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 | majdar |  |
| \*Submitted under Prism account: | | jleaniz |  |

**Contents**

[1. Requirements for Project “Analyzer” 2](#_Toc26445267)

[2. BON class diagram overview (architecture of the design) 3](#_Toc26445268)

[3. Table of modules — responsibilities and information hiding 5](#_Toc26445269)

[4. Expanded description of design decisions 7](#_Toc26445270)

[5. Summary of Testing Procedures 8](#_Toc26445271)

[6. Appendix A (Context-free Grammars) 9](#_Toc26445272)

[7. Appendix B (Contract view of all classes) 10](#_Toc26445273)

# Requirements for Project “Analyzer”

Our team was engaged by “ACME Inc.” to develop a proof of concept of a small Java-like 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 print variable assignments in a pre-determined format called “pretty printing”
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 consist of a set of terminal symbols which represent the language’s keywords and other characters used for syntax purposes such as { } or ( ). The language supports all the typical operations you would expect to see in a modern language, for example, binary operations such as addition, multiplication, subtraction, logical operators (AND, OR), comparison operations like less than, greater than, equal, and two unary operations (logical negation and numerical negation).

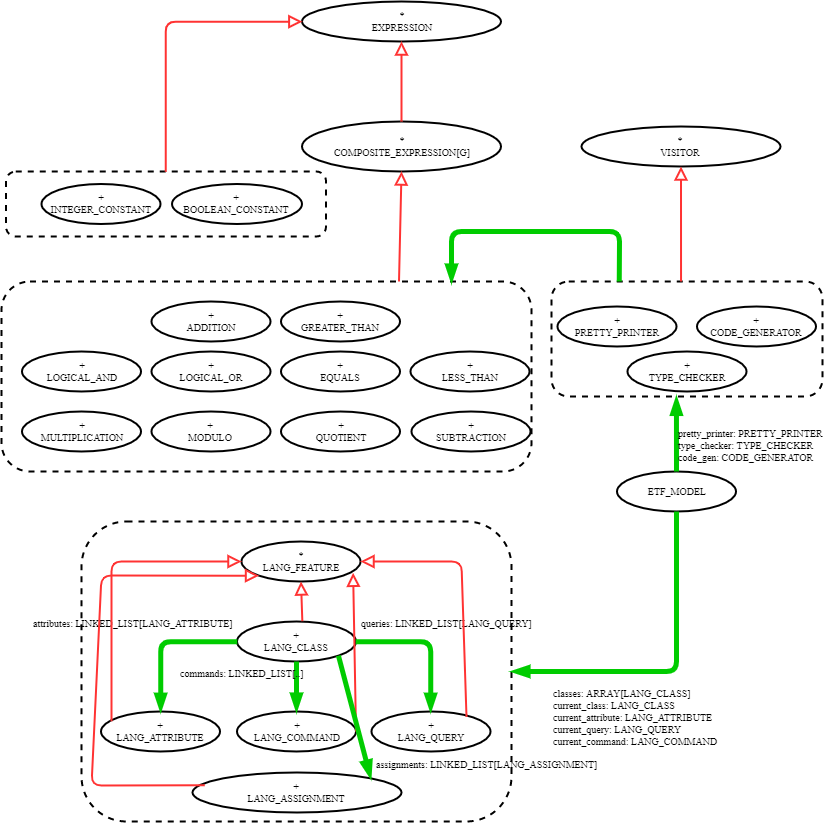
The language supports these operations via the use of Expressions, which are implemented using a context-free grammar detailed in Appendix A. The grammar may produce expressions which are not type-correct. A type checker was developed for the purpose of identifying expressions which are not type-correct and inform the user of such occurrences.

In the context of this simple programming language, the client requested that it only supports two primitive types: INTEGER and BOOLEAN. However, the language could be easily extended to support various other types.

