# Project Description

**Notes from Discussion Section**

- Don't need to add tests for code from projects 2-5
- New error handling: What if two rules conflict?

**Task**

Design the architecture of a system that takes a (simplified) parse tree, variable types, and type conversion rules as inputs, and outputs the type of the parse tree root

**For this assignment**

- Create a design document that describes your architectural decisions.
    - Should be used as a blueprint for the implementation
    - Can contain sketches and diagrams of your architecture
- Define classes or interfaces as we need. For each class we need to describe:
    - The abstraction it captures
    - Position in the inheritance hierarchy
    - Constructors and public methods
    - Private data structures
    - Pseudo-code for any complicated methods
- Approach to error handling
- Approach to testing
    - Examples of the operations of your concept
        - Should cover both nominal and border cases
        - For each example, provide the input and expected output
        - Eventually, these examples should be included in our JUnit test suite

**Process**

1. Check if the tree is valid
2. Split into subtrees by order of operations
3. Recursively apply type rules to subtrees (PEMDAS)
4. Apply type rules to children
5. Return root type

# Programming Assignment 11 Design Document

Lucas Popp and Arik Stewart

**Task**
Design the architecture of a system that takes a (simplified) parse tree, variable types, and type conversion rules as inputs, and outputs the type of the parse tree root

**Conceptual Flow**

- ⬇ Inputs
1. Check if the tree is valid
2. Split into subtrees by order of operations
3. Recursively apply type rules to subtrees (PEMDAS)
4. Apply type rules to children
- ⬇ Return root type

# Class Architecture

Utilizing Previous Parser Package, HW5
All classes/variables/methods with no declared access level are package-private

