GAMMA

Γαγ

Gamma Language Reference Manual

Matthew H. Maycock - mhm2159@columbia.edu
Arthy Sundaram - as4304@columbia.edu
Weiyuan Li - wl2453@columbia.edu
Ben Caimano - blc2129@columbia.edu

October 28, 2013

Contents

1 Introduction

1.1 Why GAMMA? – The Core Concept

We propose to implement an elegant yet secure general purpose object-oriented programming language. Interesting features have been selected from the history of object-oriented programming and will be combined with the familiar ideas and style of modern languages.

GAMMA combines three disparate but equally important tenants:

1. Purely object-oriented

   GAMMA brings to the table a purely object oriented programming language where every type is modeled as an object–including the standard primitives. Integers, Strings, Arrays, and other types may be expressed in the standard fashion but are objects behind the scenes and can be treated as such.

2. Controllable

   GAMMA provides innate security by choosing object level access control as opposed to class level access specifiers. Private and protected members of one object are inaccessible to other objects of the same type. Overriding is not allowed. No subclass can turn your functionality on its head.

3. Versatile

   GAMMA allows programmers to place "refinement methods" inside their code. Alone these methods do nothing, but may be defined by subclasses so as to extend functionality at certain important positions. The idea of refining methods are inherited from the great work that went into the BETA programming language and were further explored in BetaJava and are a part of PLTScheme/Racket (and, of course, there are perlisms to copy the behavior, as is to be expected from *that* community).

   Refinement in GAMMA is different than the above, though – the GAMMA philosophy of refinement is that methods themselves are singular objects of computation associated with classes and these objects themselves provide an extendable interface.

   In addition to refinement, anonymous instantiation allows for extension of your classes in a quick & easy fashion.

1.2 The Motivation Behind GAMMA

GAMMA is a reaction to the various object-oriented languages that came before it. Obtuse syntax, flaws in security, and awkward implementations plague the average object-oriented language. GAMMA is intended as a step toward ease and comfort as an object-oriented programmer.

The first goal is to make an object-oriented language that is comfortable in its own skin. It should naturally lend itself to constructing API-layers and

abstracting general models. It should serve the programmer towards their goal instead of exerting unnecessary effort through verbosity and awkwardness of structure.

The second goal is to make a language that is stable and controllable. The programmer in the lowest abstraction layer has control over how those higher may procede. Unexpected runtime behavior should be reduced through firmness of semantic structure and debugging should be a straight-forward process due to pure object oriented nature of GAMMA.

2 Lexical Elements

2.1 Whitespace

The new line (line feed), form feed, carriage return, and vertical tab characters will all be treated equivalently as vertical whitespace. Tokens are separated by horizontal (space, tab) and vertical (see previous remark) whitespace of any length (including zero).

2.2 Identifiers

Identifiers are used for the identification of variables, methods and types. An identifer is a sequence of alphanumeric characters, uppercase and lowercase, and underscores. A type identifier must start with an uppercase letter; all others must start with a lower case letter. Additionally, the lexeme of a left bracket followed immediately by a right bracket – `[]` – may appear at the end of a type identifier in certain contexts, and that there may be multiple present in this case (denoting arrays, arrays of arrays, etc, etc). The legal contexts for such will be described later.

2.3 Keywords

The following words are reserved keywords. They may not be used as identifiers:

```
    and      class      else      elsif    extends    false
     if       init      main       nand      new       nor
    not      null        or     private  protected  public
refinable  refine  refinement   return     super     this
     to       true      void      while      xor
```

2.4 Operators

There are a large number of (mostly binary) operators:

```
 =    =/=   <>    <    <=    >      >=
 +     -    *     /    %     ^      :=
+=    -=   *=    /=   %=    ^=
and   or   not   nand  nor   xor   refinable
```

2.5 Literal Classes

A literal class is a value that may be expressed in code without the use of the new keyword. These are the fundamental units of program.

2.5.1 Integer Literals

An integer literal is a sequence of digits. It may be prefaced by a unary minus symbol. For example:

- `777`
- `42`
- `2`
- `-999`
- `0001`

2.5.2 Float Literals

A float literal is a sequence of digits and exactly one decimal point/period. It must have at least one digit before the decimal point and at least one digit after the decimal point. It may also be prefaced by a unary minus symbol. For example:

- `1.0`
- `-0.567`
- `10000.1`
- `00004.70000`
- `12345.6789`

2.5.3 Boolean Literals

A boolean literal is a single keyword, either `true` or `false`.

