A Little Scheme in Pharo

Stéphane Ducasse with Guillermo Polito

March 17, 2018

Copyright 2017 by Stéphane Ducasse with Guillermo Polito.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

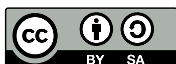
Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:
http://creativecommons.org/licenses/by-sa/3.0/

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
http://creativecommons.org/licenses/by-sa/3.0/legalcode

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations **iii**

1 Introduction **1**

2 Scheme in a (super) nutshell **3**
2.1 Physche's overview 7

3 A simple parser for Phsyche **9**
3.1 Simple interpretation architecture 9
3.2 Starting 10
3.3 Parsing input text 11

4 Limited Physche **13**
4.1 Evaluating elementary elements 13
4.2 Defining a variable 14
4.3 Introducing quote 15
4.4 Setting up the primitives 16
4.5 Adding list primitives 18
4.6 Adding if 19

5 Adding functions as a step towards closures **21**
5.1 Function 21
5.2 Defining an environment class 22
5.3 Implement an environment class 23
5.4 Procedure definition 24
5.5 Function application 25

6 Adding closures to Physche **29**
6.1 Studying a closure 29
6.2 Implementing closure 30
6.3 Adding set! and begin 32
6.4 Implementing begin 34
6.5 Fun with closures 35
6.6 Conclusion 35

7 Phemoo **37**
7.1 Reusing tests . . . 37
7.2 Phemoo interpreter . . . 37
7.3 Modeling primitives and special form . . . 38
7.4 if special form . . . 39
7.5 Initializing the environment . . . 39
7.6 Reconsidering eval:in: . . . 39
7.7 Conclusion . . . 40

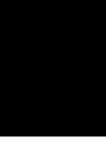
Illustrations

3-1 A naive compilation chain. 9

5-1 Function application creates an environment. 22
5-2 Examples of environment. 22

6-1 Each function application creates an environment and is evaluated in its definition environment. 31

7-1 Special forms and primitives are handled in a uniform way. 38



CHAPTER 1

Introduction

In this booklet we will build together a little interpreter for a subset of the Scheme language, that we called Physche. The idea is to implement it as simply as possible to illustrate the key aspects and share with you the fun of building language interpreters. Doing so we will explore several concepts:

- limited parsing
- basic interpreter, and
- closure concepts and implementation.

As future readings, we suggest *Structure and Interpretation of Computer Programs* by Abselson, Sussman and Sussman. I simply love it. There is also the excellent book of Jacques Chazarain (which is one of the persons who taught me Lisp) ”Programmer avec Scheme” by International Thomson Publishing.

A more personal note. We will implement a subset of Scheme because the language is simple but not trivial, really powerful and also because I love it. And while Pharo is my favorite language, it always has the taste of a Lisp language but with lovely objects. Since I implemented several mini Schemes in Scheme, I got inspired by the (How to Write a (Lisp) Interpreter (in Python)) post of Peter Norvig to write one in Pharo for fun.

Please contact me if you noticed I wrote something wrong or not fully precise.

S. Ducasse (stephane.ducasse@inria.fr) 23 December 2017

CHAPTER 2

Scheme in a (super) nutshell

We will start with a limited version of Physche, our small functional programming language inspired from Scheme. In the first version, we will not support the definition new functions (also called procedures). In the second version, we will support function definition and in particular closures. We start by presenting the subset of Scheme that we will implement.

Our objective here is not to write a Scheme following the latest language specification. We will just cover a tiny subset. Purists may not like what I will write but I take this tiny subset as a pretext for a first exploration. I will present only the parts that we will implement. Physche does not support vectors, dotted pairs, continuations, macros.

For a fast yet more complete description of Scheme I like *Teach yourself Scheme in fixnum days* by Dorai Sitaram http://ds26gte.github.io/tyscheme/index.html.

Here is simple function expressing the length of a list and one example.

```
(define len2
  (lambda (l)
    (if (null? l)
      0
      (+ 1 (len2 (cdr l))))))
```

There is another way to define functions, but for simpliciyty we will focus for now in the kind of definitions making an explicit use of `lambda`.

Once we defined our function, we can call it as follows:

```
(len2 (list 4 1 3 3))
>>> 4
```

S-Expressions

In Scheme, everything is a s-expression. A S-expression can be:

- atomic for booleans (`#t`, `#f`), number (`1`), symbols (we will treat strings as atomic),
- compound as with lists: A list starts with the opening parenthesis `(` and finishes by a closing one `)`. Lists also represent procedure application, and
- a procedure: procedures can be normal (i.e., evaluating all their arguments) or special-forms (i..e, having special ways to evaluate their arguments. This is needed to build control-flow for example).

Values

We will present booleans, number and symbols in details but focus on the procedure applications since this is a much more interesting concept. Still there is one important point to raise: the value of an atomic expression is itself. This is important since we will see that the value of list is function application.

```
#t
>>> #t
```

```
11
>>> 11
```

Procedure application

Scheme follows a prefixed syntax `(prog args ...)` where the first element refers to a function and the rest are arguments whose *values* is passed to the function. A list represents function application.

```
(+ (* 3 2) 5)
>>> 30
```

The procedure associated with the symbol `+` is looked up and the values of the arguments `(* 3 2)` and `5` are computed and passed to procedure.

