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”

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).

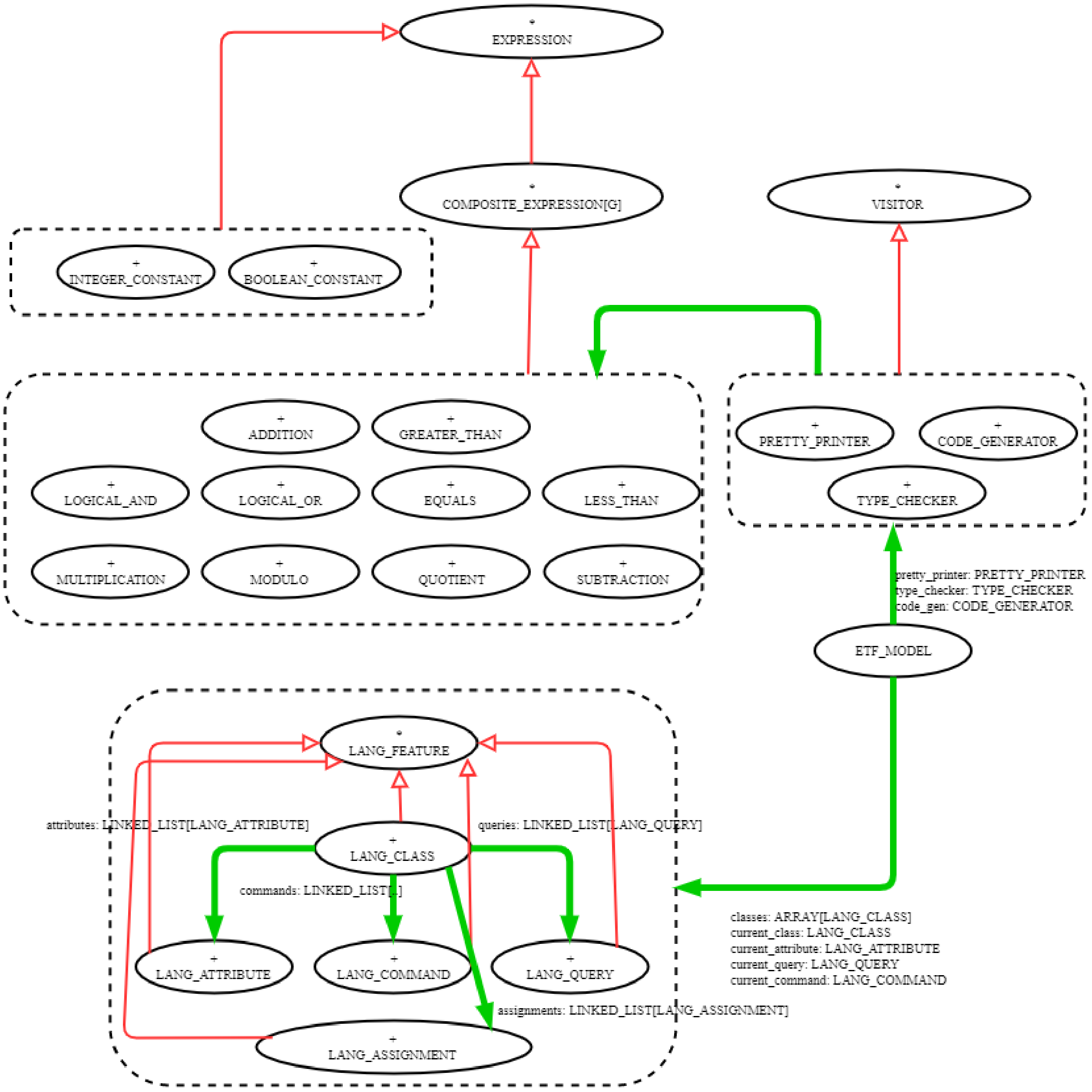
The language supports these operations via the use of Expressions, which are implemented using a context-free grammar detailed in Appendix A. The grammar may produce expressions which are not type-correct. A type checker was developed for the purpose of identifying expressions which are not type-correct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