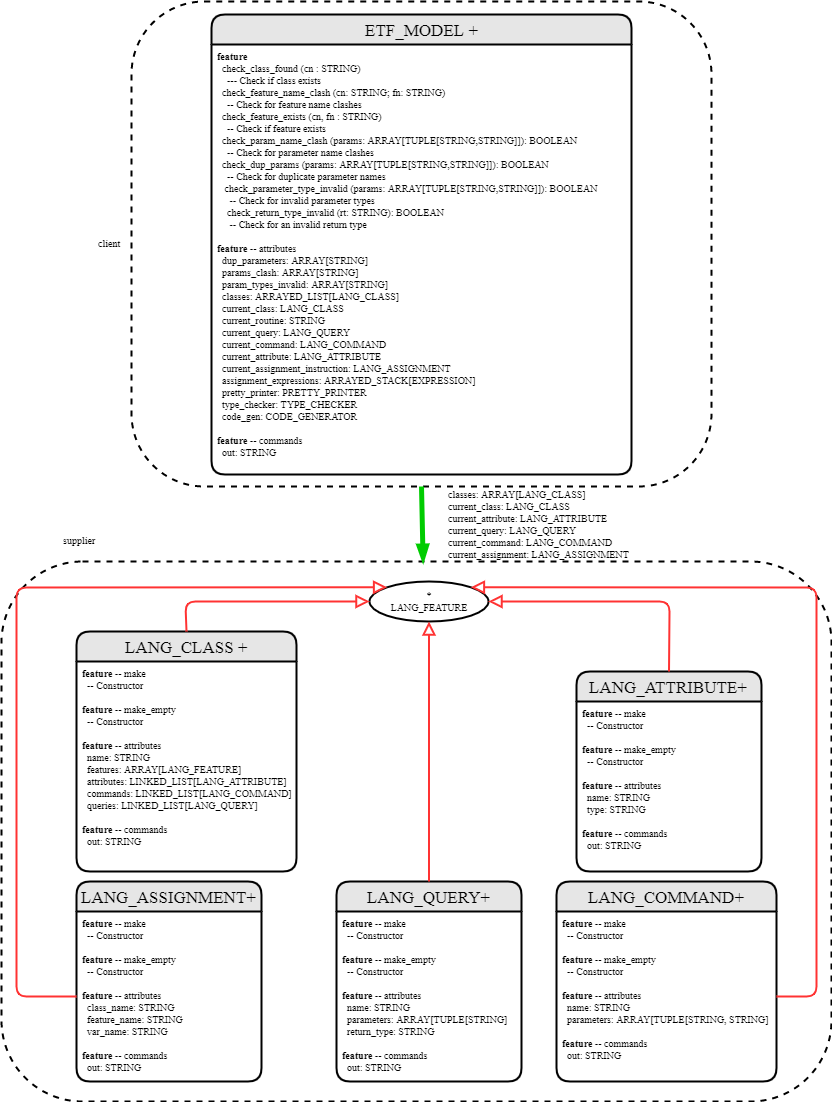
The programming language follows the syntax specified by the Context-free Grammars detailed in Appendix A.

# BON class diagram overview (architecture of the design)

The following BON diagram illustrates the relationships between various classes, including inheritance relationships and client-supplier relationships. The classes are displayed in concise view in this diagram. Subsequent diagrams will show relevant classes in expanded view (with contracts).



The following BON diagram describes the classes that represent the programming language structure. The relationships between the classes and the ETF\_MODEL class is also described, along with an expanded view of the ETF\_MODEL class which includes relevant features.