By default the evaluation of a procedure application (a list) evaluates all its components. The procedure returned as value of the first element is applied to the values returned for the rest of the list.

So far, procedures evaluate *all* their arguments before starting executing the procedure. However, we will see later on that some other forms that look like procedures should not evaluate their arguments. This is the case of, for example, `define`, `lambda`, `quote` and `if`. Such procedures are called special-forms and we will have the define their semantics.

Variable definition

To define a variable and set its value, we use the `define` special form: it sets the value of the second argument to symbol represented by the first argument. Notice that `define` does not evaluate its first argument, only its second one.

```
(define pi 3.14)

(define goldenRatio (/ (+ 1 (sqrt 5)) 2))

pi
>>> 3.14
```

Defining and applying procedures

To define a procedure we use the `lambda` special form. Its first argument is a list representing the procedure arguments and the second argument the body of the procedure.

```
(lambda (x)
  (+ 2 x))
```

To use this procedure, we need to apply it an argument using the form `(proc args)`. The following piece of code shows how we can apply the argument `3`.

```
((lambda (x)
  (+ 2 x))
  3)
>>> 5
```

To reuse a function in a program, we can assign a procedure to variable using `define`.

```
(define add2 (lambda (x) (+ 2 x)))

(add2 3)
>>> 5

(add2 33)
>>> 35
```

Closures

Now a closure is more than a function. A closure refers to its definition environment. The following example illustrates it. It defines a function that returns a function. This function (having `y` as a parameter) will add `x` to `y` and `x` is bound to `3` due to the application of the first function. The function `y` is a closure that has an environment in which the variable `x` is bound to `3`.

```
(define fy3
  ((lambda (x)
    (lambda (y)
      (+ x y)))
    3))
```

```
(fy3 4)
>>> 7
```

Similarly the following function shows that we can modify this definition environment during function execution.

```
(define sy3
    ((lambda (x)
      (lambda (y)
        (begin
          (set x (+ x 2))
          (+ x y))))
      3))
```

and now each time the function `sy3` is executed its definition environment is modified and the value of `x` is incremented.

```
(sy3 5)
>>> 10
(sy3 5)
>>> 12
(sy3 5)
>>> 14
```

Quote

`quote` is an interesting special form: it does not evaluate its argument but instead returns it. It is useful when manipulating lists.

```
(quote 1)
>>> 1
```

For example, the following expressions returns then a list which looks like a function application but it just a list.

```
(quote (add2 17))
>>> (add2 17)
```

```
(quote (quote 1))
>>> (quote 1)
```

This operation is so current that it has a special syntax: `(quote x)` can be written `'x`. Physche will not support the `'` notation.

Program as data

What is particularly interesting is that the syntax of the language is centered on the one of lists and by just using a single quote or the special function `quote` we can turn a program into its abstract syntax tree.

In the following example we can access the argument of the `+` invocation by simple quoting the invocation and accessing using plain list operator such as `cdr`.

```
(cdr (quote (+ 2 3)))
>>> (2 3)
```

List as data

To manipulate a list we can simply quote it.

The following primitive procedures allows one to manipulate lists: `car` (to access the first element), `cdr` (to access the rest of the list), `cons` (to create a list) and `()` represents the empty list.

```
(quote (1 2 3 4))
>>>  (1 2 3 4)
```

```
(car (quote (1 2 3 4)))
>>> 1
```

```
(cdr (quote (1 2 3 4)))
>>> (2 3 4)
```

```
(cons 1 (quote (2 3 4)))
>>> (1 2 3 4)
```

2.1 Physche's overview

Since we do not want to have to build a full parser, we will bend a bit the syntax of Physche to be compatible with the one of Pharo.

- Booleans will be represented as `true` and `false` instead of `#t` and `#f`.
- Numbers will be the one of Pharo.
- Symbols and strings will be the ones of Pharo: `#pharo` and `'pharo'`.
- Lists will be represented as arrays to be able to get benefit from the scanner facilities of Pharo as we will explained just after. The empty list is represented by `#()`.

We are now ready to implement the first version of Physche. We will start with a first version that does not include closure and function definition. Then in a second iteration we will add closures and function definition.

CHAPTER 3

A simple parser for Phsyche

We will start to implement a extremely simple parser. Then we will define an interpreter for a limited version of the language.

3.1 Simple interpretation architecture

When implementing language compilers, parsing is the process to takes a text as input and produces an abstract representation of the program (see Figure 3-1). This process is often composed of a scanner and a parser. The scanner cuts the text into a list of tokens. And the parser consumes this list of tokens to build an intermediate representation such as an abstract syntax tree. This abstract syntax tree is then analyzed, annotated, transformed by a compiler to finally generate different code (bytecode or assembly). The generated code embeds the semantics of the implemented language.

Besides having a compiler, we can also have an interpreter, i.e., a program that executes programs of the implemented language. The idea is that the interpreter will consume the intermediate representation and act adequately. For example, when it sees a variable definition, it will declare in a structure (usually an environment) a binding for such variable.

Note that the view depicted in Figure 3-1 is naive in sense the compiler may

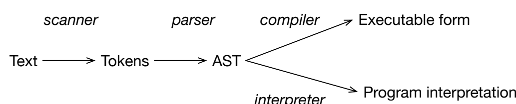


Figure 3-1 A naive compilation chain.

also emit abstract instructions (for example bytecode) that will be interpreted by an (bytecode) interpreter and may converted on the fly to assembly code. This is what the Pharo Virtual Machine does.

