PRM, The Language. Version O.2
Jean Privat

To cite this version:


HAL Id: lirmm-00102804
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00102804
Submitted on 2 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
# PRM — The Language

This document is a draft. The PRM language specification may evolve. The current prmc interpretation may differ from the present specification.

---

Jean Privat*
LIRMM† Montpellier II, France
Techical Report RR-06029

May 5, 2006 at 21h19

---

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A PRM Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>Three Simple Examples</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>The PRM Syntax: A First Impression</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Outline</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Object-Oriented Programming</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Class Definition</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>A Word about Class Specialisation</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Properties: Attributes and Methods</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Object Creation</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Visibility</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Class Specialisation</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>Genericity</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>Modules</td>
<td>32</td>
</tr>
<tr>
<td>3.1</td>
<td>Module Structure</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Module Dependence</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Class Refinement</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Procedural style</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Base Modules</td>
<td>36</td>
</tr>
</tbody>
</table>

*privat@lirmm.fr
†http://www.lirmm.fr
3.6 Base Classes ............................................ 37

4 The Base Language .............................................. 43
   4.1 Source Structure ............................................ 43
   4.2 Name .......................................................... 44
   4.3 Type .......................................................... 44
   4.4 Expression ...................................................... 45
   4.5 Statement ...................................................... 46

5 A PRM Conclusion .................................................. 52
1 A PRM Introduction

PRM stands for “Programming with Refinement and Modules”. It is an open-source programming language that has a simple straightforward style and can usually be picked up quickly, particularly by anyone who has programmed before. It is object-oriented but allows a procedural style. prmc is a PRM compiler that produces efficient machine language executables.

The PRM website: [http://www.lirmm.fr/~privat/prm](http://www.lirmm.fr/~privat/prm)

PRM is a language with a high degree of structure: it is statically typed and it allows programmers to easily produce readable source code. However it has two qualities that are mainly found in dynamically typed languages:

### Concise but Clear Syntax.

The syntax of the PRM language is clear and simple, without verbosity. A PRM program looks like a program written with a modern scripting language like Ruby or Python—in fact, the syntax mainly comes from Ruby. The PRM syntax makes difficult to have one day an *Obfuscated PRM Code Contest*.

### Small but Powerful Core.

Even if the language proposes many features, the number of different core mechanisms of the language is small. It means that the language mechanisms are clean and easy to learn. It also means that in order to provide a clear syntax with a small core, the PRM language makes a great use of syntactic sugar. Many concise pieces of syntax are strictly equivalent to a more verbose one that directly uses the core mechanism. Therefore, the semantic of these small pieces of syntax can be deduced from the semantic of the more verbose one.

1.1 Three Simple Examples

Before starting with the full description of the PRM syntax and semantic, here can be found three simple PRM programs. One can have an idea of the light syntax of the language.

The objective here is not to show the most valuable PRM characteristics but to give an idea of the syntax that will help to understand the full PRM specification.

1.1.1 Hello World

One of the simplest programs:

```prm
print("Hello ", "World.
")
```

This program simply outputs the text “Hello World.”

1.1.2 Variables and Strings

A program with two local variables:

```prm
let s: String # 's' is a local variable
s := "Hello" # Assignment with a literal strings
s.append(" World") # Concatenate two string
let a := s + "." # 'a' is another local variable
println(a) # Output 'a'
```
This program also outputs the text “Hello World.”

1.1.3 Subprograms

A program with the definition of a function and a procedure:

```plaintext
def square(i: Int): Int
# Return i^2
  do
    return i * i
  end

def main
  do
    println(square(5))
    println(square(-1))
  end
main
```

This program outputs the numbers 25 and 1.

1.2 The PRM Syntax: A First Impression

A number of points can be seen about PRM clean style and modern syntax:

- Comments are introduced by the `#` symbol and run to the end of line.
- Semicolons are superfluous—but these may optionally be placed between constructs.
- Blocks of statements start with the keyword `do` and are terminated by the keyword `end`.
- Calls use the usual notation `foo(args)`, and without arguments there is no need of parenthesis. Method invocation\(^1\) on objects uses the doted notation `x.foo(args)`.
- Procedures and functions are defined with the `def` keyword.
- Local variables are declared with the `let` keyword, and the static type may be inferred.
- Types in signatures (`def`) and in local variables definitions (`let`) use the PASCAL column notation “x: Foo”.
- Type names start with an uppercase (Int, String) while the other names—variables, functions, procedures, etc.—start with a lowercase (print, sqr, s).
- Assignments use the `:=` notation.
- Literals strings are enclosed within double quote.

\(^1\)In the current documentation, we use the terms “method invocation”. In other OO languages, one can find the same idea under the terms “method call” or “send of message”.
1.3 Outline

The present manual is divided into three parts:

- Section ?? is about object oriented programming: classes, properties, inheritance, etc.
- Section ?? is about modules management, class refinement, and the base library (base modules and base classes).
- Section ?? is about the base language: names, types, expressions, and statements.

2 Object-Oriented Programming

PRM is a pure object-oriented language. It means that:

- Each manipulated value is an object.
- Each object is an instance of a class.
- Each subprogram is a method defined in a class.
- Each method invocation corresponds to a message sending—or to a late binding i.e. depends on the dynamic type of the receiver.

However, it is possible to program software in PRM without explicitly defining classes and even subprograms: we call **procedural style** this way of programming. Procedural style is useful for simple programs or for teaching. The current section focuses on “pure” object-oriented programming. The truth about procedural style is explained in a latter section.

2.1 Class Definition

A class represents entities, the attributes of those entities and the operations that those entities can perform. Classes can represent real world entities in a model, or more artificial artefacts which occur only in computer programs. Classes are defined with the usual **class** keyword. Here is an example of a simple PRM class definition:

```prm
class Car end
```

A class represents all objects of that type. For instance in the real world we have one concept of Car, but there are many instances of Car.

2.2 A Word about Class Specialisation

Classes are elements of a **specialisation** hierarchy. A class can have superclasses (i.e. class more general) and subclasses (i.e. class more specific).

The root of the hierarchy is a class named **Any**. It means that Any is the most general class, and that each other class specialises the Any class.

---

2 PRM is used to teach algorithmic to students in first year of computer science.

3 The Any class corresponds to the Object class of the some languages like JAVA.
We talk more about specialisation, and especially inheritance, in a latter section.

2.3 Properties: Attributes and Methods

Each class has a set of properties which represents the attributes and the operations of its instances. Operations on an object typically alter its state changing the values of one or more of its attributes. In PRM, such operations are known as procedures. Computations that return a value to a query about the state of an object are known as functions. Functions and procedures are together known as methods.

2.3.1 Definition of Properties

In a class, the definition of a property starts with the def keyword followed by the name of the property.

Attributes are identified with the first letter of their name, a @ character. It can be pronounced at and stands for attribute. Attributes must have a static type.

```
class Car
  def @speed: Int
    # The current speed of the car
  def @color: String
    # The color of the car
end
```

Methods need a signature and a body. A signature is composed of some parameters (possibly none) and a return type for functions—procedures do not have return types. Bodies are blocks of statements. A function must return its result with a return statement.

```
class Foo
  def bar
    # a procedure without parameters
    do
      print("bar")
    end
  def baz: Int
    # a function without parameters
    do
      return 5
    end
  def foobar(i: Int): Int
    # a function with a parameter
    do
      return i + 1
    end
  def foobar(i: Int, j: Int): Int
```

4 The @ for attributes comes from RUBY.
# a function with two parameters
\[
\begin{align*}
    \text{do} & \quad \text{return } i + j + 1 \\
\end{align*}
\]
end

Some Remarks

- Since attributes start with an `@` character, methods and attributes can therefore share the “same name”:

```ruby
class Foo
    def @foo : Int
        def foo : Int
            do ...
            end
        end
end
```

→ ??: accessors

This characteristic is often used for attribute accessors.

- Methods can share the same name if they have a different number of parameters. Like Erlang or to a lesser extent, Smalltalk or Self.

→ ??: implicit parameter value

This characteristic is used for implicit parameter values.

- There is no static overriding: two methods with the same name and the same number of parameters cannot coexist in a same class, even if they have distinct signatures.

→ ??: covariant typing policy

This is because PRM follows a covariant typing policy.

→ ??: procedural style

- Methods can be defined outside class with the same syntax. Obviously, it is not possible with attributes.

2.3.2 Access to attributes

Objects directly access their attributes by their names:

```ruby
class Foo
    def @bar : Int
        # an attribute

        def baz
            do @bar := 5  # write access
                print(@bar)  # read access
                # output ‘5’
            end
        end
end
```

→ ??: assignment

Attribute write accesses follow the assignment rules.