2.5.4 String Literals

A string literal consists of a sequence of characters enclosed in double quotes. Note that a string literal can have the new line escape sequence within it (among others, see below), but cannot have a new line (line feed), form feed, carriage return, or vertical tab within it; nor can it have the end of file. Please note that the sequence may be of length zero. For example:

- `"Yellow matter custard"`
- `""`
- `"Dripping\n   from a dead"`
- `"'s 3y3"`

The following are the escape sequences available within a string literal; a backslash followed by a character outside of those below is an error.

- `\a` - u0007/alert/BEL

- `\b` - u0008/backspace/BB
- `\f` - u000c/form feed/FF
- `\n` - u000a/linefeed/LF
- `\r` - u000d/carriage return/CR
- `\t` - u0009/horizontal tab/HT
- `\v` - u000b/vertical tab/VT
- `\'` - u0027/single quote
- `\"` - u0022/double quote
- `\\` - u005c/backslash
- `\0` - u0000/null character/NUL

2.6 Comments

Comments begin with the sequence `/*` and end with `*/`. Comments nest within each other. Comments must be closed before the end of file is reached.

2.7 Separators

The following characters delineate various aspects of program organization (such as method arguments, array indexing, blocks, and expressions):

```
[  ]  (  )  {  }  ,  ;
```

A notable exception is that `[]` itself is a lexeme related to array types and there can be no space between the two characters in this regard.

2.8 Whitespace and Noncanonical Gamma

Canonical Gamma, which is described by this document's syntax and grammar sections, ignores whitespace outside of string literals. Gamma code with rigid whitespace (pythonesque Gamma) – a la python and without `;`, `{`, or `}` – can transformed to canonical Gamma easily. Such transformation respects the following rules:

- Tab characters are equivalent to eight spaces
- Wherever a `{` could be used to start a scope, a `:` can be used instead; in such case a closing `}` is not necessary as it will be inferred by the indentation level
- After starting a scope with `:` the next line with non-whitespace characters determines the indentation level of that scope.

  - If there is no such line (end of file) then the scope is assumed to end and is equivalent to `{}`.
  - If the next such line is indented no more than the line introducing the scope, then the scope is again equivalent to `{}`.
  - If the next line is indented more than the line introducing the scope then this sets the indentation level of the scope – all statements in this scope *must* be at the same exact level of indentation. The scope continues until end of file or the indentation returns to the indentation level of an outer scope.

- If the line after a statement *not* ending in a colon is indented more than the given line, then it is considered a continuation of that statement.
- At the end of a statement (either the end of the line starting the statement, or subsequent lines that are indented more as per the rule above), a new line is equivalent to a semicolon `;`.
- If a scope is explicitly marked with `{` and `}` then all further scoping within that scope must be handled in a whitespace insensitive manner via the use of `{`, `}`, `;`.

3 Semantics

3.1 Types and Variables

Every *variable* in Gamma is declared with a *type* and an *identifier*. The typing is static and will always be known at compile time for every variable. The variable itself holds a reference to an instance of that type. At compile time, each variable reserves space for one reference to an instance of that type; during run time, each instantiation reserves space for one instance of that type (i.e. *not* a reference but the actual object). To be an instance of a type, an instance must be an instance of the class of the same name as that type or an instance of one of the set of descendants (i.e. a subclass defined via `extends` or within the transitive closure therein) of that class. While it is possible for any reference to be `null`, the `null` value is not an instance of a class – it cannot execute methods or access any fields. `null` represents the lack of an instance. For the purposes of method and refinement return types there is a special keyword, `void`, that allows a method or refinement to use the `return` keyword without an expression and thus not produce a value.

3.1.1 Array Types

When specifying the type of a variable, the type identifier may be followed by one or more `[]` lexemes. The lexeme implies that the type is an *array type* of the *element type* that precedes it in the identifier. So if `BankAccount` is the name of a class (which is associated with a type), then `BankAccount` is the type of a zero dimensional array – i.e. the type of `BankAccount`. The type associated with `BankAccount[]` is the that of an one-dimensional array of elements of class `BankAccount`. `BankAccount[][][][]` can be considered the type of an array of four dimension and element class `BankAccount`. It is in fact an array of one dimension which has elements of type `BankAccount[][][]`. Elements of an array are accessed via an expression resulting in an array followed by a left bracket `[`, an expression producing an offset index of zero or greater, and a right bracket `]`. Elements are of one dimension less and so are themselves either arrays or are individual instances of the overall class/type involved (i.e. `BankAccount`), or of course `null`.