In our interpreter we will take a simpler route. Since Scheme syntax is simple we will just use a simple scanner and our interpreter will take as input the tokens produced by the scanner. We will use the natural structure of the arrays as simple abstract syntax trees.

3.2 Starting

Let us start by defining some tests to drive the development of Psyche's interpreter.

```
TestCase subclass: #PhsycheTest
  instanceVariableNames: 'ph'
  classVariableNames: ''
  package: 'Phsyche'
```

```
PhsycheTest >> setUp
  ph := Phsyche new
```

In the following test we see that we use the natural nesting of arrays of Pharo to represent Scheme lists.

```
PhsycheTest >> testParseLambda
  self
    assert: (ph parse: '(define squared (lambda (x) (* x x)))')
    equals: #(#define #squared #(#lambda #(#x) #(* x x)))
```

Here we check that an empty list is recognised as an empty array.

```
PhsycheTest >> testParseEmptyList
  self assert: (ph parse: '()') equals: #()
```

```
PhsycheTest >> testParseFloat
  self
    assert: (ph parse: '12.33')
    equals: 12.33
```

```
PhsycheTest >> testParseSymbol
  self
    assert: (ph parse: 'r')
    equals: #r
```

```
PhsycheTest >> testParseIsNull
  self assert: (ph parse: '(isNull (cons (quote a) #()))') equals:
     #(#isNull #(#cons #(#quote #a) #())).
  self assert: (ph parse: '(isNull (cons (quote a) ()))') equals:
     #(#isNull #(#cons #(#quote #a) #()))
```

3.3 Parsing input text

Now we are ready to implement the `parse:` method. We create the class Phsyche which is the language interperter.

```
Object subclass: #Phsyche
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Phsyche'
```

To implement the `parse:` method we take advantage of the Pharo's `Scanner` and the fact that we map list to arrays. Note that this implementation can be really bent and it absoluletely not robust but it serves our teaching purpose.

```
Phsyche >> parse: aProgramString
  aProgramString ifEmpty: [ ^ #() ].
  ^ (Scanner new scanTokens: aProgramString) first
```

As an exercise, we suggest you to represent lists with pairs as in traditional Lisp and Scheme. To do so, you will need to define a better parser.

Now we are ready to interpret the parsed programs.

CHAPTER 4

Limited Physche

Now we will add the evaluation for the limited Physche that we mentioned previously: we will not manage lambda function definition and closures.

4.1 Evaluating elementary elements

Let us start to specify the expected behavior of the language evaluation.

```
PhsycheTest >> testEvalEmptyList
  self assert: (ph parseAndEval: '()') equals: #()
```

```
PhsycheTest >> testEvalBoolean
  self assert: (ph parseAndEval: 'true') equals: true.
  self assert: (ph parseAndEval: 'false') equals: false.
```

```
PhsycheTest >> testEvalNumber
  self assert: (ph parseAndEval: '12') equals: 12.
  self assert: (ph parseAndEval: '3.14') equals: 3.14.
```

Now we can add the following methods to Phsyche

```
Phsyche >> parseAndEval: anExpression

  ^ self eval: (self parse: anExpression)
```

We define now the `eval:` method. It is a short cut to the main `eval:in:` method.

```
Phsyche >> eval: expression
  ^ self eval: expression in: nil
```

The `eval:in:` method is central to the interpreter. For now, the `eval:in:` just returns its argument. Quite limited and trivial so far.

```
Phsyche >> eval: expression in: anEnvironment
  ^ expression
```

4.2 Defining a variable

Now we will add support for the first special form: `define`. We will start with support the definition of variables.

Here is a test showing the behavior we expect.

```
PhsycheTest >> testDefineExpression
  ph parseAndEval: '(define pi 3.14)'.
  self
    assert: (ph  parseAndEval: 'pi')
    equals:  3.14.
```

First we should add a dictionary that will hold the defined variables and their values.

```
Phsyche >> initialize
  super initialize.
  environment := Dictionary new
```

We redefine the `eval:` method as follows:

```
Phsyche >> eval: expression
  ^ self eval: expression in: environment
```

Now we define a better `eval:in:` method. If the expression is a symbol, we return the value of the expression in the enviroment. Note that we do not refer to the instance variable `environment` but the parameter `anEnvironment` because in the future we will show that we may want to look method in different environment that the one of the interpreter.

```
Phsyche >> eval: expression in: anEnvironment
  expression = #()
    ifTrue: [ ^ expression ].
  expression isSymbol
    ifTrue: [ ^ anEnvironment at: expression ]. "returns the
    variable value"
  expression isArray
    ifFalse: [ "returns literals boolean, string, number" ^
    expression ]
    ifTrue: [
      expression first = #define
        ifTrue: [ ^ self evalDefineSpecialForm: expression in:
    anEnvironment ].
```

Then we check if the expression is a variable definition we define it. What you should see is that `define` is a special form since it does not evaluate its

first parameter only the second one. This is what the method `evalDefineSpecialForm:in:` is doing.

```
Phsyche >> evalDefineSpecialForm: expression in: anEnvironment
  anEnvironment
    at: expression second
    put: (self eval: expression third in: anEnvironment).
  ^ #undefined
```