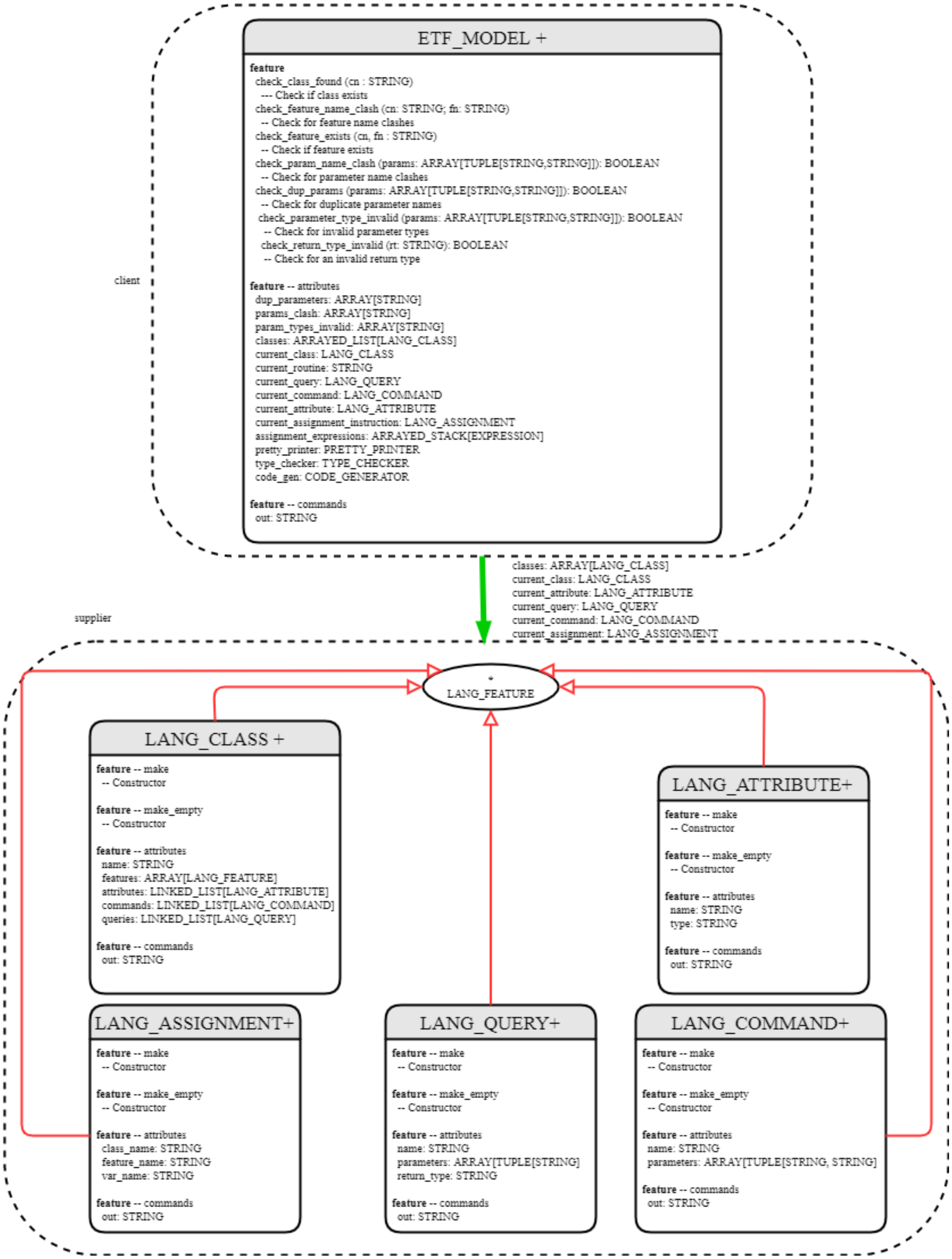
The programming language follows the syntax specified by the Context-free Grammars detailed in Appendix A.