3.2 Classes, Subclasses, and Their Members

GAMMA is a pure object-oriented language, which means every value is an object – with the exception that `null` represents the lack of an instance / object to refer to and `this` is a special reference for the object of the current context; the use of `this` is only useful inside the context of a method, `init`, or refinement and so cannot be used in a `main` where, technically, it is `null` but its use will be reported with a more descriptive error. `init` and `main` are defined later.

A class always extends another class; a class inherits all of its superclass's methods and may refine the methods of its superclass. A class must contain a

constructor routine named *init* and it must invoke its superclass's constructor via the super keyword – either directly or transitively by referring to other constructors within the class. In the scope of every class, the keyword `this` explicitly refers to the instance itself. Additionally, a class contains three sets of *members* organized in *private*, *protected*, and *public* sections. Members may be either variables or methods. Members in the public section may be accessed (see syntax) by any other object. Members of the protected section may be accessed only by an object of that type or a descendant (i.e. a subtype defined transitively via the `extends` relation). Private members are only accessible by the members defined in that class (and are not accessible to descendants). Note that access is enforced at object boundaries, not class boundaries – two `BankAccount` objects of the same exact type cannot access each other's balance, which is in fact possible in both Java & C++, among others. Likewise if `SavingsAccount` extends `BankAccount`, an object of savings account can access the protected instance members of `SavingsAccount` related to its own data, but *cannot* access those of another object of similar type (`BankAccount` or a type derived from it).

3.2.1 The Object Class

The Object class is the superclass of the entire class hierarchy in GAMMA. All objects directly or indirectly inherit from it and share its methods. By default, class declarations without extending explicitly are subclasses of Object.

3.2.2 The Literal Classes

There are several *literal classes* that contain uniquely identified members (via their literal representation). These classes come with methods developed for most operators. They are also all subclasses of Object.

3.2.3 Anonymous Classes

A class can be anonymously subclassed (such must happen in the context of instantiation) via refinements. They are a subclass of the class they refine, and the objects are a subtype of that type. Note that references are copied at anonymous instantiation, not values.

3.3 Methods

A method is a reusable subdivision of code that takes multiple (possibly zero) values as arguments and can either return a value of the type specified for the method, return null, or not return any value in the case that the return type is `void`.

It is a semantic error for two methods of a class to have the same signature – which is the return type, the name, and the type sequence for the arguments. It is also a semantic error for two method signatures to only differ in return type in a given class.

3.3.1 Operators

Since all variables are objects, every operator is in truth a method called from one of its operands with the other operands as arguments – with the notable exception of the assignment operators which operate at the language level as they deal not with operations but with the maintenance of references (but even then they use methods as `+=` uses the method for `+` – but the assignment part itself does not use any methods). If an operator is not usable with a certain literal class, then it will not have the method implemented as a member. Classes that are not literal classes may implement those methods so as to allow that operation to be performed between two instances of the appropriate type.

3.4 Refinements

Methods and constructors of a class can have *refine* statements placed in their bodies. Subclasses must implement *refinements*, special methods that are called in place of their superclass' refine statements, unless the refinements are guarded with a boolean check via the `refinable` operator for their existence – in which case their implementation is optional.

It is a semantic error for two refinements of a method to have the same signature – which is the return type, the method they refine, the refinement name, and the type sequence for the arguments. It is also a semantic error for two method signatures to only differ in return type in a given class.

A refinement cannot be implemented in a class derived by a subclass, it must be provided if at all in the subclass. If it is desired that further subclassing should handle refinement, then these further refinements can be invoked inside the refinements themselves (syntactic sugar will make this easier in future releases). Note that refining within a refinement results in a refinement of the same method. That is, using `refine extra(someArg) to String` inside the refinement `String toString.extra(someType someArg)` will (possibly, if not guarded) require the next level of subclassing to implement the extra refinement for toString.

3.5 Constructors (init)

Constructors are invoked to arrange the state of an object during instantiation and accept the arguments used for such. It is a semantic error for two constructors to have the same signature – that is the same type sequence.