The following test shows that the a variable points to a value.

```
PhsycheTest >> testEvalExpression2
  ph parseAndEval: '(define pi 3.14)'.
  ph parseAndEval: '(define pi2 pi)'.
  ph parseAndEval: '(define pi 6.28)'.
  self assert: (ph parseAndEval: 'pi2') equals: 3.14
```

4.3 Introducing quote

Quote is an interesting special form. It is the one that does not evaluate its argument.

```
PhsycheTest >> testEvalQuote
  self
    assert: (ph parseAndEval: '(quote (* x x))')
    equals: #(#* #x #x).
  self
    assert: (ph parseAndEval: '(quote (quote (* x x)))')
    equals: #(quote #(#* #x #x))
```

```
Phsyche >> eval: expression in: anEnvironment
  expression = #()
    ifTrue: [ ^ expression ].
  expression isSymbol
    ifTrue: [ ^ anEnvironment at: expression ]. "returns the
    variable value"
  expression isArray
    ifFalse: [ "returns literals boolean, string, number" ^
    expression ]
    ifTrue: [ | first |
      first := expression first.
      first = #define
        ifTrue: [ ^ self evalDefineSpecialForm: expression in:
    anEnvironment ]
      first = #quote
        ifTrue: [ ^ expression second ]
```

4.4 Setting up the primitives

Now we will introduce some primitives behavior such as addition, multiplication, list manipulation. In this implementation of Physche we will define them as block closures (a more object-oriented implementation reifying the operations is possible as we will show in the latest chapter of this booklet). Let us write a test first to specify what we want to get.

```
PhsycheTest >> testEvalExpression
  self assert: (ph parseAndEval: '(* 3 8)') equals: 24
```

```
PhsycheTest >> testEvalMoreComplexExpression
  self assert: (ph parseAndEval: '(* (+ 2 3) 8)') equals: 40.
  self assert: (ph parseAndEval: '(* 8 (+ 2 3))') equals: 40
```

We define then for example the multiplication and addition:

```
Phsyche >> multBinding
  ^ #* -> [:e :v | e * v]
```

```
Phsyche >> plusBinding
  ^ #+ ->  [:e :v | e + v]
```

The method `multBinding` returns a pair containing the primitive name and its associated Pharo closure. The primitive name will be added as a variable in the environement and its value will be the corresponding block.

```
Object subclass: #Phsyche
  instanceVariableNames: 'environment primitives'
  classVariableNames: ''
  package: 'Phsyche'
```

We redefine the `initialize` method to initialize the primitive name container and call the `initializeEnvBindings` method.

```
Phsyche >> initialize
  super initialize.
  environment := Dictionary new.
  primitives := OrderedCollection new.
  self initializeEnvBindings
```

Now we define the `initializeEnvBindings` method to automatically execute all the methods ending with 'Binding' and add the returned primitive bindings to the environment. We take the opportunity to add the primitive name to the list of primitives since it will help use later during the evaluation.

```
Phsyche >> initializeEnvBindings
  (self class selectors select: [ :each | each endsWith: 'Binding' ])
    do: [ :s |
        | binding |
        binding := self perform: s.
        primitives add: binding key.
```

```
        environment at: binding key put: binding value ]
```

Now we should change the `eval:in:` method to take into account that we have now to support primitives call. What is interesting is that we have to be clear about the semantic of primitive execution, obviously. We know that we can get the closure associated to the primitive name in the environment, and a primitive should evaluate all its arguments and pass to the closure.

```
Phsyche >> eval: expression in: anEnvironment
  expression = #()
    ifTrue: [ ^ expression ].
  expression isSymbol
    ifTrue: [ ^ anEnvironment at: expression ]. "returns the
    variable value"
  expression isArray
    ifFalse: [ "returns literals boolean, string, number" ^
    expression ]
    ifTrue: [ | first |
      first := expression first.
      (primitives includes: first)
        ifTrue: [ ^ self evalPrimitive: expression in: anEnvironment
    ]
        ifFalse: [ first = #define
            ifTrue: [ ^ self evalDefineSpecialForm: expression in:
    anEnvironment ].
          first = #quote
            ifTrue: [ ^ expression second ]]
```

At this point our tests should all pass.

Some consideration

Note that for now we consider that the mathematical operations are only working on pairs and not list of elements. We can do this changing the closure application. Another point to consider is that explicit check for primitives prevent us to overload locally their definition and this could be changed.

Some more arithmetic primitives

Here are the definitions for more primitives

```
Phsyche >> isEqualBinding
  ^ #equal -> [ :e :v | e = v ]
```

```
Phsyche >> greaterOrEqualBinding
  ^ #>= -> [ :e :v | e >= v ]
```

```
Phsyche >> isEqualBinding
  ^ #equal -> [ :e :v | e = v ]
```

```
Phsyche >> minusBinding
  ^ #- -> [ :e :v | e - v ]
```

```
Phsyche >> smallerBinding
  ^ #< -> [ :e :v | e < v ]
```

```
Phsyche >> smallerOrEqualBinding
  ^ #< -> [ :e :v | e <= v ]
```

Adding substraction and division

```
Phsyche >> minusBinding
  ^ #- -> [ :e :v | e - v ]
```

```
Phsyche >> divisionBinding
  ^ #/ -> [ :e :v | (e / v) asFloat ]
```

We add

```
PhsycheTest >> testDivision
  self should: [ ph parseAndEval: '(/ 5 0)' ] raise: ZeroDivide
```

Adding not

```
PhsycheTest >> testNot
  self assert: (ph parseAndEval: '(not false)').
  self deny: (ph parseAndEval: '(not true)')
```

```
PhsycheTest >> isNotBinding
  ^ #not -> [ :a | a not ]
```

4.5 Adding list primitives

Now we should add some primitives to manage list such as the elementary operations `cons`, `car`, and `cdr`.