2. BON class diagram overview (architecture of the design)

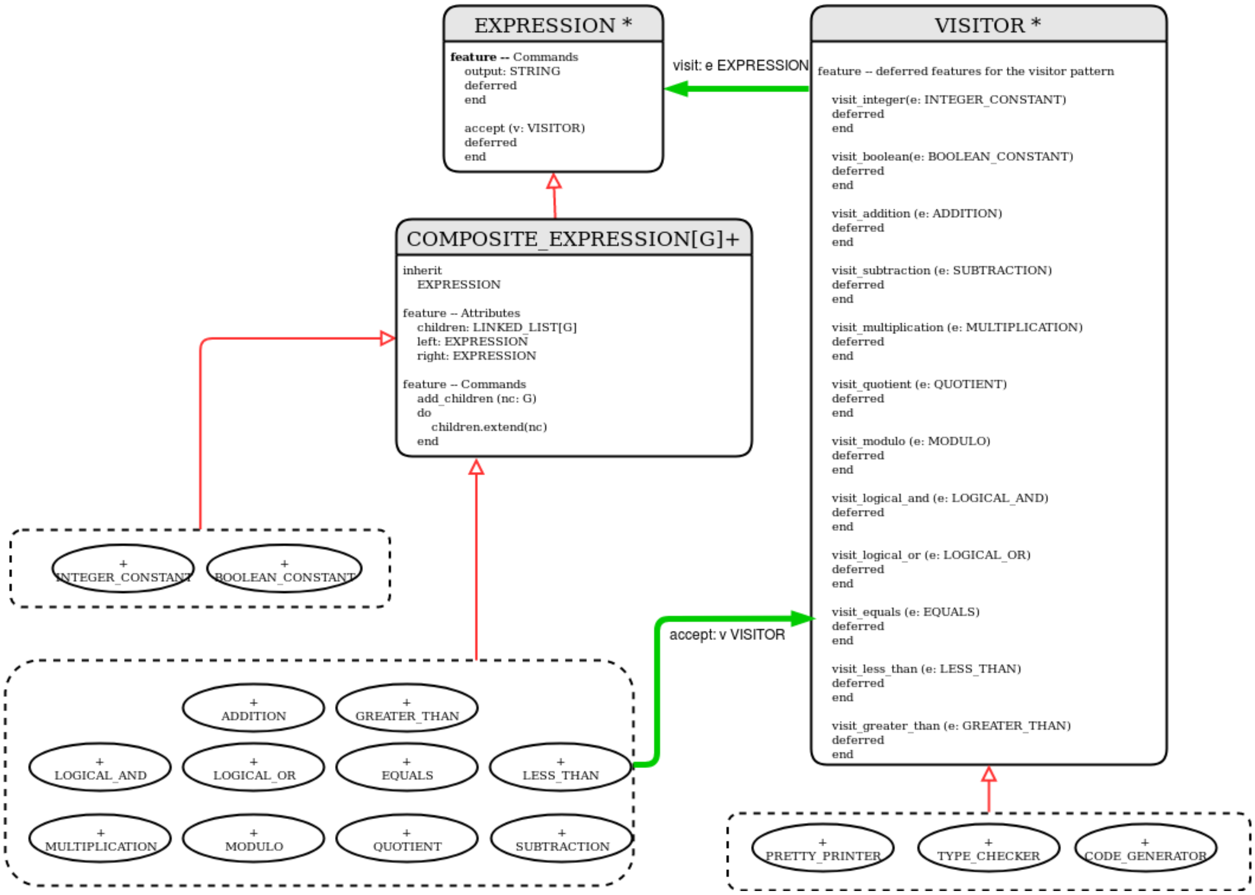
The following BON diagram illustrates the relationships between various classes, including inheritance relationships and client-supplier relationships. The classes are displayed in concise view in this diagram. Subsequent diagrams will show relevant classes in expanded view (with contracts).



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.


