EECS3311-F19 — Project Report

Submitted electronically by: Juan Leaniz Pittamiglio

|  |  |  |  |
| --- | --- | --- | --- |
| **Team members** | **Name** | **Prism Login** | **Signature** |
| Member 1: | Juan Leaniz Pittamiglio | Jleaniz |  |
| Member 2: | Majd Arkilo | Xxxxx |  |
| \*Submitted under Prism account: | | jleaniz |  |

**Contents**

[1. Requirements for Project “Analyzer” 2](#_Toc26389353)

[2. BON class diagram overview (architecture of the design) 3](#_Toc26389354)

[3. Table of modules — responsibilities and information hiding 4](#_Toc26389355)

[4. Expanded description of design decisions 6](#_Toc26389356)

[5. Significant Contracts (Correctness) 7](#_Toc26389357)

[6. Summary of Testing Procedures 8](#_Toc26389358)

[7. Appendix A (Context-free Grammars) 8](#_Toc26389359)

[8. Appendix B (Contract view of all classes) 9](#_Toc26389360)

# Requirements for Project “Analyzer”

Our team was engaged by “ACME Inc.” to develop a proof of concept programming language and some associated functionality. The customer requested three main features for this programming language:

1. Pretty Printing: The user interface for the language should pretty print variable assignments in a pre-determined format.
2. Type checking: The user interface for the language should type check each program and inform the user if there are any type errors.
3. Generate Java Code: The user interface for the language should allow the user to generate Java-like code for their programs.

The programming language follows the syntax specified by the Context-free Grammars detailed in Appendix A.

# BON class diagram overview (architecture of the design)

# Table of modules — responsibilities and information hiding

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | EXPRESSION | **Responsibility**: Represents an expression in the programming language. | **Alternative**: None |
| Abstract | **Secret**: none |
| 1.1 | INTEGER\_CONSTANT | **Responsibility**: Represents an integer constant. | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.2 | BOOLEAN\_CONSTANT | **Responsibility**: Represents a Boolean constant | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3 | COMPOSITE\_EXPRESSION[G] | **Responsibility**: Represents an expression comprised of exactly two expressions (left and right). This is used to represent the Composite design pattern. | **Alternative**: None |
| Abstract | **Secret**: “children” are represented as EXPRESSION objects inside a LINKED\_LIST |
| 1.3.1 | ADDITION | **Responsibility**: Represents an addition expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.2 | EQUALS | **Responsibility**: Represents the equality expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.3 | GREATER\_THAN | **Responsibility**: Represents a “greater than” expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.4 | LESS\_THAN | **Responsibility**: Represents a “less than” expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.5 | LOGICAL\_AND | **Responsibility**: Represents the logical AND expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.6 | LOGICAL\_OR | **Responsibility**: Represents the logical OR expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.7 | MODULO | **Responsibility**: Represents a modulo expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.8 | MULTIPLICATION | **Responsibility**: Represents a multiplication expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.9 | QUOTIENT | **Responsibility**: Represents a quotient expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 1.3.10 | SUBTRACTION | **Responsibility**: Represents a subtraction expression | **Alternative**: None |
| Concrete | **Secret**: None |
| 2 | LANG\_CLASS | **Responsibility**: This class represents a “class” object in the programming language. | **Alternative**: None |
| Concrete | **Secret**: attributes, commands, and queries are stored in three LINKED\_LIST data structures. The order the features were created is stored in an ARRAY called features. |
| 2.1 | LANG\_FEATURE | **Responsibility**: This class is an abstract class to represent different features (attributes, commands, queries) | **Alternative**: None |
|  | Abstract | **Secret**: None |  |
| 2.1.1 | LANG\_ASSIGNMENT | **Responsibility**: This class represents a “assignment” object in the programming language. | **Alternative**: None |
| Concrete | **Secret**: None |
| 2.1.2 | LANG\_ATTRIBUTE | **Responsibility**: This class represents a “attribute” object in the programming language. | **Alternative**: None |
| Concrete | **Secret**: None |
| 2.1.3 | LANG\_COMMAND | **Responsibility**: This class represents a “command” object in the programming language. | **Alternative**: None |
| Concrete | **Secret**: None |
| 2.1.4 | LANG\_QUERY | **Responsibility**: This class represents a “query” object in the programming language. | **Alternative**: None |
| Concrete | **Secret**: None |
| 3 | VISITOR | **Responsibility**: This class is used to implement the Visitor design pattern. | **Alternative**: None |
| Abstract | **Secret**: None |

# Expanded description of design decisions

We decided to use the Visitor and Composite design patterns to implement the language functionalities for pretty printing, type checking and java code generation. The expressions in the programming language can be represented in a tree-like structure, so the Composite design pattern fits well. The Visitor design pattern is then used to visit each ‘node’ in the tree, which represents each expression, and perform some operation with it. Depending on the user input, this could be type checking, generating java code, or pretty printing variable assignments for a program in the language.