Here are some tests to make sure that such primitives are acting as we expect it. Note that we expect consing is working only on list and does not produce dotted pairs.

```
PhsycheTest >> testEvalListExpression
  self assert: (ph parseAndEval: '(cons (quote a) ())') equals: #(a)
```

```
PhsycheTest >> testEvalCarExpressionEvaluatesItsArgument
  self
    assert: (ph parseAndEval:  '(car (cons (quote a) (cons (quote b)
    ())))')
    equals: #a
```

```
PhsycheTest >> testEvalCdrExpressionEvaluatesItsArgument
  self assert: (ph parseAndEval: '(cdr (quote (quote a)))') equals:
```

```
    #(a)
```

```
PhsycheTest >> testIsNull
  self assert: (ph parseAndEval: '(isNull #())').
  self assert: (ph parseAndEval: '(isNull (quote ()))').
  self deny: (ph parseAndEval: '(isNull (cons (quote a) #()))')
```

Here are the primitives definitions.

```
Phsyche >> carBinding
  ^ #car -> [ :l | l first ]
```

```
Phsyche >> cdrBinding
  ^ #cdr -> [ :l | l allButFirst ]
```

```
Phsyche >> consBinding
  ^ #cons -> [ :e :l | {e} , l ]
```

```
Phsyche >> isNullBinding
  ^ #isNull -> [ :l | l = #() ]
```

Now we can get back to the implementation of more special forms.

4.6 Adding if

Now we are ready to implement `if`. If is a special form since it does not evaluate all its arguments. Indeed, `if` should only evaluate the correct argument based on the boolean value.

```
PhsycheTest >> testEvalIf
  self assert: (ph parseAndEval: '(if true 4 5)') equals: 4.
  self assert: (ph parseAndEval: '(if false 4 5)') equals: 5
```

Well we can do better. Since numbers are auto-evaluating, this test does not verify that only one branch is evaluated.

Let us use a division by zero checks that for us as follows:

```
PhsycheTest >> testEvalIfNoSpurious
  self assert: (ph eval: (ph parse: '(if true 4 (/ 5 0))')) equals:
    4.
  self assert: (ph eval: (ph parse: '(if false (/ 5 0) 5)')) equals:
    5
```

Now we can define the `if` special form.

```
Physche >> eval: expression in: anEnvironment
  expression = #()
    ifTrue: [ ^ expression ].
  expression isSymbol
    ifTrue: [ ^ anEnvironment at: expression ]. "returns the
    variable value"
  expression isArray
```

```
    ifFalse: [ "returns literals boolean, string, number" ^
    expression ]
    ifTrue: [ | first |
      first := expression first.
      (primitives includes: first)
        ifTrue: [ ^ self evalPrimitive: expression in: anEnvironment
    ]
        ifFalse: [ first = #define
            ifTrue: [ ^ self evalDefineSpecialForm: expression in:
    anEnvironment ].
                first = #if
            ifTrue: [ ^ self evalIfSpecialForm: expression in:
    anEnvironment].
          first = #quote
            ifTrue: [ ^ expression second ]]
          ]
```

And we define the semantics of `if` execution as follows:

```
Physche >> evalIfSpecialForm: expression in: anEnvironment
  ^ (self eval: expression second in: anEnvironment)
    ifTrue: [ self eval: expression third in: anEnvironment ]
    ifFalse: [ self eval: expression fourth in: anEnvironment ]
```

Now all our lovely tests are passing.

Now our implementation is really limited in the sense that we cannot add new functions. This is what we will address in the next chapter.

CHAPTER 5

Adding functions as a step towards closures

Now we will add user function and function application to Phsyche. It is a first step towards closures. We go step by step to describe the different aspects.

5.1 Function

Let us start with a first function definition.

```
(define pi 3.14)
(define area
  (lambda (r)
    (* pi (* r r))))
```

Now we can execute the function:

```
(area 10)
>>> 314
```

Let us analyse the definition and application of the function `area`. What is important to see is that during the application `(area 10)`, the argument `r` acts as a local variable of the function. During its execution it gets the value of the argument.

Let us check that the argument value takes precedence over variables defined in outer scope.

```
(define r 5)
(define pi 3.14)
(define area
```

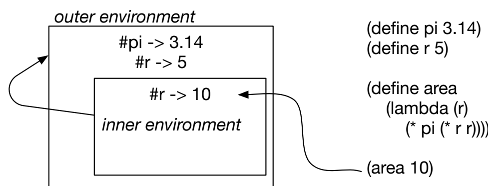


Figure 5-1 Function application creates an environment.