3.6 Main

Each class can define at most one `main` method to be executed when that class will 'start the program execution' so to speak. Main methods are not instance methods and cannot refer to instance data. These are the only 'static' methods allowed in the Java sense of the word. It is a semantic error for the main to have arguments other than a String array or for more than one main to be defined in a class.

3.7 Expressions and Statements

The fundamental nature of an expression is that it generates a value. A statement can be a call to an expression, thus a method or a variable. Not every statement is an expression, however.

4 Syntax

The syntaxic structures presented in this section may have optional elements. If an element is optional, it will be wrapped in the lexemes `<<` and `>>`. This grouping may nest. On rare occasions, a feature of the syntax will allow for truly alternate elements. The elements are presented in the lexemes `{{` and `}}`, each feature is seperated by the lexeme `|`. If an optional element may be repeated without limit, it will finish with the lexeme `...`.

4.1 Statement Grouping via Bodies

A body of statements is a series of statements bounded by curly braces.

```
{
    <<stmt1_statement>>
    <<stmt2_statement>>
    <<...>>
}
```

This is pattern is elementry to write.

```
{
    Mouse mouse = new Mouse();
    mouse.click();
    mouse.click_fast();
    mouse.click("Screen won't respond");
    mouse.defenestrate();
}
```

Example 1: Statement Grouping of a Typical Interface Simulator

4.2 Variables

4.2.1 Variable Assignment

Assigning an instance to a variable requires an expression and a variable identifier:

```
var_identifier := val_expr;
```

If we wanted to assign instances of Integer for our pythagorean theorem, we'd do it like so:

```
a := 3;
b := 4;
```

Example 2: Variable Assignment for the Pythagorean Theorem

4.2.2 Variable Declaration

Declaring a variable requires a type and a list of identifiers deliminated by commas. Each identifier may be followed by the assignment operator and an expression so as to combine assignment and declaration.

```
var_type var1_identifier << := val1_expr >> << , var2_identifier <<
    := val2_expr >> >> <<...>>;
```

If we wanted to declare variables for the pythagorean theorem, we would do it like so:

```
Float a, b, c;
```

Example 3: Variable Initialization for the Pythagorean Theorem

4.2.3 Array Declaration

Declaring an array is almost the same as declaring a normal variable, simply add square brackets after the type. Remember that an array is a type as well, and so the source type can very well be an array of arrays of a certain type before you make declaration. And note that not all dimensions need be given (and for you may want to later make non-rectangular multi dimensional arrays).

```
element_type[]...[] array_identifier << := new element_type[](
    dim1_expr,...,dimN_expr) >>;
```

If we wanted a set of triangles to operate on, for instance:

```
Triangle[] triangles := new Triangle[](42);
```

Example 4: Array Declaration and Instantiation of Many Triangles

Or perhaps, we want to index them by their short sides and initialize them later:

```
Triangle[][] triangles;
```

Example 5: Array Declaration of a 2-Degree Triangle Array

4.2.4 Array Dereferencing

To dereference an instance of an array type down to an instance its element type, place the index of the element instance inside the array instance between [and] lexemes after the variable identifier. This syntax can be used to provide a variable for use in assignment or expressions.

```
var_identifier[dim1_index]...[dimN_index]
```

Perhaps we care about the fifth triangle in our array from before for some reason.

```
Triangle my_triangle := triangles[4];
```

Example 6: Array Dereferencing a Triangle

4.3 Methods

4.3.1 Method Invocation

Invoking a method requires at least an identifier for the method of the current context (i.e. implicit `this` receiver). The instance that the method is invoked upon can be provided as an expression. If it is not provided, the method is invoked upon `this`.

```
<< instance_expr.>>method_identifier(<<arg1_expr>> <<, arg2_expr>>
    <<...>>)
```

Finishing our pythagorean example, we use method invocations and assignment to calculate the length of our third side, c.

```
c := ((a.power(2)).plus(b.power(2))).power(0.5);
```

Example 7: Method Invocation for the Pythagorean Theorem Using Methods

4.3.2 Method Invocation Using Operators

Alternatively, certain base methods allow for the use of more familiar binary operators in place of a method invocation.

```
op1_expr operator op2_expr
```

Using operators has advantages in clarity and succinctness even if the end result is the same.

```
c := ( a^2 + b^2 )^0.5;
```