→ ??: exported attribute

Except in some special cases, attributes can only be accessed by the objects that own them.

2.3 Properties: Attributes and Methods
2.3.3 Invocation of Methods

Methods are usually invoked with the common doted notation \texttt{x.foo(args)} where \texttt{x} is the receiver, \texttt{foo} the name of a method and \texttt{args} some arguments. If there is no arguments, parentheses are optional—providing superfluous ones may provoke a warning during compilation.

```ruby
class Foo
  def bar
    do
      print("bar")
    end
  end
  def baz: Int
    do
      return 5
    end
  end
  def foobar(a: Int)
    do
      print("foobar", a)
    end
  end
  def foobaz(a: Int): Int
    do
      return a + 1
    end
end

let x: Foo
...
x.bar             # Output "bar"
p(x.baz)          # Output 5
x.foobar(6)       # Output "foobar6"
p(x.foobar(7))    # Output 8
```

**self, the Current Receiver** If the receiver is the current receiver, called \texttt{self} in PRM\(^5\) it can be implicit. Therefore, \texttt{self.foo(args)} is equivalent with \texttt{foo(args)}.

```ruby
class Foo
  def bar
    do
      print("bar")
    end
  end
  def baz
    do
      bar
      self.bar
    end
end
...
let x: Foo
...
x.bar              # Output "bar"
```

\(^5\) In C++ and JAVA, the current receiver is called \texttt{this} ; in EIFFEL it called \texttt{Current}.
It is important to distinguish invocation on `self` with other invocations since visibility does not apply on invocation on `self`.

Some methods that have a special name are not invoked with the usual dotted syntax. These methods are `operators` and `assignment procedures`.

### 2.3.4 Operators

Operators are methods often used for mathematical operations. There are three kinds of operators: infix operators, prefix operators and bracket operators.

#### Infix Operator

Infix operators are: `+`, `-`, `*`, `/`, `%`, `=`, `!=`, `<`, `<=`, `>`, `>=`, `<>`, `<<`, and `>>`. They are methods with one parameter and should be defined and invoked as follows:

```plaintext
# Definition signature
class C
def -(o: T): U
    ...
end

# Invocation: x is the receiver and y the argument
let x: C
let y: T
let r: U
r := x - y
```

#### Prefix Operator

Prefix operators are: `+` and `-`. They are methods without parameters and should be defined and invoked as follows:

```plaintext
# Definition signature
class C
def -: U
    ...
end

# Invocation: x is the receiver
let x: C
let r: U
r := -x
```

#### Bracket Operator

Bracket operators are methods mainly used for indexed access (arrays for instance). They are defined and invoked as follows:

```plaintext
# Definition signature
class C
def [](o: T, p: U): V
    # Example with two parameters
    ...
end

# Invocation:
# * x is the receiver, y and z are the arguments
let c: C
```

2.3 Properties: Attributes and Methods
let y: T
let z: U
let r: V
r := x[y, z]

Two remarks about operators:

- Invocation of operators requires an explicit receiver.
- Some statements looks like operations but are not. For instance the
  Boolean’s pseudo-operators: and, or, not.

2.3.5 Assignment procedures

- ??: assignment

Their names are ended with :=. They must have at least one parameter and
no return value. They are invoked with a syntax that looks like the
assignment statement and follow its rules.

# Definition signature
class C
def foo := (o: T)
    # Example of the simple form, i.e. with one parameter
    ...
def foo := (o: T, p: U, q: V)
    # Example with three parameters
    ...
end

# Invocation
let x: C
let y: T
let z: U
let k: V

# * x is the receiver, y is the argument
x.foo := y

# * x is the receiver, y is the first argument,
#  z the second one, and k the third one
x.foo(y, z) := k

Bracket Assignment Procedure

There are also bracket assignment procedures:

# Definition signature
# * Example with three parameters
class C
def [] := (o: T, p: U, q: V)
    ...
end

# Invocation
# * x is the receiver, y is the first argument,
#  z the second one, and k the third one
let x: C
let y: T
let z: U
Let \( k : V \)
\[
x[y, z] := k
\]

Some remarks about assignment procedures:

- They also exist in Ruby in the simplest form (i.e. with exactly one parameter).

- With one parameter, they are mainly used to write attributes accessors.

- With more than one parameter, they are mainly used with indexed access when different kinds of indexes exist.

- Assignment procedure is different with the user-defined C++ assignment operator. In C++, “\( x.a = 5 \)” may correspond to the invocation of the `operator=` method on the attribute `a` of the `x` object. In PRM “\( x.a := 5 \)” corresponds to the invocation of the `a:=` method on the `x` object.

2.3.6 Implicit Parameter Value

The current prmc compiler does not yet implement this part of the specification. PRM can yield a kind of implicit argument value:

```ruby
class C
def foo(a: Int, b: Int := 5, c: Int)
do...
end...
end
```

However, implicit argument is only syntactic sugar since the last code example is strictly equivalent to:

```ruby
class C
def foo(a: Int, b: Int, c: Int)
do...
end
def foo(a: Int, c: Int)
dofoo(a, 5, c)
end
```

Multiple Implicit Parameter Values

A method can have multiple implicit parameter values:

```ruby
class C
def foo(a: Int := 5, b: Int := 6)
do...
end
```
In order to avoid ambiguities, the first parameters are less implicit than the last parameters. Therefore the two following listings are equivalent:

```ruby
class C
  def foo(a: Int, b: Int)
    do
      ...
    end
  end

  def foo(a: Int)
    do
      foo(a, 6)
    end
  end

  def foo
    do
      foo(5, 6)
    end
  end
end
```

Comparison with Other Languages

Implicit parameter values exist in many other languages like C++, JAVA 5.0 or RUBY. However, their semantic differ in the way that there is one method defined, and the implicit parameter values are integrated to the arguments when the method is invoked.

For instance, let “void foo(int a, int b = 5)” be a C++ method. The two expressions foo(1, 5) and foo(1) invoke this method with the same arguments, since 5 is implicitly added in the last expression.

With PRM and the equivalent foo method “def foo(a: Int, b: Int := 5)”, the two expressions foo(1, 5) and foo(1) invoke two distinct methods—respectively, foo with two parameters and foo with one parameter.

The PRM way offers two advantages:

- Implicit parameter values are only sugar syntax: it does not extend the PRM core mechanism.
- Each method is independent and can be independently redefined. Thus more flexibility is offered to the programmer.

2.3.7 Variable Argument Number

This part of the specification is not considered as stable and may change in future version.

In PRM, some methods can be invoked with an unbounded number of arguments.

Definition

In those method definitions, one special parameter is declared as t: T* where t is the name of the parameter and T the type of arguments. In such definitions, the static type of t is Array[T]. Example:

```ruby
class C
  def foo(a: Int, b: Int*, c: Int)
end
```
The static type of 'b' is 'Array[Int]' do
    print(a, "-", b.length, "-", c)
end

Invocation

In method invocation, the special parameter is associated with one or more arguments:

```kotlin
let c: C
c.foo(1, 2, 3)  # 'Output 1-1-3'
c.foo(1, 2, 3, 4, 5, 6) # 'Output 1-4-6'
c.foo(1, 2)  # compilation error, unknown foo method # with two parameters
```

→ ??: print

Without surprise, the standard `print` method, used in all those examples, accepts multiple arguments. Its signature is `print(a: Any*)`.

Passing Array

One can pass an array object instead of a list of elements with the `*a` notation—here, `*` is not an operator, it is just a notation:

```kotlin
let a: Array[Int]
let c: C...
c.foo(1, *a, 3)
```

Passing array is often used to chain calls:

```kotlin
class C
    def printprint(a: Any*)
        # Remember, the static type of 'a' is 'Array[Any]' do
        print(*a)
        print(*a)
end
let c: C
c.printprint("Hello") # Output 'HelloHello'
```

<→ In a class, cannot coexist:

- Two methods with the same name that both accept a variable number of arguments:

```kotlin
class Foo # ERROR !
    def bar(a: Int, b: Int*, c: Int)
        do ... end
    def bar(d: Int*)
        do ... end
end
let f: Foo
f.bar(1, 2, 3) # Ambiguous
```

- Two methods with the same name, one accepts a variable number of arguments, and the other has more parameters than the minimal number of the first:
class Foo
    def bar(a: Int, b: Int*)
        do ... end
    end
    def bar(c: Int, d: Int)
        do ... end
end
let f: Foo
f.bar(1, 2) # Ambiguous

However, the following listing is OK:

class Foo # ERROR /
    def bar(a: Int, b: Int*)
        # bar1
        do ... end
    end
    def bar(c: Int)
        # bar2
        do ... end
end
let f: Foo
f.bar # Not ambiguous, it is an error
f.bar(1) # Not ambiguous, it is bar2
f.bar(1, 2) # Not ambiguous, it is bar1
f.bar(1, 2, 3) # Not ambiguous, it is bar1

2.4 Object Creation

new Objects are created with the special new statement:

new Car
new Car("red")
new Car.with_color("blue")

The point to note is constructors need to be declared in classes in order to allow them to be instantiated. In PRM, constructors are a little different from those of languages like C++ and Java; EIFFEL constructors are the closest.