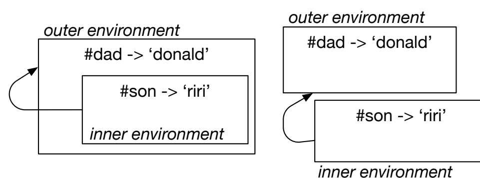


Figure 5-2 Examples of environment.

```
  (lambda (r)
    (* pi (* r r))))
```

Now the application.

```
(area 10)
>>> 314
```

It means that a function should have its own environment during application but that this environment should be linked to the global one. Indeed in the first definition of `area` the variable `pi` is found (Figure 5-1). Therefore we will define an environment.

5.2 Defining an environment class

An environment is just a dictionary that when a binding is not found locally continues the binding lookup in another environment called its parent or an outer scope (See Figure 5-2)

Let us define some tests...

```
TestCase subclass: #PEnvironmentTest
  instanceVariableNames: 'outer inner'
  classVariableNames: ''
  package: 'Physche'
```

Our setup makes sure that the inner environment is pointing to another context.

```
PEnvironmentTest >> setUp
  outer := PEnvironment new.
  inner := PEnvironment new.
  inner outerEnvironment: outer
```

The first test is to check that we can access the values set in the each level.

```
PEnvironmentTest >> testLookupAtRightLevel
  outer at: #dad put: 'donald'.
  self assert: (outer at: #dad) equals: 'donald'.
  inner at: #son put: 'riri'.
  self assert: (inner at: #son) equals: 'riri'
```

The second test is to check that we can reach the outer value from the inner one.

```
PEnvironmentTest >> testLookingOuterFromInner
  outer at: #dad put: 'donald'.
  inner at: #son put: 'riri'.
  self assert: (inner at: #dad) equals: 'donald'
```

The final test checks that unknown keys are not found.

```
PEnvironmentTest >> testLookupInFails
  outer at: #dad put: 'donald'.
  inner at: #son put: 'riri'.
  self should: [ outer at: #nodad ] raise: KeyNotFound.
  self should: [ outer at: #noson ] raise: KeyNotFound.
  self should: [ inner at: #nodad ] raise: KeyNotFound
```

We will improve the environment implementation to cover the definition of new binding but we will do that when we will add functionality to change the value of binding (i.e., implementing `set` in Phsyche).

5.3 Implement an environment class

An environment is just one special kind of dictionary that when it does not find the value of a key, looks up in its father dictionary and this recursively. We define the class `PEnvironment` as a subclass of `Dictionary` as follows:

```
Dictionary subclass: #PEnvironment
  instanceVariableNames: 'outerEnvironment'
  classVariableNames: ''
  package: 'Physche'
```

We need an accessor to set the outer context.

```
PEnvironment >> outerEnvironment: anEnvironment
  outerEnvironment := anEnvironment
```

We redefine the method `at:` so that we look up first locally for a value. When this is the case we return it, else when there is an outer environment we re-

cursively try to access the value. When there is no outer environment, we simply execute the default behavior which will lead to raise an error.

```
PEnvironment >> at: aKey
  | value |
  value := self at: aKey ifAbsent: [ nil ].
  ^ value
    ifNil: [ outerEnvironment
      ifNil: [ super at: aKey ]
      ifNotNil: [ outerEnvironment at: aKey ] ]
    ifNotNil: [ :v | v ]
```

Now we are ready to define what is a procedure (or function).

5.4 Procedure definition

We need a way to represent a function. A function has a list of parameter and a body. Let us start to write a test.

```
PhyscheTest >> testProcedureDefinition
  | proc |
  ph parseAndEval: '(define squared (lambda (x) (* x x)))'.
  proc := ph parseAndEval: #squared.
  self assert: proc parameters equals: #(#x).
  self assert: proc body equals: #(#* #x #x)
```

We should define the class `PProcedure`. It is for now straightforward.

```
Object subclass: #PProcedure
  instanceVariableNames: 'parameters body'
  classVariableNames: ''
  package: 'Phsyche'
```

Define the accessors for its instance variables. Now we will extend the interpreter to create procedure. When an expression starts with a lambda keyword, we should return a procedure.

```
Physche >> eval: expression in: anEnvironment
  expression = #()
    ifTrue: [ ^ expression ].
  expression isSymbol
    ifTrue: [ ^ anEnvironment at: expression ]. "returns the
    variable value"
  expression isArray
    ifFalse: [ "returns literals boolean, string, number" ^
    expression ]
    ifTrue: [ | first |
      first := expression first.
      (primitives includes: first)
        ifTrue: [ ^ self evalPrimitive: expression in: anEnvironment
     ]
```

```
        ifFalse: [ first = #define
            ifTrue: [ ^ self evalDefineSpecialForm: expression in:
    anEnvironment ].
          first = #lambda
            ifTrue: [ ^ self evalLambdaSpecialForm: expression in:
    anEnvironment ].
          first = #if
            ifTrue: [ ^ self evalIfSpecialForm: expression in:
    anEnvironment].
          first = #quote
            ifTrue: [ ^ expression second ] ] ]
```

```
Physche >> evalLambdaSpecialForm: expression in: anEnvironment
  ^ PProcedure new
    parameters: expression second;
    body: expression third
```

Note that this implementation of `lambda` is not correct since it does not keep a reference to its defining environment but we will do it later with closures.

Now Physche supports the *definition* of procedures but not their application. Let us look at that now.