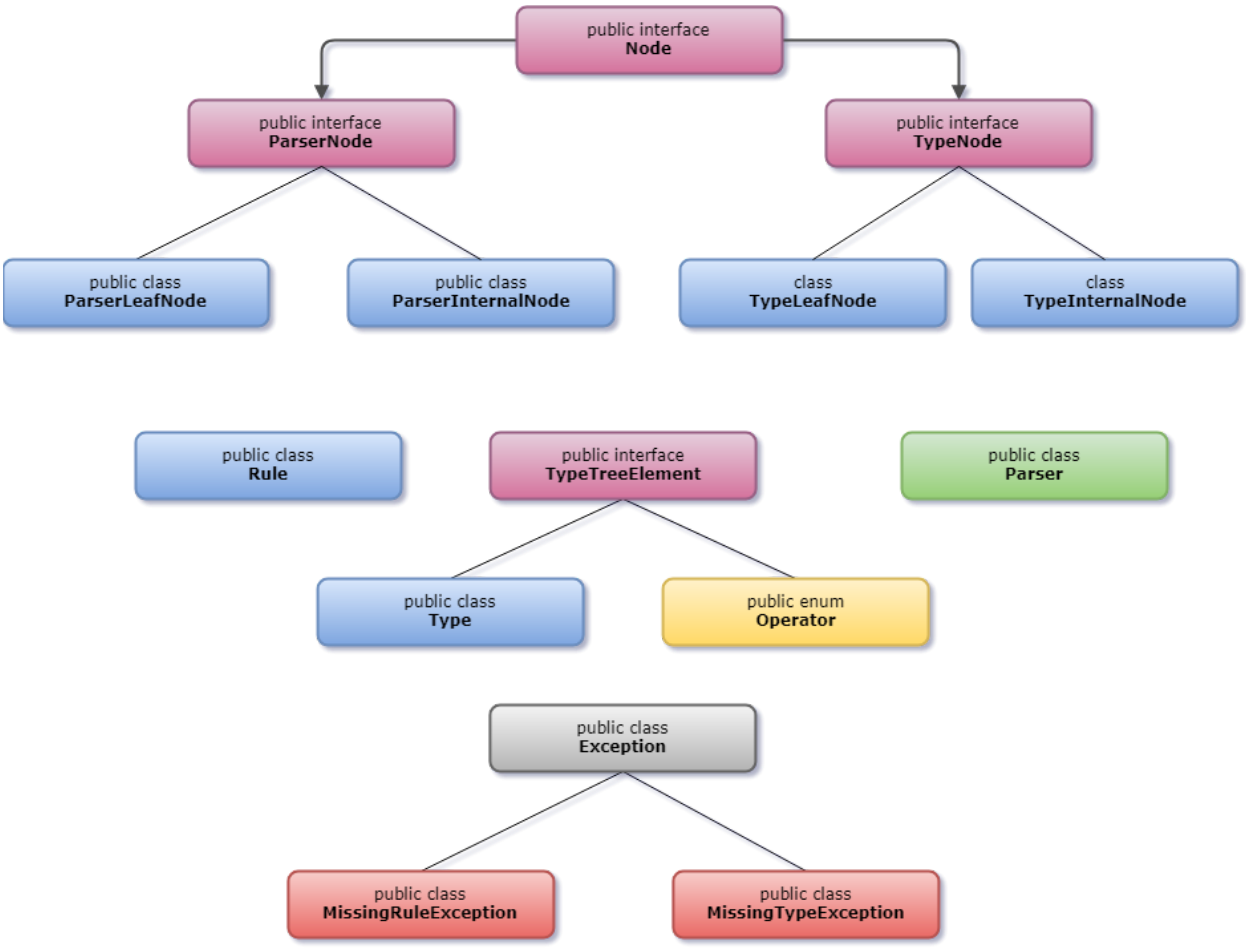
Rename Node, LeafNode, InternalNode → ParserNode, ParserLeafNode, ParserInternalNode
Make a new marker interface Node. ParseNode implements Node

```
public interface TypeTreeElement
	Marker interface

public class Type implements TypeTreeElement
	private static Cache<String, Type> cache
	private String stringValue

	private Type(String stringValue)
	public static Type build(String stringValue)
	public String getStringValue()

public enum Operator implements TypeTreeElement
	PLUS, MINUS, TIMES, DIVIDE

public interface TypeNode implements Node
	public List<TypeNode> getChildren()
	public boolean hasBeenEvaluated()
	public Optional<Type> getEvaluatedType()
```

```
class TypeLeafNode implements TypeTreeNode
	private TypeTreeElement element

	private TypeLeafNode(element)
	public static TypeLeafNode build(element)
	public boolean isOperator()
	public TypeTreeElement getElement()

	Inherited
	public List<TypeNode> getChildren() → empty
	public bool hasBeenEvaluated() → true
	public Optional<Type> getEvaluatedType()

class TypeInternalNode implements TypeTreeNode
	private boolean evaluated
	private List<TypeNode> children
	private Optional<Type> evaluatedType

	private TypeInternalNode(children)
	public static TypeInternalNode build(children)
	public arrangeChildren()
	public evaluateTypeUsingRules(Set<Rule> rules) throws MissingRuleException
	private boolean matchesRule(Rule rule)

	Inherited
	public List<TypeNode> getChildren()
	public boolean hasBeenEvaluated()
	public Optional<Type> getEvaluatedType()

public class Parser
	public String evaluateExpressionType(ParseInternalNode root,
		Map<Variable, String> types, Set<Rule> rules)
	private boolean treeIsValid()
	private TypeInternalNode convertToTypeTree(ParseInternalNode root)
		throws MissingTypeException

public class Rule
	public Rule(Type leftType, operator, Type rightType, Type resultingType)
	private Type leftType, rightType, resultingType
	private Operator operator
	boolean matches(Type leftType, Operator operator, Type rightType)
	boolean matchesRule(List<> children)
	Type getResultingType()
```

```
public class MissingRuleException extends java.lang.Exception
	private Type leftType, rightType
	private Operator operator
	public MissingRuleException(Type leftType, Operator operator, Type rightType)

public class MissingTypeException extends java.lang.Exception
	private Variable variable
	public MissingTypeException(Variable variable)
```

# Type Hierarchy



# Error Handling

Error handling here does not leave much room for error, as the program requires a certain level of precision in order to perform as expected in all cases.

| **Error** | **Handling** |
|---|---|
| Inputted parse tree, map of variable types, or set of rules is null | Throw NullPointerException |
| Inputted parse tree is invalid. If it contains…<br>● Two consecutive variables/terms with no operator in the middle<br>● Two consecutive operators, unless…<br>○ The second is a minus and the first isn't<br>○ The first is a closing parenthesis, and the second isn't an opening parenthesis, or visa versa<br>● Unbalanced parentheses | Throw IllegalArgumentException with message "Invalid parse tree" |
| Map of variable types contains a null key/value | Throw IllegalArgumentException |
| A type/operator/type combination is encountered during evaluation that isn't covered by the given rules | Throw MissingRuleException |
| A variable is encountered during parsing that doesn't have a type specified by the given variable types | Throw MissingTypeException |