ETF_MODEL +
feature
check_class_found (cn : STRING)
--- Check if class exists
check_feature_name_clash (cn: STRING; fn: STRING)
-- Check for feature name clashes
check_feature_exists (cn, fn : STRING)
-- Check if feature exists
check_param_name_clash (params: ARRAY[TUPLE[STRING,STRING]]): BOOLEAN
-- Check for parameter name clashes
check_dup_params (params: ARRAY[TUPLE[STRING,STRING]]): BOOLEAN
-- Check for duplicate parameter names
check_parameter_type_invalid (params: ARRAY[TUPLE[STRING,STRING]]): BOOLEAN
-- Check for invalid parameter types
check_return_type_invalid (rt: STRING): BOOLEAN
-- Check for an invalid return type
feature -- attributes
dup_parameters: ARRAY[STRING]
params_clash: ARRAY[STRING]
param_types_invalid: ARRAY[STRING]
classes: ARRAYED_LIST[LANG_CLASS]
current_class: LANG_CLASS
current_routine: STRING
current_query: LANG_QUERY
current_command: LANG_COMMAND
current_attribute: LANG_ATTRIBUTE
current_assignment_instruction: LANG_ASSIGNMENT
assignment_expressions: ARRAYED_STACK[EXPRESSION]
pretty_printer: PRETTY_PRINTER
type_checker: TYPE_CHECKER
code_gen: CODE_GENERATOR
feature -- commands
out: STRING
client
classes: ARRAY[LANG_CLASS]
current_class: LANG_CLASS
current_attribute: LANG_ATTRIBUTE
current_query: LANG_QUERY
current_command: LANG_COMMAND
current_assignment: LANG_ASSIGNMENT
supplier
*
LANG_FEATURE
LANG_CLASS +
feature -- make
-- Constructor
feature -- make_empty
-- Constructor
feature -- attributes
name: STRING
features: ARRAY[LANG_FEATURE]
attributes: LINKED_LIST[LANG_ATTRIBUTE]
commands: LINKED_LIST[LANG_COMMAND]
queries: LINKED_LIST[LANG_QUERY]
feature -- commands
out: STRING
LANG_ATTRIBUTE+
feature -- make
-- Constructor
feature -- make_empty
-- Constructor
feature -- attributes
name: STRING
type: STRING
feature -- commands
out: STRING
LANG_ASSIGNMENT+
feature -- make
-- Constructor
feature -- make_empty
-- Constructor
feature -- attributes
class_name: STRING
feature_name: STRING
var_name: STRING
feature -- commands
out: STRING
LANG_QUERY+
feature -- make
-- Constructor
feature -- make_empty
-- Constructor
feature -- attributes
name: STRING
parameters: ARRAY[TUPLE[STRING]
return_type: STRING
feature -- commands
out: STRING
LANG_COMMAND+
feature -- make
-- Constructor
feature -- make_empty
-- Constructor
feature -- attributes
name: STRING
parameters: ARRAY[TUPLE[STRING, STRING]
feature -- commands
out: STRING

The following BON diagram represents the visitor and composite design patterns.



3. 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	

4. 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.