5.5 Function application

As we saw languages following Lisp like syntax follow the pattern `(proc args)` to mean that the function `proc` is applied to the arguments `args`. Such arguments are evaluated prior to be pass to the function.

Let us write tests to control such a behavior.

```
PhyscheTest >> testLambdaProcedureExecution
  self assert: (ph parseAndEval: '((lambda (x) (* x x)) 3)') equals:
    9.
```

```
PhyscheTest >> testProcedureExecution
  ph parseAndEval: '(define squared (lambda (x) (* x x)))'.
  self assert: (ph parseAndEval: '(squared 3)') equals: 9
```

Again we will implement function application step by step to understand the different aspects.

What is important to see if that while executing a procedure body, we have to have access to the environment, for example, to access primitive definitions. We have to define a new environment where we bind the arguments to their values. Such an environment will only be used for one application. It will be the inner environment of Figure 5-1.

A first function application

The not really good implementation is then the following:

```
Phsyche >> eval: expression in: anEnvironment
  expression = #()
    ifTrue: [ ^ expression ].
  expression isSymbol
    ifTrue: [ ^ anEnvironment at: expression ]. "returns the
    variable value"
  expression isArray
    ifFalse: [ "returns literals boolean, string, number" ^
    expression ]
    ifTrue: [ | first |
      first := expression first.
      (primitives includes: first)
        ifTrue: [ ^ self evalPrimitive: expression in: anEnvironment
    ]
        ifFalse: [
            first = #define
              ifTrue: [ ^ self evalDefineSpecialForm: expression in:
    anEnvironment ].
            first = #lambda
              ifTrue: [ ^ self evalLambdaSpecialForm: expression in:
    anEnvironment ].
            first = #if
              ifTrue: [ ^ self evalIfSpecialForm: expression in:
    anEnvironment].
            first = #quote
              ifTrue: [ ^ expression second ]
              ifFalse: [
                ^ self evalApplicativeOrder: expression in:
    anEnvironment ] ] ]
```

The following method is responsible for the function application: it evaluates its first element, which will return a procedure. It creates an new environment with the parameters and the values of the arguments as values and execute the body of the procedure within this environment.

```
Phsyche >> evalApplicativeOrder: expression in: anEnvironment
  "Now we have function application ((lambda (x) (+ x 3)) (+ 9 1))"

  | proc newEnv |
  proc := self eval: expression first in: anEnvironment.
  newEnv := proc
    setEnvironmentForParameters: (expression allButFirst collect: [
    :e | self eval: e in: anEnvironment ])
    in: anEnvironment.
  ^ self eval: proc body in: newEnv
```

Note that for function application, the application environment (`newEnv`) has as parent the global interpreter environment. This will not be the case for closures as we will see later.

The method `setEnvironmentForParameters:in:` is a simple helper func-

tion that creates a new environment based on values and the procedure parameter and an outer environment.

```
PProcedure >> setEnvironmentForParameters: values in:
    outerEnvironment
  "Create a new environment inheriting from the procedure one, for
    the current application."
  | applicationEnvironment |
  applicationEnvironment := PEnvironment newFromKeys: self
    parameters andValues: values.
  applicationEnvironment outerEnvironment: outerEnvironment.
  ^ applicationEnvironment
```

The class method `newFromKeys:andValues:` only exist in Pharo 70. Here is its definition.

```
Dictionary class >> newFromKeys: keys andValues: values
  "Create a dictionary from the keys and values arguments which
    should have the same length."
  "(Dictionary newFromKeys: #(#x #y) andValues: #(3 6)) >>>
    (Dictionary new at: #x put: 3; at: #y put: 6 ;yourself)"

  | dict |
  dict := self new.
  keys with: values do: [ :k :v | dict at: k put: v ].
  ^ dict
```

With such an implementation, all our current tests should pass. Now we are ready to implement closures.

CHAPTER 6

Adding closures to Physche

Now we add closures to Phsyche. A closure is a function capturing the environment in which is defined. We start studying some examples and then we will implement on the closure semantics.

6.1 Studying a closure

A closure is a function which contains a reference to the environment at its definition time. Since evaluating a lambda defines a temporary environment in which the parameters are bound, we can use this fact to create an environment local to a function. Let us have a look at a simple example:

```
  (
   ((lambda (x)
     (lambda (y)
       (+ x y)))
    3)
   7)
>>> 10
```

The following subexpression returns a function adding 3 to its parameter `y`.

```
 ((lambda (x)
   (lambda (y)
     (+ x y)))
  3)
```

This is why the previous expression result is 10. It does so by executing first function with 3 as parameter value for `x`. This first execution creates an environment in which `x` is bound to 3. Then it returns a function taking `y` as

parameter and referring to this environment. This is why when the `(+x y)` body is executed `x` is bound to `3`.

The following expressions illustrate the same by defining a function `y` and executing such a function.

```
(define fy
 ((lambda (x)
   (lambda (y)
     (+ x y)))
  3))
```

```
(fy 7)
>>> 10
```

About let

In fact defining local environment is so frequent that Scheme and Lisp languages offer the `let` special form to define local environment as follow:

```
(let ((x 3) (+ x x))
  <=>
((lambda (x) (+ x x)) 3)
```

You can add `let` to Phsyche as an exercise.