Constructor

Constructors are procedures defined in a class after the constructor keyword. More than one procedure can be defined as constructors.

In the following listing, the two init procedures and the with_color one are constructors, but paint is a “normal” procedure—note that if a lot of code is duplicated it is only for the need of the illustration:

class Car
    def @color: String
        # The color of the car
    end
    def paint(c: String)
        # Repaint the car
        do
            @color := col
        end

end

constructor
def init
do
  @color := "black" # Mr. Ford?
end

def init(col: String)
do
  @color := col
end

def with_color(col: String)
do
  @color := col
end

<➔ The PRM naming convention is to reserve the methods named init or with_something to be constructor procedures.

➔ ??: Visibility

It important to notice that constructors look like “normal” procedures. The visibility section will show the true about the constructor keyword and status of constructors.

Implicit Constructor

Implicitly, the constructor named init is called on object instantiation. Thus new Car is equivalent to new Car.init, and new Car("Blue") is equivalent to new Car.init("Blue")

<➔ Since both object creation and method invocation use a dot in their notation, some cases should be disambiguated:

  • new Foo.bar is always considered as the instantiation of a Foo object with a constructor named bar.

  • (new Foo).bar and new Foo.init.bar are the instantiation of a Foo object with a constructor named init; and followed by the invocation of a property named bar on this newly created object.

  • new Foo(5).bar and new Foo.init(5).bar are the instantiation of a Foo object with a constructor named init with 5 as argument; and followed by the invocation of a property named bar on this newly created object.

2.4.1 Abstract Classes

Abstract classes are classes that can not be instantiated. Classes that are not abstract are called concrete classes.

<➔ In PRM, Abstract classes are simply classes without constructors. In corollary, classes without constructor are abstract, therefore can not be instantiated.

Empty Constructor

In comparison with other languages, there are no default constructors since their use is marginal, even if they are the cause of many errors.
However, sometimes, programmers need to define concrete classes with empty constructors. In PRM they just have to explicitly do it:

```plaintext
class Foo
constructor
  def init
do
  end
end
```

### 2.4.2 Garbage Collector

PRM has no delete operator. This is because, as many other modern languages, PRM is garbage collected. Garbage collection is known to completely cure the programming ills of dangling pointers and memory leaks. This greatly simplifies the programming effort by removing one of the largest bookkeeping headaches for programmers. Garbage collection has also proved to be very efficient in modern implementations.

### 2.5 Visibility

C++ and Java programmers might be wondering how to make methods public, protected and private. With PRM you have far more control: as in Eiffel, any set of methods can be exported to all, to none or to some specific classes. Thus you have the possibility of many shades of grey between public and private. You might want a method to be public to some specific classes, but private to others. Moreover, method visibility and constructors are related together in a nice original way.

 Visibility is not related to method invocation on `self`. Therefore, properties are always accessible to the current receiver.

#### 2.5.1 Method Visibility Blocks

Visibility is controlled by three keywords that delimit visibility blocks: `public`, `private`, and the already known `constructor`. Method defined after such a keyword belong the corresponding visibility block.

A class definition can contain any number of blocks, in any order:

```plaintext
class Foo
  ...
public  ...
public  ...
public  ...
private ...
public ...
constructor ...
private ...
end
```
It is recommended to regroup related methods with the same visibility in the same block. And it is also recommended to put two unrelated sets of methods in two different visibility blocks, even if they share the same visibility.

### 2.5.2 Public Method Visibility

Methods defined in a `public` block are exported and can be used by other classes. If the name of a class is added after the `public` keyword, methods are only exported to this class and to its subclasses. If there is not such a class name, method are exported to any classes—in fact, they are exported to the `Any` class and to its subclasses.

Example:

```ruby
class Car
  ...
  public
    def speed : Int
      # Get the speed of the car
      do
        return @speed
      end
  end
  public Driver
    def stop
      # Stop the car
      do
        @speed := 0
      end
end
```

Let `c` be a variable statically typed by a `Car`. Here the function `speed` is exported to any class, therefore `c.speed` is valid in any class. The procedure `stop` is exported to the class `Driver` (and all its subclasses), therefore `c.stop` is only valid in the class `Driver` and in any subclasses of `Driver`.

Implicit Visibility Block

The *implicit visibility block* (i.e., the visibility block above the first visibility keyword) is a `public` one. For example, the three following listings are equivalent:

```ruby
class Foo
  public Any
    def bar
      do
        print("baz")
      end
  end
end
```

```ruby
class Foo
  public
    def bar
      do
        print("baz")
      end
end
```

### 2.5 Visibility
Method defined in a private block are not exported. Therefore, private methods are only accessible to the current receiver.

Let c be a variable statically typed by a Car. The procedure `stop_car` is exported to nobody, therefore `c.stop_car` is valid nowhere. The only way to invoke such a method is to use the current receiver.

In PRM, private methods are usable only by self—it is an instance visibility. In C++, private methods are usable only by instances of the current classes—it is a class visibility.

The following listing will try to illustrate the difference:
2.5.4 Constructor Method Visibility

Methods defined in a constructor block are usable as a constructor method. As with the public keyword, constructor can be used to control the visibility of constructors: if the name of a class is added after the constructor keyword, methods are only exported as constructor to this class and to its subclasses. If there is not such a name, the Any class is considered.

```ruby
class Car
  ...
  constructor CarFactory
    def init
      do
        @color := "black"
        @speed := 0
      end
  ...
end
```

The statement `new Car` is only valid in the class `CarFactory` and in its subclasses.

Arguments: object creation

Even if some procedures have a status of constructor they can be invoked on the current receiver—constructor status is only a matter of visibility, and is not related with invocation on self. Such invocations allow a better factorisation:

```ruby
class Car
  def @color: String
    constructor
      def init
        do
          with_color("black")
        end
      def with_color(col: String)
        do
          @color := col
        end
    end
end
```

Arguments: object creation

Methods can not be both public and constructor since from a user point of view, object creation and send of message correspond to two different needs. Allowing exporting a procedure public and constructor will be a reusability limitation because of class refinement. However, code duplication should be avoided:

```ruby
class Car
  def @color: String
end
```
2.5.5 Attribute Accessor

As in Smalltalk, attributes are “private”: they can only be accessed by the objects that own them. Therefore, some methods should be defined in order to access attributes. Methods that play this role are called accessors.

Usually, there is the need of two accessors, one for the read access and one for the write access. In PRM, you can use the same name for the attribute and for the two accessors: the attribute is distinguished with the @ and the write accessor is usually an assignment procedure, therefore distinguished with the :=.

Example:

```ruby
class Car
  def @speed: Int  # Attribute
    def speed: Int  # Read accessor
      do
        return @speed
      end
    end
    def speed:= (s: Int)  # Write accessor
      do
        @speed := s
      end
    end
  end
end
```

In this example, let c be a Car. c.speed returns the value of the attribute @speed and c.speed := 5 assigns 5 to the attribute @speed:

```ruby
let c := new Car
c.speed := 5
print(c.speed)  # Output "5"
c.speed := 10
print(c.speed)  # Output "10"
```

Automatic Accessor The keywords def_read and def_write can be used to simplify the declaration of such accessors. On attribute definition, def_read
automatically generates a read accessor and def_write automatically generate a write accessor.

The following example is equivalent to the previous one.

```ruby
class Car
  def @speed: Int def_read def_write
  ...
end
```

Automatic Accessor Visibility

Since def_read and def_write only correspond to syntax sugar, the visibility of automatic accessors is the one of the current visibility block.

```ruby
class Car
  public
  def public_price : Int
do
    return @cost + @margin
end

public CarSeller
  def @cost : Int def_read
def @margin : Int def_read def_write
end
```

In this example, only a car seller can access the real price of a car.

Pseudo-accessor

Accessors are just a role playing by some methods. It is possible to define “pseudo-accessors”, i.e. methods that act like accessors from the user point of view. The following example defines two pairs of accessors on the speed attribute of a Car class but with different speed units, one in kilometre per hour and the other in miles per hour:

```ruby
Class Car
  def @speed_kmph: Int def_read def_write
  # Speed in kmph
  def speed_mph: Int
  # Speed in mph
  do
    return @speed_kmph * 63 / 100
  end
  def speed_mph:=(s: Int)
  # Speed in mph
  do
    @speed_kmph := s * 100 / 63
  End
  ...
end
```

Thus, from a user point of view, it is not possible to distinguish the “true” accessor from the pseudo-accessor:

```ruby
let c := new Car
c.speed_kmph := 80
print(c.speed_kmph, " ", s.speed_mph)
```

2.5 Visibility
2.5.6 Exported Attribute

The specification is not considered as stable and may change. Moreover, the current prmc compiler do not yet implement it.