Example 8: Method Invocation for the Pythagorean Theorem Using Operators

4.3.3 Operator Precedence

In the previous examples, parentheses were used heavily in a context not directly related to method invocation. Parentheses have one additional function: they modify precedence among operators. Every operator has a precidence in relation to its fellow operators. Operators of higher precedence are enacted first. Please consider the following table for determining precidence:

:=	+=	-=	*=	/=	%=	^=
or	xor	nor				
and	nand					
=	<>	=/=				
>	<	>=	<=			
+	-					
*	/	%				
unary minus						
not	^					
array dereferencing		(	)			
method invocation						

Table 1: Operator Precedence

4.3.4 Method Declaration & Definition

A method definition begins with the return type – either a type (possibly an n-dimensional array) or void. The identifier for the function is followed by a pair of parentheses that may enclose the parameter declarations. There is one type and one identifier for each parameter; and they are delimited by commas. Following the parentheses are a pair of braces around the body of the method. There can be zero or more statements in the body. Additionally, refinements may be placed throughout the statements.

```
{{return_type | Void}} method_identifier (<<arg1_type
    arg1_identifier>> <<, arg2_type arg2_identifier>> <<...>>)
    method_body
```

Finally, we may define a method to do our pythagorean theorem calculation.

```
Float pythagorean_theorem(Float a, Float b){
    Float c;
    c := ( a^2 + b^2 )^0.5;
    return c;
}
```

Example 9: Method Definition for the Pythagorean Theorem

4.4 Classes

4.5 Section Definition

Every class always has at least one section that denotes members in a certain access level. A section resembles a body, it has the lexemes `{` and `}` surrounding a set of variable and method declarations, including `init` methods.

```
{
    <<{{method1_decl | var1_decl | init1_decl}}>>
    <<{{method2_decl | var2_decl | init2_decl}}>>
    <<...>>
}
```

4.5.1 Class Declaration & Definition

A class definition always starts with the keyword `class` followed by a type (i.e. capitalized) identifier. There can be no brackets at the end of the identifier, and so this is a case where the type must be purely alphanumeric mixed with underscores. It optionally has the keyword `extends` followed by the identifier of the superclass. What follows is the class body enclosed in braces: an optional `main` method, the three access-level member sections, and refinements. There may be `init` methods in any of the three sections, and there must be (semantically enforced, not syntactically) an `init` method either in the protected or public section (for otherwise there would be no way to generate instances).

While the grammar allows multiple main methods to be defined in a class, any more than one will result in an error during compilation.

```
class class_identifier <<extends superclass_identifier>> {
    <<main_method>>
    <<{{private | protected | public | refinement}} section1>>
    <<{{private | protected | public | refinement}} section1>>
    <<...>>
}
```

Let's make a basic geometric shape class in anticipation of later examples. We have private members, two access-level sections and an init method. No extends is specified, so it is assumed to inherit from Object.

```
class Geometric_Shape {
    private {
        String name;
        Float area;
        Float circumfrence;
    }
    public {
        init (String name) {
            this.name = name;
            if (refinable(improve_name)) {
              this.name += refine improve_name() to String;
            }
            return;
        }
        Float get_area() {
            Float area;
            area := refine custom_area() to Float;
        }
    }
}
```

Example 10: Class Declaration for a Geometric Shape class

4.5.2 Class Instantiation

Making a new instance of a class is simple.

```
new class_identifier(<<arg1_expr>> <<,arg2_expr>> <<...>>)
```

For instance:

```
Geometric_Shape = new Geometric_Shape("circle");
```

Example 11: Class Instantiation for a Geometric Shape class

4.5.3 Anonymous Classes

An anonymous class definition is used in the instantiation of the class and can only provide refinements, no additional public, protected, or private members. Additionally no init or main can be given.

```
new superclass_identifier(<<arg1_expr>> <<,arg2_expr>> <<...>>) {
    <<refinements>>
}
```

4.6 Conditional Structures

4.6.1 If Statements

The fundamental unit of an if statement is a keyword, followed by an expression between parentheses to test, and then a body of statements between curly braces. The first keyword is always `if`, each additional condition to be tested in sequence has the keyword `elsif` and a final body of statements may optionally come after the keyword `else`.

```
if (test1_expr) if1_body
<<elsif (test2_expr) if2_body>>
<<elsif(test3_expr) if3_body>>
<<...>>
<<else if4_body>>
```