Different components of the language are represented using various classes. For example, an abstract class LANG\_FEATURE is used as the root of the class hierarchy for language classes. LANG\_CLASS (represents classes), LANG\_ATTRIBUTE (represents attributes), LANG\_ASSIGNMENT (represents variable assignments), LANG\_COMMAND (represents commands), and LANG\_QUERY (represents queries), are all descendants of LANG\_FEATURE. While LANG\_FEATURE itself has no specific functionality, it is useful to store different type of LANG\_\* objects within a generic, polymorphic array, in order to store all the features in a program (i.e. classes, commands, attributes, assignments, and queries).

The VISITOR class is an abstract class that includes all the deferred methods required to visit each type of expression in the language. Therefore, each of: CODE\_GENERATOR, PRETTY\_PRINTER, TYPE\_CHECKER must implement all the deferred visit declared in VISITOR.

Each of the programming language’s expressions are represented as their own classes: ADDITOIN, EQUALS, GREATER\_THAN, LESS\_THAN, LOGICAL\_AND, LOGICAL\_OR, MODULO, MULTIPLICATION, QUOTIENT, SUBTRACTION are all descendants of COMPOSITE\_EXPRESSION which is an abstract class used to represent a composite expression with a LHS and RHS as children or nodes in a tree.

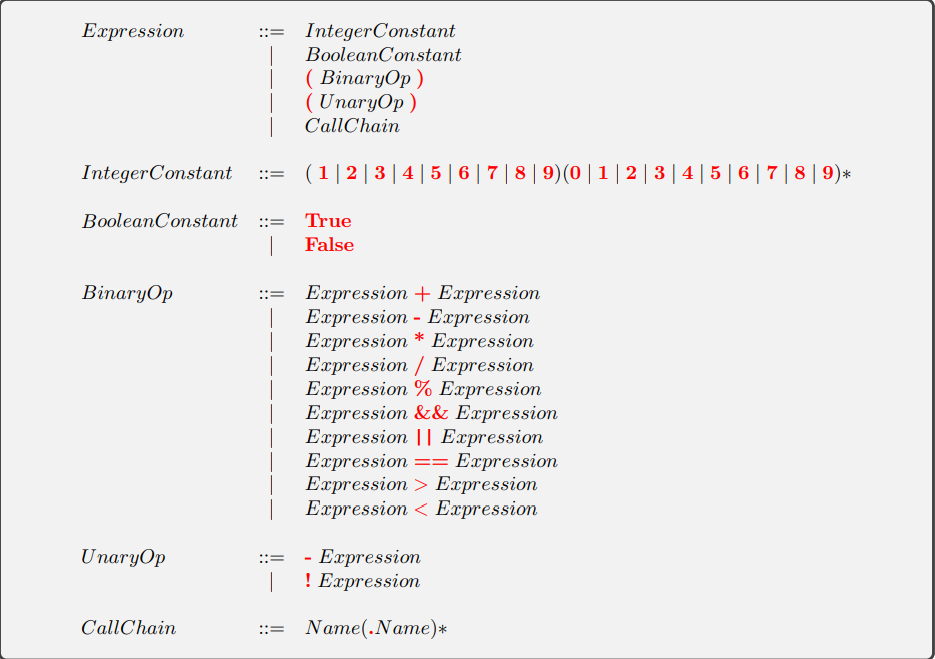
# Significant Contracts (Correctness)

(only for the module with the most significant contracts)

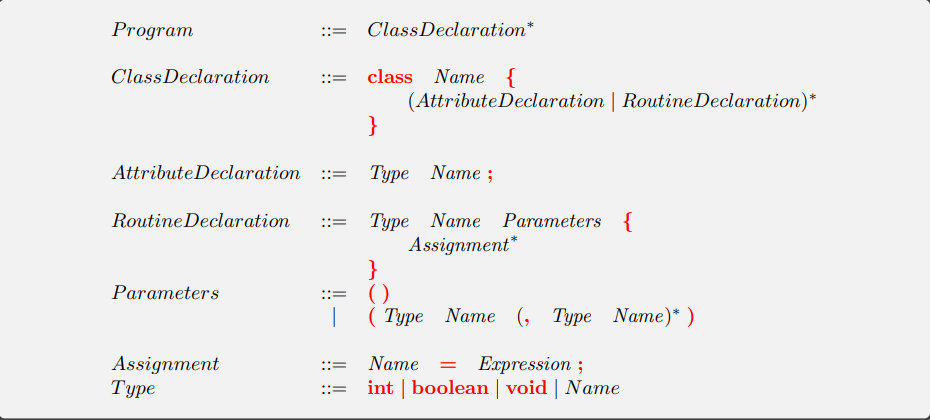
# Summary of Testing Procedures

# Appendix A (Context-free Grammars)

The follow figure illustrates the Context-free grammar for the expressions in the programming language:



The following figure illustrates the Context-free grammar for the program’s classes and features in the programming language:



# Appendix B (Contract view of all classes)

(Only classes that you created; do not include user input command classes, only model classes)