→ ??: attribute access
Software engineering considers that attribute should be accessible only for the current receiver (self). However, in some exceptional case, attributes need to be directly accessed by different objects.

export
The keyword export permits to change the visibility of an attribute. The visibility granted is the one of the current visibility block.

In the following example, the attribute @baz is visible in the class Bar and in all its subclasses:

```plaintext
class Foo
  public Bar
    def @baz: Int
      export @baz
  end
end
```

Exported Attribute Access
Exported attributes are accessed with the doted notation x.@baz where x is the receiver (i.e. the instance that owns the attribute) and @baz the name of the attribute.

→ ??: assignment
Exported attributes can be used as an expression or as the left part of an assignment:

```plaintext
f.@baz := f.@baz + 1
```

2.6 Class Specialisation

Specialisation has three main uses:

→ ??: property inheritance
• Build new classes out of existing classes since classes inherit properties defined their superclasses.

→ ??: visibility
• Gain property visibility since properties exported to a class (public and constructor) are visible to their subclasses.

→ ??: type
• Permit subtyping since objects of a class can be used where objects of the superclasses are expected.

In many object-oriented languages, inheritance is mainly a way to reuse property already defined. The semantic of inheritance of the PRM language is a bit different since it strictly corresponds to the natural semantic of specialisation: If A is a superclass of B then each instance of B is also an instance of A. The three uses of specialisation are simply corollaries of this strict semantic. It also means that two uses of specialisation, frequent in some OO languages, are forbidden in PRM: inheritance of implementation and repeated inheritance.
inherit The inherit keyword is used to declare the superclass of the class. This keyword must be used before any property declarations.

The following listing is a very simple example of inheritance where a Car class is a subclass of a Vehicle class:

```ruby
class Vehicle
end

class Car
  inherit Vehicle
end
```

Multiple Class Specialisation With multiple specialisation, the inherit keyword is repeated:

```ruby
class Drake
  inherit Duck
  inherit Male
end
```

Transitive Specialisation In PRM, transitive specialisation relation links are ignored. Therefore, the two following listings are equivalent:

```ruby
class Ambulance
  inherit Car
end

class Ambulance
  inherit Car
  inherit Vehicle
end
```

Moreover, the last one may produce a warning during compilation because of the superfluous inherit Vehicle.

2.6.1 Property Inheritance and Redefinition

Properties Inheritance Subclasses inherit the properties—attributes and methods—of their superclasses.

```ruby
class Car
  def @color: String def_read
    def sound: String
      do
        return "vroom"
      end
    end
end

class Convertible
  inherit Car
  def @roof_is_open: Boolean
end
```

This example shows a superclass Car and a subclass Convertible. The Convertible class inherits the following properties: the attribute @color,
accessors

the automatic color accessor, and the sound function. It also defines a new property, the attribute @roof_is_open.

Properties Redefinition

Subclasses can redefine some inherited properties by providing a new definition of a property.

The following example shows the redefinition of the sound function inherited from the Car class:

```eiffel
class Ambulance
  inherit Car
  def sound : String
    do
      return "wo-wo"
    end
end
```

Precursor

With the Eiffel terminology, we say that the sound method of the Car class is a precursor of the sound method of the Ambulance class.

Global Property

In the previous example, the sound method of the Car class and the sound of the Ambulance class are two different methods. However, they belong to a “same property idea”, here the idea is something like “sound of cars”. We call global property this “same property idea”.

Global properties are introduced when its first property is defined. Example, the global property “sound of cars” is introduced in the Car class by the sound method.

In PRM, global properties are not strictly related to properties names. For example, in the following listing, the two properties @height belong to distinct global properties:

```eiffel
class Person
  def @height : Int # in cm
  def @weight : Int # in kg
end

class Button
  # A button for a graphical user interface
  def @height : Int # in pixels
  def @width : Int # in pixels
end
```

This notion of global property is one of the PRM exclusivity. In great majority of other OO languages, the absence of this notion yields quantities of problems.

Attribute Redefinition

Obviously, redefinition is majorly used for methods. It is also possible to redefine an attribute by specialising its static type:

```eiffel
class Car
  def @driver : Person
end

class PoliceCar
  def @driver : Policeman
end
```
This is because has a covariant typing policy.

2.6.2 Deferred Method

Deferred methods (called pure virtual method in C++) are methods without implementation. A deferred method is declared without a body, instead it has the as deferred keywords.

```plaintext
class Car
  def has_priority: Bool as deferred
end

class Ambulance inherit Car
  def has_priority: Bool
    do
      return false
    end
end
```

Remarks:

- Usually, classes that contain deferred methods are mainly abstract classes—i.e. do not have constructors.
- Concrete classes with deferred methods can be useful with refinement.

2.6.3 Multiple Inheritance

When a class has only one super-class, inheritance and redefinition are quite intuitive mechanisms. PRM multiple inheritance mechanism is also intuitive.

Which Properties to Inherit? The inherited properties are the most specific ones—i.e. the properties defined in the most specific classes. This base behaviour is quite simple but slightly differs from the majority of OO languages.

Example:

```plaintext
class A
  def foo
    do
      print("fooA")
    end
end

class B inherit A
end

class C inherit A
  def foo
    do
      print("fooC")
    end
end
```
In the BC class, there are two potential inherited methods: `foo` defined in the A class, and `fooC` defined in the C class; the second is the most specific because C specialise A; therefore the BC class inherit the "foo" method defined in C.

**Multiple Precursors**

A property can redefine more than one property inherited from super-classes:

```ruby
class D
  inherit A
  def foo
    do
      print("fooD")
    end
  end
end

class CD
  inherit C
  inherit D
  def foo
    do
      print("fooCD2")
    end
  end
end
```

Here, the `foo` method of the CD class has two precursors since it redefines the `foo` methods of the classes C and D.

**Property Conflict**

A property conflict occurs when the most specific property to inherit is not unique:

```ruby
class CD2
  inherit C
  inherit D
end
```

The solution to avoid them is to redefine the conflicting property.

→ ??: deferred method

When all properties but one are deferred, the conflict is automatically resolved: the one that is not deferred is inherited.

**Global Property Conflict**

A global property conflict occurs when a class inherits homonym properties that belong to distinct global properties:

```ruby
class O
  def foo
    do
      print("fooO")
    end
end
```
The solution to avoid them is to rename at least one of the two conflicting method with the rename keyword:

```plaintext
class AO
inherit A
inherit O
end
```

```plaintext
class AO2
inherit A rename foo(0) as fooA
inherit O
constructor
def init do end
end
```

```plaintext
let x := new AO2
x.foo # Output 'fooO'
x.fooA # Output 'fooA'
```

When renaming methods, the first name has to precise between parentheses the number of parameters.

One can rename more than one property

```plaintext
class Y
  inherit X rename foo(0) as fooX,
    @bar as @barX, -(1) as minus
end
```

The PRM renaming differs from the EIFFEL one and is slightly simpler and more coherent:

- One renaming per global property is enough, even if the global property comes from many super-classes:

```plaintext
class AC2 # WARNING: Superfluous renaming.
inherit A rename foo as foo2
inherit C rename foo as foo2
end
```

- A global property can not have two names in a same class:

```plaintext
class AC2 # ERROR: Multiple renaming.
inherit A rename foo as fooA
inherit C rename foo as fooC
end
```

- Two distinct global properties cannot be renamed to have the same name:

```plaintext
class P
def bar
do
  print("barP")
end
```
2.6.4 Visibility

Visibility Inheritance

The visibility `public` and `private` of method inherited. However, constructors are inherited as private methods. This is because constructors of a class are not adapted to its subclasses.