# Test Cases

**Nominal Case**

**Parse Tree**
[a, +, b]

**Output**
Double

**Variable Types**

| | |
|---|---|
| a | Int |
| b | Double |

**Rules**

| Type 1 | Operator | Type 2 | Output |
|---|---|---|---|
| Int | + | Double | Double |

**Unconventional Types**

**Parse Tree**
[f, +, a]

**Output**
T

**Variable Types**

| | |
|---|---|
| f | Function<T> |
| a | T |

**Rules**

| Type 1 | Operator | Type 2 | Output |
|---|---|---|---|
| Function<T> | + | T | T |

**Order of Operations**

**Parse Tree**
[a, +, b, +, b, +, b, +, b, +, b]

**Variable Types**

| | |
|---|---|
| a | Double |
| b | Int |

**Rules**

| Type 1 | Operator | Type 2 | Output |
|---|---|---|---|
| double | + | int | double |

**Output**
Double

**Nested Conversions**

**Parse Tree**
[a, +, [b, *, c]]

**Variable Types**

| | |
|---|---|
| a | int |
| b | int |
| c | int |

**Rules**

| Type 1 | Operator | Type 2 | Output |
|---|---|---|---|
| int | * | int | double |
| int | + | double | double |

**Output**
double

**Invalid Parse Tree**

**Parse Tree**
[a, *, /, b]

**Variable Types**

| | |
|---|---|
| a | int |
| b | int |

**Rules**

| Type 1 | Operator | Type 2 | Output |
|---|---|---|---|
| int | * | int | double |
| int | / | int | double |

**Output**
IllegalArgumentException: “Invalid tree”

**Missing Variable Type**

**Parse Tree**
[a, +, b]

**Variable Types**

| | |
|---|---|
| a | int |

**Rules**

| Type 1 | Operator | Type 2 | Output |
|---|---|---|---|
| int | + | int | double |
| int | / | int | double |

**Output**
MissingTypeException variable b

**Missing Rule Coverage**

**Parse Tree**
[a, +, b, -, c]

**Variable Types**

| | |
|---|---|
| a | Int |
| b | Double |
| C | Float |

**Rules**

| Type 1 | Operator | Type 2 | Output |
|---|---|---|---|
| Int | + | Double | Double |

**Output**
**Output**
MissingRuleException: “Missing the conversion rule for Double - Float”

# Some Helpful Notes

Valid reasons to create a class:

- Model real-world objects
- Model abstract objects
- Reduce complexity
- Isolate complexity
- Hide implementation details
- Limit effects of changes
- Hide global data
- Streamline parameter passing
- Make central points of control
- Facilitate reusable code
- Plan for a family of programs
- Package related operations
- Accomplish a specific refactoring