4.6.2 While Statements

A while statement consists of only the `while` keyword, a test expression and a body.

```
while(test_expr) while_body
```

4.7 Refinements

4.7.1 The Refine Invocation

A refine invocation will eventually evaluate to an expression as long as the appropriate refinement is implemented. It is formed by using the keyword `refine`, the identifier for the refinement, the keyword `to`, and the type for the desired expression. Note that a method can only invoke its own refinements, not others – but refinements defined *within* a class can be called. This is done in addition to normal invocation. Also note that all overloaded methods of the same name share the same refinements.

```
refine refine_identifier to refine_type
```

4.7.2 The Refinable Test

The original programmer cannot garuantee that future extenders will implement the refinement. If it is allowable that the refinement does not happen, then the programmer can use the `refinable` keyword as a callable identifier that evaluates to a Boolean instance. If the programmer contrives a situation where the compiler recognizes that a refinement is guarded but still executes a refine despite the refinement not existing, a runtime error will result.

```
refinable(refinement_identifier)
```

4.7.3 The Refinement Declaration

To declare a refinement, declare a method in your subclass' refinement section with the special identifier `supermethod_identifier.refinement_identifier`.

5 Operators and Literal Types

The following defines the approved behaviour for each combination of operator and literal type. If the literal type is not listed for a certain operator, the operator's behaviour for the literal is undefined. These operators never take operands of different types.

5.1 The Operator =

5.1.1 Integer

If two Integer instances have the same value, `=` returns `true`. If they do not have the same value, it returns `false`.

5.1.2 Float

If two Float instances have an absolute difference of less than or equal to an epsilon of 2^{-24}, `=` returns `true`. If the absolute difference is greater than that epsilon, it returns `false`.

5.1.3 Boolean

If two Boolean instances have the same keyword, either `true` or `false`, `=` returns `true`. If their keyword differs, it returns `false`.

5.1.4 String

If two String instances have the same sequence of characters, `=` returns `true`. If their sequence of characters differs, it returns `false`.

5.2 The Operators =/= and <>

5.2.1 Integer

If two Integer instances have a different value, `=/=` and `<>` return `true`. If they do have the same value, they returns `false`.

5.2.2 Float

If two Float instances have an absolute difference of greater than than an epsilon of 2^{-24}, `=` returns `true`. If the absolute difference is less than or equal to that epsilon, it returns `false`.

5.2.3 Boolean

If two Boolean instances have different keywords, `=/=` and `<>` return `true`. If their keywords are the same, they return `false`.

5.2.4 String

If two String instances have the different sequences of characters, `=/=` and `<>` return `true`. If their sequence of characters is the same, they return `false`.

5.3 The Operator <

5.3.1 Integer and float

If the left operand is less than the right operand, `<` returns `true`. If the right operand is less than or equal to the left operand, it returns `false`.

5.3.2 String

If the left operand comes before the right operand in dictionary order, `<` returns `true`. If the left operand comes after the right operand in dictionary order , it returns `false`. If the two operands have the same sequence of characters, it returns `false`.

5.4 The Operator >

5.4.1 Integer and float

If the left operand is greater than the right operand, `>` returns `true`. If the right operand is greater than or equal to the left operand, it returns `false`.

5.4.2 String

If the left operand comes after the right operand in dictionary order, `<` returns `true`. If the left operand comes before the right operand in dictionary order , it returns `false`. If the two operands have the same sequence of characters, it returns `false`.

5.5 The Operator <=

5.5.1 Integer and float

If the left operand is less than or equal to the right operand, `<` returns `true`. If the right operand is less than the left operand, it returns `false`.

5.5.2 String

If the left operand comes before the right operand in dictionary order, `<` returns `true`. If the left operand comes after the right operand in dictionary order , it returns `false`. If the two operands have the same sequence of characters, it returns `true`.

5.6 The Operator >=

5.6.1 Integer and float

If the left operand is greater than or equal to the right operand, `>` returns `true`. If the right operand is greater than the left operand, it returns `false`.

5.6.2 String

If the left operand comes after the right operand in dictionary order, `<` returns `true`. If the left operand comes before the right operand in dictionary order , it returns `false`. If the two operands have the same sequence of characters, it returns `true`. ¡¡¡¡¡¡¡ HEAD

5.7 The Operator +

5.7.1 Integer and Float

`+` returns the sum of the two operands.