In the following listing, `Ambulance`, a subclass of a `Car` class inherit the `with_color` as a private method. Therefore, `Ambulance` defines `init`, a new specific constructor.

```plaintext
class Car
  def @color: String def_read
  constructor
    def with_color(col: String)
    do
      @color := col
    end
  end
end

class Ambulance
  inherit Car
  constructor
    def init
    do
      with_color("white")
      # OK, since with_color is inherited
    end
  end
end
```

Here some uses of the two classes:

```plaintext
let c := new Car.with_color("black")
print(c.color) # Output 'black'
let a1 := new Ambulance
print(a1.color) # Output 'white'
let a2 := new Ambulance.with_color("blue") # Error!
# -> 'with_color' is not a constructor,
# it is a private method
```

Visibility Redefinition

One can redefine the visibility of inherited method by redefining the method in the wanted visibility block. The visibility of inherited method can also be redefined without having to redefine the whole method.

```plaintext
export
The `export` keyword enables to change the visibility of inherited method to the one of the current block:

```plaintext
class Car
  constructor
    def init
```
do  
@speed := 0
end

class Ambulance
inherit Car
constructor
    export init(0)
end
let c := new Ambulance # OK

As for renaming, the number of parameters has to be indicated between parentheses.
Multiples methods can be exported at the same time:

class Bar
inherit Foo
public Baz
    export foo(1), bar(0), baz=(1), +(1)
end

2.6.5 Call to Super

super In a method redefinition, the programmer can refer to the previous property with the super keyword:

class Foo
def foo(i: Int): Int
    do
        return i + 1
    end
end
class Bar
inherit Foo
def foo(i: Int): Int
    do
        return super(i*2) * 2
    end
constructor
def init do end
end
let b := new Bar
print(b.foo(2)) # Output '10'

Implicit Super Arguments Arguments of a super call are implicitly the parameters of the method. Therefore, the two following listings are equivalent:

...  
def foo(a: Int, b: String)
do...

2.6 Class Specialisation
Multiple Precursor

When a method has more than one precursor, any call to super must be prefixed with a class name in order to remove the ambiguity. Such prefixes use the :: notation:

```prm
class CD3
  inherit C
  inherit D
  def foo(a: Int)
    # This method redefines the ones
    # of the classes C and D
    do
      C::super(a+2)
      D::super(a-1)
    end
  end
end
```

2.7 Genericity

Inheritance is one of the fundamental mechanisms for reuse; so is genericity. Genericity is also important in making programs type safe without resorting to type casts. Java 5.0 introduces genericity, in previous version, many type casts where needed to make up for this deficiency. C++ has genericity in the form of template classes. If you have had problems understanding C++ templates, don’t worry, PRM’s generic syntax is much easier, and more powerful, as it also allows generic parameters to be bounded; this is known as bounded genericity.

In PRM, genericity is mainly the one of the EIFFEL language, please refers its specification to know more about genericity.

Generic Class Definition

In order to use genericity, you create a generic class with formal generic parameters. In the following listing, Pair is a generic class with one formal parameter bounded by Any and VehiclePark is a generic class with one formal parameter bounded by Vehicle.

```prm
class Pair[E: Any]
  end
class Vehicle
  end
class VehiclePark[E: Vehicle]
  end
```

2  Object-Oriented Programming
In program, the generic class can be used to construct many kinds of *generic types*:

```scala
definition class Pair[E: Any]
  def @first: E def_read def_write
  def @second: E def_read def_write

  def switch
  do
    let t: E
    t := @first
    @first := @second
    @second := t
  end

  def display
  do
    print(@first, " ", @second)
  end

  constructor
  def init(f: E, s: E)
  do
    @first := f
    @second := s
  end
end
```

Here some examples of use:

```scala
let pi := new Pair[Int].init(5, 4)
pi.display  # Output '5 4'
pi.switch
pi.display  # Output '4 5'

let ps := new Pair[String]("Hello", "Town")
ps.second := "World"
ps.display  # Output 'Hello World'
```

Genericity yields a kind of subtyping. For example, `VehiclePark[Car]` is a subtype of `VehiclePark[Vehicle]`:

```scala
let vehiclepark: VehiclePark[Vehicle]
let carpark: VehiclePark[Car]
```
vehicle := carpark # OK

This is because PRM follows a covariant typing policy.

### 3 Modules

In the PRM language, classes are organised into modules. A modules corresponds to a source file and contains class definitions. They are also the compilation units: each module can be separately compiled and then linked to produce an executable.

#### 3.1 Module Structure

A PRM source file (a module) is divided into four parts:

1. Importation of modules.

2. Definition of classes.

3. Definition outside classes of procedures and functions.

4. Definition of the module main statements.

Each part is optional but the order must be respected.

#### 3.2 Module Dependence

Modules can depend on others modules and import their classes. With analogy with the class terminology, we call supermodules of a module m the modules it depends on, and submodules[6] the modules that depend on m.

**Import**

The dependence between modules is declared with the import keyword followed by the name of the module:

```prm
import crypt
import http
```

As in class hierarchy, cycles are forbidden in the module dependency! A module m cannot require itself nor require a module that requires m.

**Implicit Dependence**

Implicitly, modules depend on the module standard that contains standard classes.

**Class Conflict**

There is a class conflict when a module imports two homonymous distinct classes.

Such a conflict can be resolved with the rename keyword:

```prm
import automobile # import Car
import tramway rename Car as Tramcar
```

**Property Renaming**

Class renaming follows and property renaming follows the same rules.

---

[6] In some language, “submodules” refers to nested modules (i.e. modules defined into a modules). Since there is no module nesting in PRM, there are no ambiguities.
3.3 Class Refinement

PRM modules can extend imported classes, this is called *class refinement*:

```plaintext
# File m1.prm
class Foo
  def bar
do
    print("before")
  end
end

def init do end
end

# File m2.prm
import m1
class Foo
  def bar
do
    print("after")
  end
end

(new Foo).bar # Output 'after'
```

Properties: Definition and Redefinition

The main usage of refinement is to add new properties (methods and attributes) or to redefine them.

It’s important to note that refinement is not specialisation: if you specialise a class, you have two classes, if you refine a class, you still have one class.

Class refinement is one of the greatest PRM features. It improves the reusability of OO software since it provides an answer to the separation of concern problem: a module can adapt existing classes to new concerns. Class refinement is clearly not a new OO feature and exists in many dynamically typed languages (like RUBY or LISP) and in some statically typed language (like OBJECTIVE-C).

Multiple Refinement

Refinement can be also combined without difficulties:

```plaintext
# File m3.prm
import m1
class Baz
  def @foo: Foo
    def bar
do
      @foo.bar
    end
end

def init do end
end

(new Foo).bar # Output 'after'
```
Class refinement works for any classes; even with build-in ones:

```ruby
# File m4.prm
import m2
import m3
(new Baz).bar  # Output 'after'
```

```ruby
class Int
def fib : Int
  # Fibonacci numbers
  # The inefficient recursive algorithm
  do
    if self <= 0 then
      return 0
    elsif self <= 2 then
      return 1
    else
      return (self - 1).fib + (self - 2).fib
    end
  end
end

print(6.fib) # Output '8'
```

Addition of Superclasses

It is also possible to refine a class by giving it new superclasses. This kind of refinement is quite rare, even in dynamically typed languages:

```ruby
class Foo
def foo
do
  print(self, "-foo")
end
end
class String
inherit Foo
end

"Hello".foo  # OK since String inherit the foo method
  # Output 'Hello-foo'
let f: Foo
f := "World"  # OK since Strings are now Foos
f.foo         # Output 'World-foo'
```

Refinement, Specialisation and Multiple Inheritance

For property inheritance, refinement behaves like specialisation.

### 3.4 Procedural style

From a programmer point of view, PRM can be used as a procedural language:

- Some methods, like `print` seems to not have receiver.
- Procedures and functions can be defined outside classes.
The main statements can be written outside procedures and functions. However, PRM is a pure object-oriented language: procedures and functions are always methods, and statements always belong to method bodies.

### 3.4.1 Method Without Receiver

→ ??: self

Since each method invocation has a receiver, it means that each `print("Hello World")` use `self` as receiver.

In fact, `print` is a private method defined in `Any`, thus inherited as a private method in any other classes:

```
print("Hello ")  # Output "Hello \\
self.print("World")  # Output "World"
5.print(".")        # Error
# -> 'print' is a private method
```

### 3.4.2 Method Definition outside Classes

Procedures and functions defined outside classes are implicitly defined as private methods of the class `Any`. Therefore, as `print`, they can be used everywhere.

\( \rightarrow \) Methods defined outside classes correspond to an implicit refinement of the class `Any`. Example: the two following listings are equivalent:

```ruby
def foo
do
    print("hello world")
end

class Any
private
def foo
do
    print("hello world")
end
end
```

### 3.4.3 Module Main Statement

The module main statements belong to the body of an implicit private `main` procedure defined in an implicit `Sys` class.

→ ??: Sys class

The `main` method of the `Sys` class corresponds to the entry point of programs.

\( \leftarrow \) Main statements correspond to an implicit refinement of the class `Sys`. The two following listings are equivalent:

```ruby
print("Hello world")
```

is syntactically equivalent to

### 3.4 Procedural style

35
3.5 Base Modules

The PRM standard library is made of 6 modules:

→ ??: base classes

standard The standard module is the implicitly required by other modules. It contains all the necessary base classes.

net implicitly depends on standard.

It specialises some IO classes in order to provide networking socket communication.

http depends on net.

It defines classes related to basic HTTP communication.

exec implicitly depends on standard.

It defines classes used to execute arbitrary commands of the shell system.

sdl implicitly depends on standard.

It defines wrapper classes around the Simple DirectMedia Layer C library

http://www.libsdl.org/

It is primarily defined to experiment the feasibility of wrapping C libraries.

opts implicitly depends on standard.

It defines classes related to the parse and the analysis of command line options.

This base module is the only one developed by someone else: Floréal Morandat.

3.5.1 Standard Module

In fact, the standard module is not the root of the module hierarchy. standard is an empty module that only require base classes from some “super-standard” modules.

This section is mainly an illustration to how module hierarchy and class refinement can be used to develop modular applications—i.e. where modules are clearly “concern” units.

The PRM standard module depends on 11 super-modules:
kernel depends on nothing.
It is the root of the module hierarchy and minimally defines the most basic classes like Any, Sys, Int, Float, Bool or Char.

math depends on kernel.
It refines the Int and Float classes with useful mathematical methods; for instance trigonometry. In a near future it will also define some class like complexes or big numbers.

abstract_collection depends on kernel.
It defines the PRM collection abstract generic class hierarchy. From the general Collection[E: Any] and Iterator[E: Any] classes to more specific like abstract set or abstract maps (i.e. associative arrays).

range depends on abstract_collection.
It defines the Range class and related ones.

array depends on abstract_collection.
It defines the Array class and related ones. It also specialises many abstract collection classes into concrete ones implemented with arrays—ArrayMap, ArraySet, etc.

sorter depends on array.
It refines the classes of the array module by the addition of sort method.

list depends on abstract_collection.
It specialises some collection classes—i.e. like the array module but with linked lists.

string depends on array.
It defines the String class and related ones. It also refines many classes by adding a to_s method used to convert any objects to a human readable representation.

hash depends on string.
It refines classes with a hash function and implements some hashes collections—HashMap, HashSet, etc.

io depends on string.
It define input/output related classes likes File. It also refines the Any by adding the print method.

string_search depends on string.
This module is about string searching and matching. It also implements the Boyer-Moore fast string searching algorithm.

3.6 Base Classes

Kinds of Classes Classes can be classified into 4 categories:
Primitive Classes: They correspond to the primitive values of the computer. They are mainly defined in the kernel module. Primitive classes and their super-classes can have some restrictions.

Built-in Classes: They are known by the compiler since they have literals representation or are needed in some particular statements or expressions. Obviously, they include primitive classes.

Base Classes: They are the classes defined or imported by the standard module. They include Built-in classes.

User Classes: They are the other classes and are defined by PRM programmers.

Literal Value Some built-in classes have literate value: programmers can create objects without explicitly instantiate them. The table ?? summarises build-in classes and gives example of literals values.

<table>
<thead>
<tr>
<th>examples</th>
<th>PRM types</th>
</tr>
</thead>
<tbody>
<tr>
<td>integers</td>
<td>Int</td>
</tr>
<tr>
<td>floats</td>
<td>Float</td>
</tr>
<tr>
<td>characters</td>
<td>Char</td>
</tr>
<tr>
<td>strings</td>
<td>String</td>
</tr>
<tr>
<td>Booleans</td>
<td>Bool</td>
</tr>
<tr>
<td>range</td>
<td>Range[Int], Range[Char]</td>
</tr>
<tr>
<td>arrays</td>
<td>Array[Int], Array[Char]</td>
</tr>
<tr>
<td>void</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 1: The Basic Types

3.6.1 Any

The Any built-in class is the root of the class hierarchy.

Here some notable properties that will be inherited or redefined in other classes:

```python
public # Equality tests
def ==(a: Any): Bool
    # The identity equality
    # Return 'true' if 'self' and 'a'
    # are the same object
    # False otherwise
    # /!\ This method cannot be redefined

def !=(a: Any): Bool
    # Return 'not self == a'
    # /!\ This method cannot be redefined

def ==(a: Any): Bool
    # The value identity
    # Return 'true' if 'self' and 'a'
    # have the same 'contents'

def !=(a: Any): Bool
    # Return 'not self = a'
```

38 3 Modules
public # String
    def to_s: String
        # Convert 'self' to a human readable form
private # Basic IO
    def print(a: Any*)
        # For each argument, output the
        # human readable form ('to_s')
end

3.6.2 Int

The Int primitive class represents internal machine integers. Literals are sequence of digits.

→ ??: operator

Notable Int properties are their operators. Many of them cannot be redefined.

3.6.3 Float

The primitive Float class represents internal machine float numbers.

Literals are sequence of digits a dot and another sequence of digit.

3.6.4 Character

The Char primitive class represents characters.

Literals are delimited with single quotes. Characters can include escaped sequence—Table ??.

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>ASCII Linefeed (LF)</td>
</tr>
<tr>
<td>\r</td>
<td>ASCII Carriage Return (CR)</td>
</tr>
<tr>
<td>\t</td>
<td>ASCII Horizontal Tab (TAB)</td>
</tr>
<tr>
<td>\0</td>
<td>ASCII Nul (NUL)</td>
</tr>
<tr>
<td>\</td>
<td>Backslash ()</td>
</tr>
</tbody>
</table>

in strings only...

| \" | Double quote (") |
| \\ | Hash (#)         |

Table 2: Char and String Escape Sequences

3.6.5 String of Characters

The String built-in class represents strings of characters—i.e. pieces of text.

Literals are delimited with double quotes. As with character, some escape sequence can be used to represent some characters—Table ??.

Extended String Literal String literal can contain embedded expression using #{]}

The #{] notation comes from Ruby
Programmers are encouraged to use extended strings because they are better for internationalisation since only entire sentences should be presented to the translator. Therefore having one string `"Replace #o with #o?"` is obviously better than having the three strings `"Replace ", " with ", and "?"`.

### 3.6.6 Boolean

The `Bool` primitive class represents the two Booleans `true` and `false`. Pseudo-operators `and`, `or` and `not` There is also three special operators on Booleans that are not operators of the `Bool` class: `and`, `or`, and `not`. These keyword have a special status since `and`, `or` are lazy and will return the value, as soon as falsity (`and`) or truth (`or`) is established.

Booleans are used in many places, especially:

- Conditional statement.
- While loop.
- Check statement.

### 3.6.7 Collection

The `Collection` built-in generic abstract class is the root of the collection class hierarchy.

Here some notable deferred method:

```plaintext
class Collection[E: Any]
def is_empty: Bool
    # Is there no item in the collection ?

def length: Int
    # Number of items in the collection.

def has(item: E): Bool
    # Is ‘item’ in the collection ?
    # Comparisons are done with =

def iterator: Iterator[E]
    # Get a new iterator on the collection.
end
```
3.6.8 Iterator

The Iterator built-in generic abstract class is mainly used with collections. Instances of the Iterator class generates a series of elements, one at a time. Here some notable deferred method:

```plaintext
class Iterator[E: Any]
def item: E
    # The current item.
def next
    # Jump to the next item.
def is_ok: Bool
    # Is there a current item?
end
```

3.6.9 Array

→ ??: genericity

The Array built-in generic class is a subclass of Collection. It is also the preferred representation form of collections of items.

Array Literal

Literals representation use brackets and elements are separated with comma. Example:

```plaintext
let ai := [5, 4, 6, 1]
print(ai.length)  # Output '4'
print(ai.has(5))  # Output 'true'
print(ai.has(9))  # Output 'false'
print(ai)         # Output '5461'

let ai := ["Hello", "", "World"]
print(ai.length)  # Output '3'
print(ai)         # Output '"Hello World"'
```

Static Type of Array Literal

Since Array is generic, the type of literal should be computed. Literals expressions are valid if and only if there is a unique more general static type. The static type of the literal expression is build with this type:

```plaintext
let ai := [4, 5]    # more general type: Int
    # Therefore ai is an Array[Int]
let ae := [4, '5']   # more general types: Int and Char
    # Therefore, it is an error
let x: Any
let aa := [4, '5', x] # more general type: Any
    # Therefore ai is an Array[Any]
```

Array Constructor

When literals are invalid, one can use the init array constructor therefore explicitly precise the desired static type:

```plaintext
let ae := new Array[Any](4, '5') # OK
```

→ ??: multiple arguments

Arrays are used in multiple argument procedures. The Array init constructor uses multiple arguments.
3.6.10 Range

The built-in a generic class Range is for Discrete elements. It represents intervals between a first element and a last element.

There are two kinds of ranges, inclusive ranges and exclusive ranges.

Inclusive Range

They include the first element, the last element and each element between them. Their literals use the \([a..b]\) notation \((a \text{ and } b \text{ can be any expression})\):

Literals are valid if first element and the last element have the same static types.