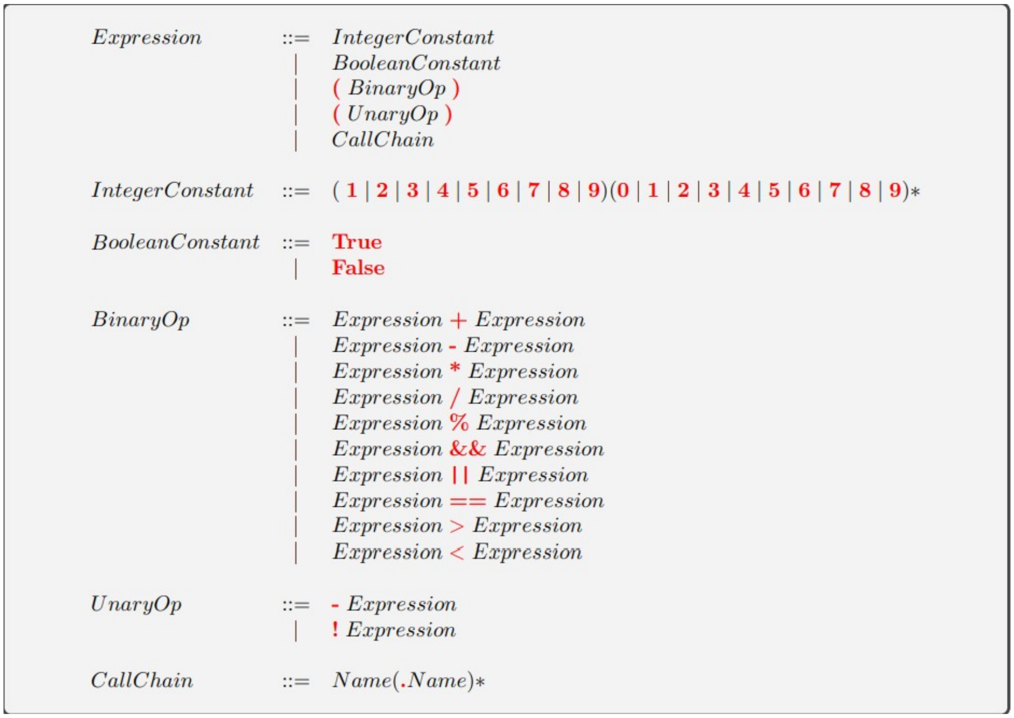
5. Summary of Testing Procedures

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

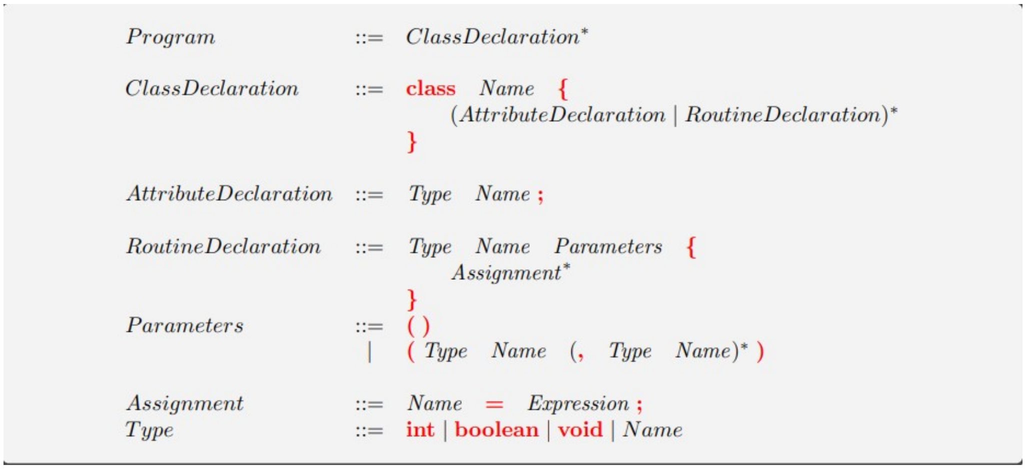
6. Appendix A (Context-free Grammars)

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



```
Expression        ::=  IntegerConstant
                   |   BooleanConstant
                   |   ( BinaryOp )
                   |   ( UnaryOp )
                   |   CallChain

IntegerConstant   ::=  (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)*

BooleanConstant   ::=  True
                   |   False

BinaryOp          ::=  Expression + Expression
                   |   Expression - Expression
                   |   Expression * Expression
                   |   Expression / Expression
                   |   Expression % Expression
                   |   Expression && Expression
                   |   Expression || Expression
                   |   Expression == Expression
                   |   Expression > Expression
                   |   Expression < Expression

UnaryOp           ::=  - Expression
                   |   ! Expression

CallChain         ::=  Name(.Name)*
```

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



```
Program               ::=  ClassDeclaration*

ClassDeclaration      ::=  class   Name   {
                                (AttributeDeclaration | RoutineDeclaration)*
                           }

AttributeDeclaration  ::=  Type   Name ;

RoutineDeclaration    ::=  Type   Name   Parameters   {
                                Assignment*
                           }
Parameters            ::=  ( )
                       |   ( Type   Name   (,   Type   Name)* )

Assignment            ::=  Name   =   Expression ;
Type                  ::=  int | boolean | void | Name
```

7. 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
```