5.7.2 String

`+` returns the concatenation of the right operand onto the end of the left operand.

5.8 The Operator -

5.8.1 Integer and Float

`-` returns the right operand subtracted from the left operand.

5.9 The Operator *

5.9.1 Integer and Float

`*` returns the product of the two operands.

5.10 The Operator /

5.10.1 Integer and Float

`/` returns the left operand divided by the right operand.

5.11 The Operator %

5.11.1 Integer and Float

`%` returns the modulo of the left operand by the right operand.

5.12 The Operator ^

5.12.1 Integer and Float

^ returns the left operand raised to the power of the right operand.

5.13 The Operator :=

5.13.1 Integer, Float, Boolean, and String

:= assigns the right operand to the left operand and returns the value of the the right operand. This is the sole right precedence operator.

5.14 The Operators +=, -=, *=, /= %=, and ^=

5.14.1 Integer, Float, Boolean, and String

This set of operators first applies the operator indicated by the first character of each operator as normal on the operands. It then assigns this value to its left operand.

5.15 The Operator and

5.15.1 Boolean

and returns the conjunction of the operands.

5.16 The Operator or

5.16.1 Boolean

or returns the disjunction of the operands.

5.17 The Operator not

5.17.1 Boolean

not returns the negation of the operands.

5.18 The Operator nand

5.18.1 Boolean

nand returns the negation of the conjunction of the operands.

5.19 The Operator nor

5.19.1 Boolean

nor returns the negation of the disjunction of the operands.

5.20 The Operator `xor`

5.20.1 Boolean

`xor` returns the exclusive disjunction of the operands.

5.21 The Operator `refinable`

5.21.1 Boolean

`refinable` returns `true` if the refinement is implemented in the current subclass. It returns `false` otherwise.

6 Grammar

The following conventions are taken:

- Sequential semicolons (even separated by whitespace) are treated as one.
- the 'digit' class of characters are the numerical digits zero through nine
- the 'upper' class of characters are the upper case roman letters
- the 'lower' class of characters are the lower case roman letters
- the 'ualphanum' class of characters consists of the digit, upper, and lower classes together with the underscore
- a program is a collection of classes; this grammar describes solely classes
- the argument to main is semantically enforced after parsing; its presence here is meant to increase readability

The grammar follows:

- *Classs may extend another class or default to extending Object*

⟨class⟩ ⇒
    class ⟨class id⟩⟨extend⟩ **{** ⟨class section⟩***** **}**
⟨extend⟩ ⇒
    ϵ
  | **extends** ⟨class id⟩

- *Sections – private protected public refinements and main*

⟨class section⟩ ⇒
    ⟨refinement⟩
  | ⟨access group⟩
  | ⟨main⟩

- *Refinements are named method dot refinement*

⟨refinement⟩ ⇒
    refinement { ⟨refine⟩***** **}**
⟨refine⟩ ⇒
    ⟨return type⟩⟨var id⟩**.**⟨var id⟩⟨params⟩ **{** ⟨statement⟩***** **}**

- *Access groups contain all the members of a class*

⟨access group⟩ ⇒
    ⟨access type⟩ **{** ⟨member⟩***** **}**
⟨access type⟩ ⇒
    private
  | **protected**

| **public**

⟨member⟩ ⇒
    ⟨var decl⟩
  | ⟨method⟩
  | ⟨init⟩

⟨method⟩ ⇒
    ⟨return type⟩⟨var id⟩⟨params⟩ **{** ⟨statement⟩***** **}**

⟨init⟩ ⇒
    init ⟨params⟩ **{** ⟨statement⟩***** **}**

• *Main is special – not instance data starts execution*

⟨main⟩ ⇒
    main (String[] ⟨var id⟩ **) {** ⟨statement⟩***** **}**

• *Finally the meat and potatoes*

⟨statement⟩ ⇒
    ⟨var decl⟩ **;**
  | ⟨var decl⟩ **:=** ⟨expression⟩ **;**
  | ⟨super⟩ **;**
  | ⟨return⟩ **;**
  | ⟨conditional⟩
  | ⟨loop⟩
  | ⟨expression⟩ **;**