```
let x := [1..5] # x is Range[Int]
print(x.length) # Output '5'
print(x.has(0)) # Output 'false'
print(x.has(1)) # Output 'true'
print(x.has(4)) # Output 'true'
print(x.has(5)) # Output 'true'
print(x.has(6)) # Output 'false'
```

Exclusive Range

Like inclusive range but they exclude the last element. Their literals use the \([a..b[\) notation:

```
let x := [1..5[ # x is Range[Int]
print(x.length) # Output '4'
print(x.has(5)) # Output 'false'
```

Ranges are often used in for loops.

3.6.11 None

None is the absurd class, it is the class that specialise each other except primitives classes. It is not a “real” class since it does not have a definition. It is also the only class that cannot be specialised or refined.

nil None has only one instance nil, often called the void object. nil correspond to the null constant of JAVA or the Void object of EIFFEL.

Each method invocation on nil will fail. The only exception are equality operators =, ==, !=, and !==. Therefore, it is frequent to verify if potential receiver is nil before sending a message:

```
def safe_array_length(a: Array[Int]): Int
do
    if a = nil then
        return 0
    else
        return a.length
end
```

3.6.12 Sys

The Sys built-in class

Program Start When the program starts, it instantiates the Sys class then invoke the init
The \textit{Sys} class is defined in the \texttt{kernel} module as follow:

```prm
class Sys
private
def init
    # The entry point of the program
do
        init_begin
            main
    end
def init_begin
    # Initialisation of library objects
do
end
def main
    # The main part of the program
do
end
end
```

The \texttt{init_begin} procedure is used to initialise library object. For instance, the module \texttt{io} build the standard IO file objects in \texttt{init_begin}

\texttt{main} procedure corresponds to the main statements of the module.

\section{The Base Language}

This section is dedicated to the base language programming.

\subsection{Source Structure}

PRM is a line-oriented language. PRM statements are terminated at the end of a line unless the statement is obviously incomplete—for example if the last token on a line is an operator or comma. A semicolon can be used to separate multiple expressions on a line.

Comments start with `#` and run to the end of the physical line. They are ignored during compilation. Currently, there are no multi-line comments.

```prm
# One statement, one line
a := 1

# Two statements, one line
b := 2; c := 3

# One statement, two lines
d := 4 + 5 +
    6 + 7

# Two statements, two lines
# But the second one clearly do nothing
# and may provoke a warning during compilation
e := 8 + 9
```
4.2 Name

PRM names are used to refer to variables, properties (methods and attributes), classes, and modules. The first character of a name helps PRM to distinguish its intended use.

**Reserved Names**

Certain names, listed in Table 3, are reserved and should not be used as variable, property, class, or module names.

<table>
<thead>
<tr>
<th>and</th>
<th>as</th>
<th>check</th>
<th>class</th>
<th>constructor</th>
<th>def</th>
<th>def_attr</th>
</tr>
</thead>
<tbody>
<tr>
<td>def_read</td>
<td>deferred</td>
<td>do</td>
<td>else</td>
<td>elsif</td>
<td>end</td>
<td>extern</td>
</tr>
<tr>
<td>false</td>
<td>for</td>
<td>if</td>
<td>import</td>
<td>in</td>
<td>inherit</td>
<td>intern</td>
</tr>
<tr>
<td>isa</td>
<td>let</td>
<td>let</td>
<td>new</td>
<td>nil</td>
<td>not</td>
<td>not</td>
</tr>
<tr>
<td>once</td>
<td>or</td>
<td>private</td>
<td>public</td>
<td>rename</td>
<td>return</td>
<td>self</td>
</tr>
<tr>
<td>then</td>
<td>true</td>
<td>until</td>
<td>while</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Reserved Names

In these descriptions, a lowercase letter means the characters “a” through “z”, as well as “_”, the underscore. An uppercase letter means “A” through “Z”, and digit means “0” through “9”. Name characters mean any combination of upper- and lowercase letters and digits.

A local variable name, a method name, or a module name consists of a lowercase letter followed by name characters. Examples: `foo, foo_bar_baz, _x`.

A class name starts with an uppercase letter followed by name. Examples: `Int, Any`.

An attribute name starts with an “at” sign (“@”) followed by a lowercase letter, followed by any name characters. Examples: `@name, @x, @_`.

4.3 Type

PRM is a statically typed language, it’s mean that “things” should have a static type.

**Type Annotation**

Type annotation use the PASCAL notation style: `thing: Type`. Such type annotations are used in the following places:

- Local variable declarations.
- Method signatures—for parameters, and in functions for the return type.
- Generic classes—for formal generic parameters.

**Type Language**

A type is:

- A non generic class—e.g. `Int, Car`...
- A generic type—e.g. `Array[Int], Iterator[Car]`
- A formal generic parameter.
Covariant Type Policy  As Eiffel, PRM uses a covariant typing policy.

The covariant typing policy allows the programmer to redefine properties with a more specific signature:

```eiffel
class Food end
class Grass inherit Food end
class Animal
  def eat(f: Food)
    ...
  end
class Cow inherit Animal
  def eat(g: Grass)
end
```

However, such a typing policy is unsafe. In concrete term it means that in some case, type error may occur at runtime and stop the program execution.

4.4 Expression

There are ten kinds of expressions:

- The current receiver `self`.
- Literal values.
- Function invocation.
- Variable read.
- Attribute read.
- Exported attribute read.
- Object creation.
- Boolean pseudo operators.
- Type checks.
- Once expressions.

4.4.1 Type Checks

Type check can be used to tests if an object is an instance of a giver class (or an instance of a subclass). However, since PRM uses bounded genericity and a covariant typing policy, there is a very few need of such type checks.

- The majority of their use corresponds to assertions and to assignment attempts.
- Type checks can be explicitly performed with the `isa` keyword:

In a future version, runtime type error will raise an exception.
4.4.2 Once Expression

This part of the specification is not stable and may change in a future version.

This expression is constituted by the once keyword followed by another expression called sub-expression:

```plaintext
let x := once "Message"
let y := once new Car
```

The semantic of the once expression is to evaluate the sub-expression only one time during the execution of the program. Successive evaluations of the once expression will return the first evaluated value.

Once expressions are mainly used to create singletons and to perform some local optimisation—it is often used with literals string and arrays.

Examples:

```plaintext
def only_one(i: Int): Int
do
   return once i
end
print(only_one(1)) # Output '1'
print(only_one(2)) # Output '1'
print(only_one(3)) # Output '1'
```

```plaintext
class Person
...
end
def immortal: Person
# There can be only one
do
   return once new Person
end
```

The once expression is a generalisation of the EIFFEL once keyword.

4.5 Statement

There are nine different statements in PRM: statement block, local variable declaration, assignment, procedure invocation, conditionals, while loop, for loop, return and check.

Statements are always defined in a statement block or belong to the main module statements.
4.5.1 Statement Block

Blocks of statements often start with the `do` keyword and are ended with the `end` keyword:

```plaintext
# outside
do
  # inside
  do
    # more inside
  end
  # inside again
end
# outside again
```

The only exceptions are the main statements of the program and the statements of the `if` statement.

PRM statement blocks are slightly different from other language ones:

- `do/end` differs from usual `begin/end` of others PASCAL-style languages. In fact, it is almost a 50% less characters\(^\text{10}\).
- Curly brackets from C-influenced languages do not fit with the overall PRM PASCAL style.

In a near future version, statement block will be extended to allow exception management. A potential syntax can be:

```plaintext
do
  ...
rescue e: IOException
  ...
rescue e: EmptyListException
  ...
rescue ...
finally ...
end
```

4.5.2 Local Variable Declaration

The `let` keyword is used to declare local variables:

```plaintext
let i: Int       # i is an integer
let j, k: String # j and k are strings
```

Initial Value and Type Inference

An initial value can be directly assigned with the local variable. If the initial value is present and the static type absent, the static type of the local variable is implicitly the static type of the initial value. Examples:

\(^{10}\) `do/end` statement blocks are used in some languages (PL/1, Rexx). In Ruby, `begin/end` corresponds to statement blocks, although `do/end` and curly brackets correspond to closures.
let j: Int := 5 + 3  # an integer with the value 8
let k := j + 1  # an integer with the value 9
let c := new Car("Blue")  # a blue car

Default Value  Without an explicit initial value, local variables are initialised at 0 for Int, '0' for Char, false for Bool, and nil for the other types.

Visibility  The visibility of local variable runs from its declaration until the end of the current block.

do
  # 'i' is not yet known
  let i: Int
  # 'i' is known
  do
    # 'i' is still known
  end
  # 'i' is still known
end
# 'i' is no more known

One can declare in a same block two local variables with the same name. The last declared will mask the others. However, the compiler may produce a warning.

let i: String
i := "foo"
do
  let i: Int  # Warning!
i := 5  # Correct, the Int variable
  # masks the String variable
end
i := "bar"  # Correct, the Int variable
# is no more known

PRM encourages the use of local variables to store intermediate results. Therefore, it allows the programmer to have a liberal use of local variables:

- New local variables can be declared when they are needed. Some languages like EIFFEL or SMALLTALK only allow local variable declaration at begin of subprograms. Some other languages, like ADA, MODULA 3, C, or LISP, only allow them at begin of statement blocks.

- The static type is optional any can be inferred from the initial value. This feature is quite rare in statically typed languages even if it was one of the first that appears in during the PRM specification development. MODULA-3 has it and it is planned for the future C# 3.0.

4.5.3 Assignment Statement

The assignment statement uses the quite common := and is widely used in PRM programs:

- Local variable assignment.
- Attribute assignment.
• Assignment procedure.
• Implicit parameter declaration.
• Exported attribute assignment.
• Initial local variable value.

Conformance

An assignment \( a := b \) is statically valid if the static type of the left-value \( a \) is a subtype of static type of the right-value \( b \):

\[
\begin{align*}
\text{let } x : \text{Int} & \\
x := 4 & \quad \# \text{ OK} \\
x := 'a' & \quad \# \text{ Error} \\
\text{let } y : \text{Any} & \\
y := x & \quad \# \text{ OK} \\
y := 'a' & \quad \# \text{ OK}
\end{align*}
\]

Assignment Attempt

The assignment attempt use the EIFFEL \( ?= \) notation. It works exactly like the assignment, except that conformance is not checked statically but at runtime. If an assignment attempt fails, the program execution will stop.\textsuperscript{11}

Example:

\[
\begin{align*}
\text{let } x : \text{Any} & \\
\text{let } y : \text{Int} & \\
\text{let } z : \text{Char} & \\
x := 5 & \\
y ?= x & \quad \# \text{ OK} \\
z ?= x & \quad \# \text{ Error at run-time}
\end{align*}
\]

\textsuperscript{11} In a future version, it will raise an exception.

→ ??: type check

In order to avoid run-time error, dynamic types can be checked before any assignment attempts:

\[
\begin{align*}
\text{let } x : \text{Any} & \\
\text{let } y := 0 & \\
\ldots & \\
\text{if } x \text{ isa } \text{Int} \text{ then} & \\
\quad y := x & \\
\text{end}
\end{align*}
\]

4.5.4 Conditional Statement

Conditionals use the standard if then elsif else keywords. The elsif and else parts are optional, and there can be more than one elsif part.

\[
\begin{align*}
\text{def } \text{game}(\text{guess} : \text{Int}, \text{solution} : \text{Int}) : \text{String} & \\
\# \text{ A simple game} & \\
\text{do} & \\
\quad \text{if } \text{guess} > \text{solution} \text{ then} & \\
\qquad \text{return } "\text{It’s less}" & \\
\qquad \text{elsif } \text{guess} < \text{solution} \text{ then} & \\
\qquad \quad \text{return } "\text{It’s more}" & \\
\quad \text{else} & \\
\qquad \text{return } "\text{Correct}" & \\
\text{end}
\end{align*}
\]

4.5 Statement
The question about the one-liner if was raised but we did not find a clear and concise syntax. We find only the PERL-ish post-test if:

```
instr if expr
```

but it does not satisfy us.

### 4.5.5 While Loop

The while loop is the main PRM loop structure. It is constituted with a Boolean expression and a statement block:

```python
def gcd(x, y: Int)
    # The greatest common divisor between x and y
    # using the Euclid’s algorithm
    do
        while y != 0 do
            let t := y
            y := x % y
            x := t
        end
    return x
end
```

### 4.5.6 For Loop

The for loops are used for collection traversal. It is quite different of the C or C++ for. In fact it is comparable to the PERL foreach or to the new JAVA 5.0 for/in loop.

This loop can be used with any expression subtype of the built-in class `Collection`, even those defined by the programmer. Since `Array` and `Range` are subclasses of `Collection`, here are two examples:

```python
let pricelist := [34.50, 21.95, 4.95, 8.45]
for price in pricelist do
    let gstprice := price * 1.1
    print(" Price is ", gstprice, " including GST.
")
end

for i in [0..10] do
    print(" Value of i: ", i, "\n")
end
```

In fact, the for loops are no more than a while loops adaptation. The last example with a range is equivalent with:

```python
do
    let x := [0..10].iterator
    while x.is_ok do
        let i := x.value
        print(" Value of i: ", i, "\n")
        x.next
```
4.5.7 Return Statement

This statement has two usages according to the kind of method it is used: in a procedure or in a function. In both case, it terminate the method.

In a function, the return statement is mandatory and must provide a result value that is conform to the declared result type in the signature of the function:

```java
def sign(i: Int): Int
  do
    if i > 0 then
      return 1
    elseif i < 0 then
      return -1
    else
      return 0
  end
end
```

In a procedure, the return statement is optional ant must not provide a value:

```java
def stars(nb: Int)
  do
    if nb <= 0 then
      println("I want stars.")
      return
    end
    println("A star : *")
    for nb in [2..i] do
      println("Another star : *")
    end
  end
end
```

4.5.8 Check Statement

The check statement is about correctness. It corresponds to assertions and helps to check validity of programs and identify bugs.

→ ??: Bool

A check statement is constituted by the check keyword, optionally an assertion label followed by a colon, then a Boolean expression.

```java
def hello(name: String)
  check correct_name: name !== nil and
    not name.is_empty
  println("Hello ", name)
end
```
During its execution, the program will halt on the check if the expression is evaluated to \textit{false}.

5 A PRM Conclusion

The PRM language focuses expressive, clear, simple, and coherent concepts in a statically typed object-oriented language whereas the other languages of the same family rarely focus simplicity.

Currently, the PRM specification is almost complete however some characteristics are currently instable and others are missing like constant values, enumeration types, introspection, module visibility (import/export), exceptions, contracts, regular expressions...

The standard module hierarchy needs also to be extended. Actually there is less than 6000 line of code in the base modules—that is not a lot even if PRM has a concise syntax.

The last work is about the compiler and other tools. Actually, the PRM compiler, \texttt{prmc}, is just a prototype and does not yet implement entirely the current specification. However, it produces efficient executable.

\footnote{12 In a future version, it will raise an exception.}
Index

* variable argument number, 12
  method invocation, 8
  method, 8

  Float literal, 39
  :=, see Assignment
  ?=, 49

# comment, 43
extended string literal, 39

Abstract class, 15
abstract_collection, 37
Accessor, 20
  automatic, 20
  pseudo-accessor, 21
and, 40
Any, 38
Array, 41
array, 37
Assignent
  conformance, 49
Assignment
  assignment procedure, 10
  attempt, 49
  attribute access, 7
  bracket assignment procedure, 10
  implicit parameter value, 11
  local variable initial value, 47
Attribute
  access, 4
  accessor, see Accessor
  definition, 6
  exported access, 22
  redefinition, 24
  visibility, 22

Block
  statement, 47
  visibility, 16

Bool, 40
  pseudo-operator, 40
Boolean, see Bool
Bracket
  array literal, 11
  assignment procedure, 10
  generic class, 30
  generic type, 31
  operator, 9
range literal, 12

Char, 39
Character, see Char
Character string, see String
check, 51
Class, 5
  abstract, 15
  concrete, 15
  conflict, 32
  definition, 9
  generic, 30
  instantiation, 14
  refinement, 33
  specialisation, 22
class, 5
Collection, 40
Comment, 43
Concrete class, 15
Conflict
  global property, 26
  property, 26
constructor, 14, 19
  empty, 15
  implicit, 15
  visibility, 19
def, 6
defered, 25
def_read, 20
def_write, 20
defelse, 49
def_if, 49
Escape sequence, 39
eexec, 36
export, 22, 28
Expression, 45
false, 40
Float, 39
for, 50

Garbage Collector, 16
Genericity, 30
Global property, 24
hash, 37
http, 36
if, 49

53
string, 37
string_search, 37
super, 42
    lstinline super29
Sys, 42
then, 49
type, 40
Type, 44
check, 45
generic, 31

Variable
    instance, see Attribute
    local, see Local variable
Visibility, 16
    attribute, 22
    block, 16
    constructor, 19
    implicit block, 17
    inheritance, 28
    local variable, 48
    private, 18
    public, 17
    redefinition, 28

while, 50