# 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.

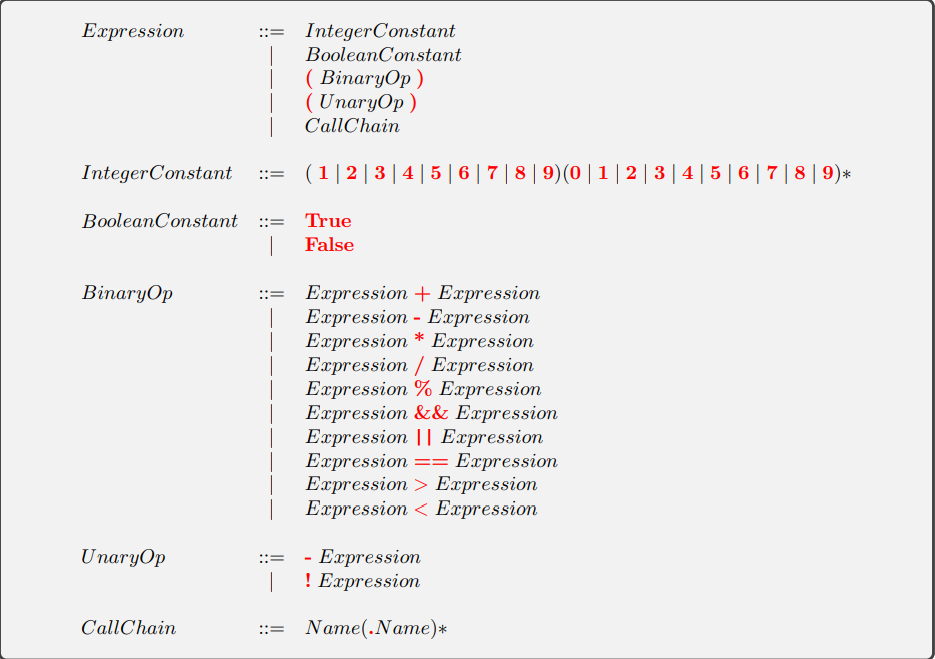
# Summary of Testing Procedures

This table describes the Acceptance Tests that were run against the code. The table also shows whether the test passed or failed.

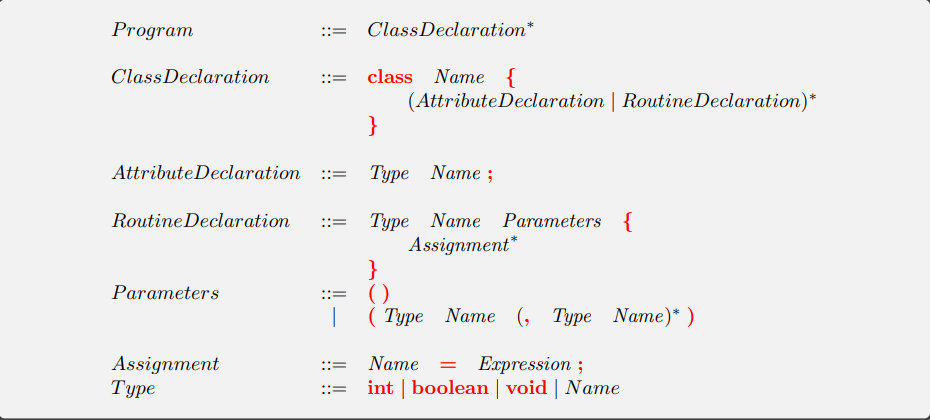
|  |  |  |
| --- | --- | --- |
| **Test file** | **Description** | **Passed** |
| *at01.txt* | Instructor provided acceptance test | ✓ |
| *at02.txt* | Instructor provided acceptance test | ✓ |
| *at03.txt* | Instructor provided acceptance test | ✓ |
| *at04.txt* | Instructor provided acceptance test | ✓ |
| *at05.txt* | Instructor provided acceptance test | ✓ |
| *at06.txt* | Instructor provided acceptance test | ✓ |
| *at07.txt* | Instructor provided acceptance test | ✓ |
| *at08.txt* | Instructor provided acceptance test | X |
| *at09.txt* | Instructor provided acceptance test | X |
| *at10.txt* | Instructor provided acceptance test | X |
| *at11.txt* | Instructor provided acceptance test | X |
| *at12.txt* | Instructor provided acceptance test | X |
| *at13.txt* | Instructor provided acceptance test | X |
| *at14.txt* | Instructor provided acceptance test | X |

