Input:

a → Cat

b → Dog

Example ExpressionType Input

Expression (Type.empty, -, Dog) → Dog

Expression (Cat, +, Dog) ⟶ Cat

- variableTypes field set to passed in VariableType
- typeConversionRules set to passed in ExpressionType
- Tree is valid, so calls traverseTree on parsedTree
- exp is initialized with Type.empty, Type.empty, and Connector.empty
- a is looked up and is type Cat
- Since exp's expressionSymbol is Connector.empty, Cat is set to exp's leftExpressionType
- **exp is now [Cat, Connector.empty, Type.empty]**
- Since + is not a parenthesis and is an operator, + is set to exp's expressionSymbol
- **exp is now [Cat, +, Type.empty]**
- Since [(,-,b,)] is an internalNode, traverseTree is recursively called on it
    - Parenthesis are ignored
    - Since - is not a parenthesis and and is an operator, - is set to exp's expressionSymbol
    - **exp is now [Type.empty, -, Type.empty]**
    - b is looked up and is type Dog
    - Since exp's expressionSymbol is not Connector.empty, Dog is set to exp's rightExpressionType
    - **Exp is now [Type.empty, -, Dog]**
    - Since exp's rightExpressionType is not Type.empty, we look up exp in TypeConversionRules to get Type Dog
    - Dog is set to exp's leftExpressionType, exp's rightExpressionType set to Type.empty, exp's expressionSymbol is set to Connector.empty.
    - **Exp is now [Dog, Connector.empty, Type.empty]**
    - Dog is returned
- [(,-,b,)] is determined to be a Dog
- Since exp's expressionSymbol is not Connector.empty, Dog is set to exp's rightExpressionType
- **exp is now [Cat, +, Dog]**
- Since exp's rightExpressionType is not Type.empty, we look up exp in TypeConversionRules to get Type Cat
- Cat is set to exp's leftExpressionType, exp's rightExpressionType set to Type.empty, exp's expressionSymbol is set to Connector.empty.
- **exp is now [Cat, Connector.empty, Type.empty]**
- Cat is returned

----

public abstract class AbstractToken implements Token

//the abstract token representation

FIelds:

protected TerminalSymbol type

Methods:

public final boolean matches(TerminalSymbol type) public TerminalSymbol getType()

final class Cache<T,V>

//Get an item from the cache if one exists, and if not, call the provided constructor
FIelds:
private Map<T, V> cache = new HashMap<T, V>()
Methods:
V get(T key, Function<? super T, ? extends V> constructor)

public final class Connector extends AbstractToken

FIelds:
//this class is for the connectors in a numerical expression
private static final List<TerminalSymbol> allowedSymbols ⟵ added empty
private static final List<TerminalSymbol> operatorSymbols
private static Cache<TerminalSymbol, Connector> cache
public static final Connector empty = TerminaSymbol.empty ⟵ terminal symbol will be edited such that we can have an empty connector
Constructor:
private Connector(TerminalSymbol type)
Methods:
static Function<TerminalSymbol, Connector> connectorConstructor
public static final Connector build (TerminalSymbol type)
public String toString()
public boolean isOperator()
@Override
public boolean equals(Object obj)

@Override
public boolean hashcode()

public class InternalNode implements Node

Field:
private final List<Node> children
private List<Token> cachedTokenList
private String cachedStringRepresentation
Constructor:
private InternalNode(List<Node> children)
Methods:
public List<Token> toList()
public List<Node> getChildren()
public static InternalNode build(List<Node> children)
public String toString()
public boolean isFruitful()
public boolean isOperator()
public boolean isStartedByOperator()
public Optional<Node> firstChild()

public boolean isSingleLeafParent()

public static class Builder (contained within InternalNode)

Field:
private List<Node> children
Methods:
public boolean addChild(Node node)
private List<Node> getChildren()
private static List<Node> unnestSingleChildren(List<Node> list)
private static <T> void addAllToListIterator(List<T> list, ListIterator<T> iterator)
private static boolean startsWithBinaryOperator(Node curChild, Node lastChild)
private static List<Node> unnestOperators(List<Node> list)
private static List<Node> unnestSingleLeafParent(List<Node> list)
public Builder simplify()
public InternalNode build()

public class LeafNode implements Node

//class for leaf nodes in the tree representation
Field:
private final Token token
Constructor:
private LeafNode(Token token)
Methods:
public List<Token> toList()
public Token getToken()
public static LeafNode build(Token token)
public String toString()
public List<Node> getChildren()
public boolean isFruitful()
public boolean isOperator()
public boolean isStartedByOperator()
public Optional<Node> firstChild()
public boolean isSingleLeafParent()

public abstract class ListHandler

Constructor:
private ListHandler()
Methods:
public static <T> T listHead(List<T> list)
public static <T> T listSecond(List<T> list)
Public static <T> T listLast(List<T> list)
public static <T> boolean nonEmptyList(List<T> list)
public static <T> List<T> replaceItemsMatching(List<T> list, Function<T, Boolean> predicate, Function<T, T> replacement)
public static <T> boolean containsSingleItem(List<T> list)

public interface Node

//This class is for the nodes in the tree representation of a numerical expression

public enum NonTerminalSymbol implements Symbol

Field:
private static final Map<NonTerminalSymbol, Map<TerminalSymbol, SymbolSequence>> production
Methods:
private static Map<TerminalSymbol, SymbolSequence> getProductionTable(NonTerminalSymbol lookupSymbol)
private static void addProduction(NonTerminalSymbol lookupSymbol, TerminalSymbol lookAhead, SymbolSequence symbols)
private static void addProduction(NonTerminalSymbol lookupSymbol, TerminalSymbol lookAhead, List<Symbol> symbols)
private static SymbolSequence makeSymbolSequence(TerminalSymbol firstSymbol, List<Symbol> remainder)
public static final Optional<Node> parseInput(List<Token> input)
public ParseState parse(List<Token> input)
//includes a static block that places each production into the expression sequence

abstract class ObjectHandler

Constructor: private

Methods:

public static <T> T returnIfNotNull(Object valueToCheck, T valueToReturn)

public static <T> T createObjectIfNull(T objectToCheck, Function<Void, T> constructor)

final class ParseState

Fields:

private final boolean success

private final Node node

private final List<Token> remainder

Constructor:

private ParseState(boolean success, Node node, List<Token> remainder)

Methods:

public static ParseState build(Node node, List<Token> remainder)

public boolean getSuccess()

public Node getNode()

public List<Token> getRemainder()

public final boolean hasNoRemainder()

interface Symbol

public ParseState parse(List<Token> input)

final class SymbolSequence

Fields:

private final List<Symbol> production

final static SymbolSequence EPSILON

Constructor:

private SymbolSequence(List<Symbol> production)

Methods:

public static SymbolSequence build(List<Symbol> production)

static final SymbolSequence build(Symbol... symbols)

public String toString()

public ParseState match(List<Token> input)

public enum TerminalSymbol implements Symbol

Fields:

private final String stringRepresentation

private final static Map<String, Token> connectorMappings

Methods:

private TerminalSymbol(String stringRepresentation)

public String toString()

public ParseState parse(List<Token> input)

public static Token stringToToken(String string)

public interface Token

Methods:

TerminalSymbol getType()
boolean matches(TerminalSymbol type)
boolean isOperator()

public final class Variable extends AbstractToken

Fields:

private final String representation

private static Cache<String, Variable> cache

Constructor:

private Variable(String representation)

Methods:

public final String getRepresentation()

public static final Variable build(String representation)

public String toString()

public boolean isOperator()