• *Super invocation is so we can do constructor chaining*

⟨super⟩ ⇒
    super ⟨args⟩

• *Methods yield values (or just exit for void/init/main)*

⟨return⟩ ⇒
    return
  | **return** ⟨expression⟩

• *Basic control structures*

⟨conditional⟩ ⇒
    if (⟨expression⟩ **) {** ⟨statement⟩***** **}** ⟨else⟩

⟨else⟩ ⇒
    ϵ
  | ⟨elseif⟩ **else {** ⟨statement⟩***** **}**

⟨elseif⟩ ⇒
    ϵ
  | ⟨elseif⟩ **elsif (** ⟨expression⟩ **) {** ⟨statement⟩***** **}**

⟨loop⟩ ⇒
    while (⟨expression⟩ **) {** ⟨statement⟩***** **}**

• *Anything that can result in a value*
⟨expression⟩ ⇒
    ⟨assignment⟩
  | ⟨invocation⟩
  | ⟨field⟩
  | ⟨var id⟩
  | ⟨deref⟩
  | ⟨arithmetic⟩
  | ⟨test⟩
  | ⟨instantiate⟩
  | ⟨refine expr⟩
  | ⟨literal⟩
  | **(** ⟨expression⟩ **)**
  | **this**
  | **null**

• *Assignment – putting one thing in another*
⟨assignment⟩ ⇒
    ⟨expression⟩⟨assign op⟩⟨expression⟩
⟨assign op⟩ ⇒
    :=
  | **+=**
  | **-=**
  | ***=**
  | **/=**
  | **%=**
  | **^=**

• *Member / data access*
⟨invocation⟩ ⇒
    ⟨expression⟩ **.** ⟨var id⟩⟨args⟩
  | ⟨var id⟩⟨args⟩
⟨field⟩ ⇒
    ⟨expression⟩ **.** ⟨var id⟩
⟨deref⟩ ⇒
    ⟨expression⟩ [⟨expression⟩]

• *Basic arithmetic can and will be done!*
⟨arithmetic⟩ ⇒
    ⟨expression⟩⟨bin op⟩⟨expression⟩
  | ⟨unary op⟩⟨expression⟩
⟨bin op⟩ ⇒
    +
  | **-**

| *****
| /
| %
| ^
⟨unary op⟩ ⇒
-

• *Common boolean predicates*
⟨test⟩ ⇒
⟨expression⟩⟨bin pred⟩⟨expression⟩
| ⟨unary pred⟩⟨expression⟩
| **refinable (** ⟨var id⟩ **)**
⟨bin pred⟩ ⇒
and
| **or**
| **xor**
| **nand**
| **nor**
| <
| <=
| =
| <>
| =/=
| >=
| >
⟨unary pred⟩ ⇒
not

• *Making something*
⟨instantiate⟩ ⇒
new ⟨type⟩⟨args⟩⟨optional refinements⟩
⟨optional refinements⟩ ⇒
ϵ
| **{** ⟨refine⟩***** **}**

• *Refinement takes a specialization and notes the required return type*
⟨refine expr⟩ ⇒
refine ⟨var id⟩⟨args⟩ **to** ⟨type⟩

• *Literally necessary*
⟨literal⟩ ⇒
⟨int lit⟩
| ⟨bool lit⟩
| ⟨float lit⟩
| ⟨string lit⟩

⟨float lit⟩ ⇒
    ⟨digit⟩+ **.** ⟨digit⟩+
⟨int lit⟩ ⇒
    ⟨digits⟩+
⟨bool lit⟩ ⇒
    true
  | **false**
⟨string lit⟩ ⇒
    “⟨string escape seq⟩**”**

• *Params and args are as expected*
⟨params⟩ ⇒
    (**)**
  | **(** ⟨paramlist⟩ **)**
⟨paramlist⟩ ⇒
    ⟨var decl⟩
  | ⟨paramlist⟩ **,** ⟨var decl⟩
⟨args⟩ ⇒
    (**)**
  | **(** ⟨arglist⟩ **)**
⟨arglist⟩ ⇒
    ⟨expression⟩
  | ⟨arglist⟩ **,** ⟨expression⟩

• *All the basic stuff we've been saving up until now*
⟨var decl⟩ ⇒
    ⟨type⟩⟨var id⟩
⟨return type⟩ ⇒
    void
  | ⟨type⟩
⟨type⟩ ⇒
    ⟨class id⟩
  | ⟨type⟩[]
⟨class id⟩ ⇒
    ⟨upper⟩⟨ualphanum⟩*****
⟨var id⟩ ⇒
    ⟨lower⟩⟨ualphanum⟩*****