# 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)

class interface

LANG\_ASSIGNMENT

create

make,

make\_empty

feature -- Attributes

class\_name: STRING\_8

feature\_name: STRING\_8

var\_name: STRING\_8

feature -- Commands

 -- New string containing terse printable representation

-- of current object

out: STRING\_8

set\_class\_name (n: STRING\_8)

set\_feature\_name (n: STRING\_8)

set\_var\_name (n: STRING\_8)

feature -- Constructors

make (cn, fn, vn: STRING\_8)

make\_empty

end -- class LANG\_ASSIGNMENT

class interface

LANG\_ATTRIBUTE

create

make,

make\_empty

feature -- Attributes

name: STRING\_8

type: STRING\_8

feature -- Commands

set\_name (n: STRING\_8)

set\_type (n: STRING\_8)

feature -- Constructor

make (an: STRING\_8; att\_type: STRING\_8)

make\_empty

feature -- Queries

out: STRING\_8

-- New string containing terse printable representation

-- of current object

end -- class LANG\_ATTRIBUTE

class interface

LANG\_CLASS

create

make,

make\_empty

feature -- Attributes

attributes: LINKED\_LIST [LANG\_ATTRIBUTE]

-- list of attributes

commands: LINKED\_LIST [LANG\_COMMAND]

-- list of commands

features: ARRAY [LANG\_FEATURE]

-- array of features

name: STRING\_8

-- class name

queries: LINKED\_LIST [LANG\_QUERY]

-- list of queries

feature -- Commands

-- add a new attribute to the class

add\_attribute (a: LANG\_ATTRIBUTE)

add\_command (a: LANG\_COMMAND)

add\_query (a: LANG\_QUERY)

feature -- Constructor

make (cn: STRING\_8)

make\_empty

feature -- Queries

out: STRING\_8

-- New string containing terse printable representation

-- of current object

end -- class LANG\_CLASS

class interface

LANG\_COMMAND

create

make,

make\_empty

feature -- Attributes

name: STRING\_8

-- command name

parameters: ARRAY [TUPLE [STRING\_8, STRING\_8]]

-- command parameters

feature -- Commands

set\_name (n: STRING\_8)

set\_params (p: ARRAY [TUPLE [STRING\_8, STRING\_8]])

feature -- Constructor

make (n: STRING\_8; params: ARRAY [TUPLE [pn: STRING\_8; ft: STRING\_8]])

make\_empty

feature -- Queries

out: STRING\_8

-- New string containing terse printable representation

-- of current object

end -- class LANG\_COMMAND

class interface

LANG\_QUERY

create

make\_empty,

make

feature -- Attributes

name: STRING\_8

-- command name

parameters: ARRAY [TUPLE [STRING\_8, STRING\_8]]

-- command parameters

return\_type: STRING\_8

-- return type

feature -- Commands

set\_name (n: STRING\_8)

set\_params (p: ARRAY [TUPLE [STRING\_8, STRING\_8]])

set\_return\_type (r: STRING\_8)

feature -- Constructor

make (fn: STRING\_8; ps: ARRAY [TUPLE [STRING\_8, STRING\_8]]; rt: STRING\_8)

make\_empty

feature -- Queries

out: STRING\_8

end -- class LANG\_QUERY

deferred class interface

LANG\_FEATURE

end -- class LANG\_FEATURE

deferred class interface

VISITOR

feature -- deferred features for the visitor pattern

visit\_addition (e: ADDITION)

visit\_boolean (e: BOOLEAN\_CONSTANT)

visit\_equals (e: EQUALS)

visit\_greater\_than (e: GREATER\_THAN)

visit\_integer (e: INTEGER\_CONSTANT)

visit\_less\_than (e: LESS\_THAN)

visit\_logical\_and (e: LOGICAL\_AND)

visit\_logical\_or (e: LOGICAL\_OR)

visit\_modulo (e: MODULO)

visit\_multiplication (e: MULTIPLICATION)

visit\_quotient (e: QUOTIENT)

visit\_subtraction (e: SUBTRACTION)

end -- class VISITOR

 deferred class interface

EXPRESSION

feature -- Commands

accept (v: VISITOR)

output: STRING\_8

end -- class EXPRESSION

deferred class interface

COMPOSITE\_EXPRESSION [G]

feature -- Attributes

children: LINKED\_LIST [G]

left: EXPRESSION

right: EXPRESSION

feature -- Commands

add\_children (nc: G)

end -- class COMPOSITE\_EXPRESSION

class interface

ADDITION

create

make

feature

accept (v: VISITOR)

make (lc, rc: EXPRESSION)

output: STRING\_8

invariant

binary\_operation: children.count = 2

end -- class ADDITION