Defensive programming checklist

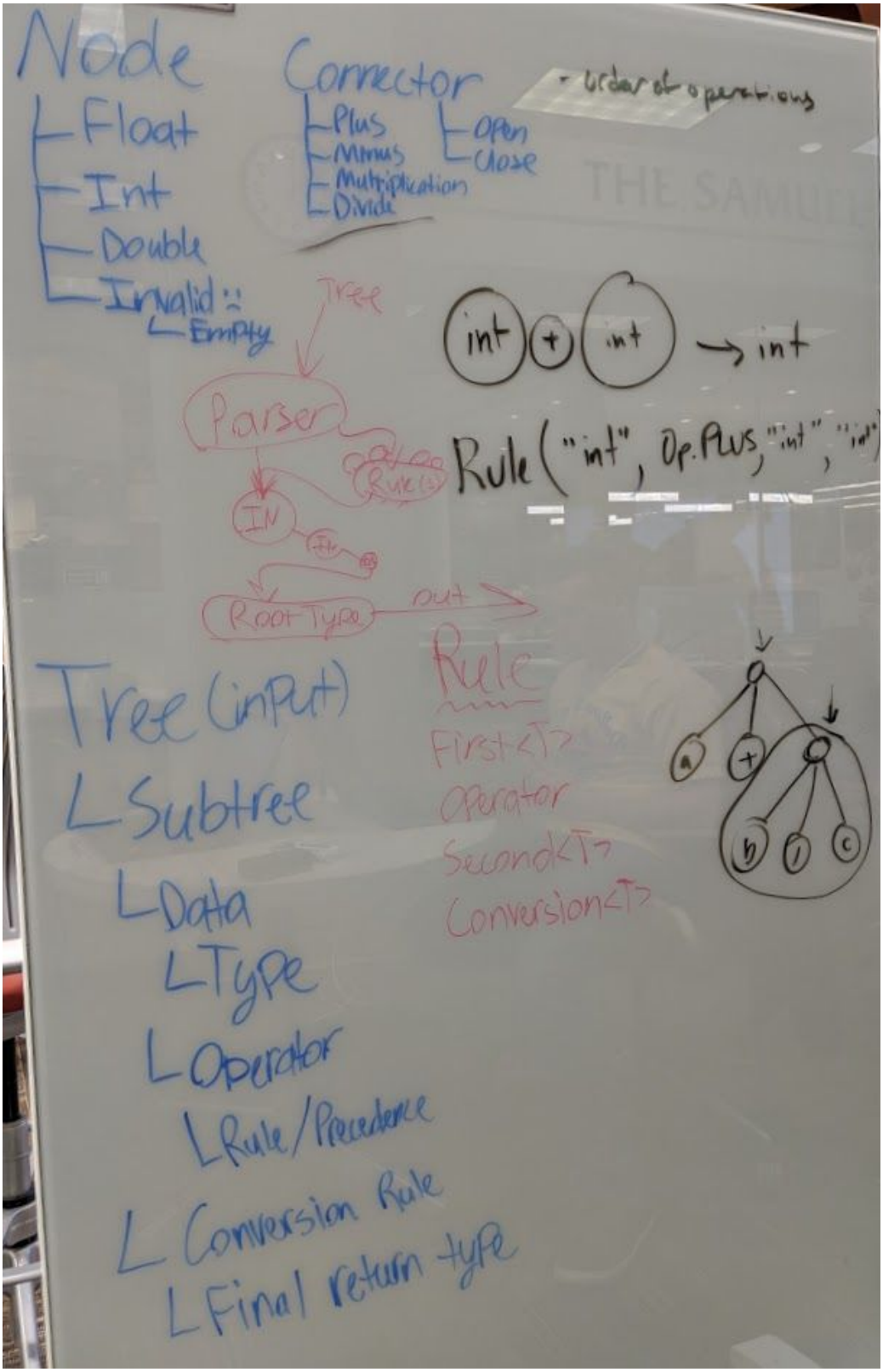
- ❏ Do expressions use true and false rather than 1 and 0?
- ❏ Are boolean values compared to true and false implicitly?
- ❏ Are numeric values compared to their test values explicitly?
- ❏ Have expressions been simplified by the addition of new boolean variables and the use of boolean functions and decision tables?
- ❏ Are boolean expressions stated positively?
- ❏ Are braces used everywhere they're needed for clarity?
- ❏ Are logical expressions fully parenthesized?
- ❏ Have tests been written in number-line order?
- ❏ Do Java tests uses a.equals(b) style instead of a == b when appropriate?
- ❏ Are null statements obvious?
- ❏ Have nested statements been simplified by retesting part of the conditional, converting to if-then-else or case statements, moving nested code into its own routine, converting to a more object-oriented design, or have they been improved in some other way?

Exceptions checklist

- ❏ Has your project defined a standardized approach to exception handling?
- ❏ Have you considered alternatives to using an exception?
- ❏ Is the error handled locally rather than throwing a nonlocal exception, if possible?
- ❏ Does the code avoid throwing exceptions in constructors and destructors?
- ❏ Are all exceptions at the appropriate levels of abstraction for the routines that throw them?
- ❏ Does each exception include all relevant exception background informa- tion?

- ❏ Is the code free of empty catch blocks? (Or if an empty catch block truly is appropriate, is it documented?)

Board Stuff


Node
Float
Int
Double
Invalid
Empty
Connector
Plus
Minus
Multiplication
Divide
open
close
order of operations
Tree
Parser
Rule(s)
IN
Root Type
out
int + int → int
Rule("int", Op.Plus, "int", "int")
Tree (input)
Subtree
Data
Type
Operator
Rule/Precedence
Conversion Rule
Final return type
Rule
First<T>
operator
Second<T>
Conversion<T>
a + b / c