6.2 Implementing closure

The first point is that we should change the `lambda` special form to refer to the environment in which it is defined.

To test that `lambda` effectively creates an environment. We can use this expression:

```
(define fy3
  ((lambda (x)
      (lambda (y)
       x))
    3))

(fy3 7)
```

Here the nested function having y as parameter is just returning the value of x. And during the function execution, the value of x will be looked up in the created environment. Here is a first test. Note that it is difficult to test the fact that a function declaration defines a *new* environment without using function application.

```
PhyscheTest >> testSimpleClosureIntrospection

  | proc |
```

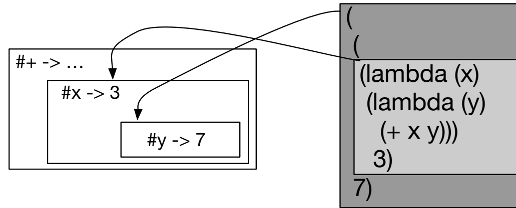


Figure 6-1 Each function application creates an environment and is evaluated in its definition environment.

```
  ph eval: (ph parseAndEval: '(define fy3
  ((lambda (x)
   (lambda (y)
      x))
  3))').
  proc := ph parseAndEval: '#fy3'.
  self assert: proc parameters equals: #(y).
  self assert: (proc environment at: #x) equals: 3.
```

```
PhyscheTest >> testSimpleClosure

  | res |
  res := ph eval: (ph parse: '(
 ((lambda (x)
   (lambda (y)
     (+ x y)))
  3)
 7)').
  self assert: res equals: 10
```

What is important to see is that the function `lambda (y)...` will be executed in an environment where `y` is bound to 7 and this environment will have as outer environment an environment with `x` bound to 3 as shown in Figure 6-1

We add the environment instance variable to the class `PProcedure` and set the current environment of the interpreter when creating the procedure in during execution of the lambda special form.

```
Object subclass: #Procedure
  instanceVariableNames: 'parameters body environment'
  classVariableNames: ''
  package: 'Pheme-Interpreters'
```

```
Phsyche >> evalLambdaSpecialForm: expression in: anEnvironment
  ^ PProcedure new
    parameters: expression second;
    body: expression third;
    environment: anEnvironment
```

Now we should revisit function evaluation to use the procedure environment instead of the one of the interpreter.

```
Phsyche >> evalApplicativeOrder: expression in: anEnvironment
  "Now we have function application ((lambda (x) (+ x 3)) (+ 9 1))"
  | proc newEnv |
  proc := self eval: expression first in: anEnvironment.
  newEnv := proc
    setEnvironmentForParameters: (expression allButFirst collect: [
    :e | self eval: e in: anEnvironment ])
    in: proc environment.
  ^ self eval: proc body in: newEnv
```

Here we create the newEnv using now the procedure environment as expressed here proc environment. What is important to see is that with closure application is that the execution environment has as outer environment the one of the procedure.


```
Phsyche >> evalApplicativeOrder: expression in: anEnvironment
  "Now we have function application ((lambda (x) (+ x 3)) (+ 9 1))"
  | proc newEnv |
  proc := self eval: expression first in: anEnvironment.
  newEnv := proc
    setEnvironmentForParameters: (expression allButFirst collect: [
    :e | self eval: e in: anEnvironment ])
    in: proc environment.
  ^ self eval: proc body in: newEnv
```


We have implemented the most important aspect of closure semantics. Now we will add some support to modify environments and conclude with this first version of Phsyche.

6.3 Adding set! and begin

To be able to experiment more with closures, we add support for changing the value a binding using the set! special form and performing a sequence of instructions using the begin special form.

The following tests specify that the modification should happen in the environment defining the existing binding. In particular when a binding is not right in the current environment but in one of the outer environment, this is the environment that contains the binding that should be modified.

```
PEnvironmentTest >> testSetAtRightLevel

  outer at: #dad put: 'donald'.
  inner at: #son put: 'riri'.
  self assert: (inner at: #son) = 'riri'.
  inner lookupAt: #son put: 'fifi'.
```

```
  self assert: (outer at: #dad) = 'donald'.
  outer lookupAt: #dad put: 'piscou'.
  self assert: (outer at: #dad) = 'piscou'.
```

```
PEnvironmentTest >> testSetLookup
  outer at: #dad put: 'donald'.
  inner at: #son put: 'riri'.
  self assert: (inner at: #dad) = 'donald'.
  inner lookupAt: #dad put: 'picsou'.
  self assert: (outer at: #dad) = 'picsou'.
  self assert: (inner at: #dad) = 'picsou'.
  self deny: (inner keys includes: #dad)
```

We implement a new method called `lookupAt:put:` that

```
PEnvironment >> lookupAt: aKey put: aValue
  "Change the value of the binding whose key is aKey, but looking in
    the complete ancestor chain.
  If the binding does not exist, it raises an error to indicate that
    we should define it first."
  | found |
  found := self at: aKey ifAbsent: nil.
  found
    ifNil: [ outerEnvironment
          ifNotNil: [ outerEnvironment lookupAt: aKey put: aValue]
          ifNil: [ KeyNotFound signal: aKey , ' not found in the
    environment' ]]
    ifNotNil: [ self at: aKey put: aValue ]
```

Now we write a simple test checking that we can change the value of a binding. We will add more complex tests once we get `begin` implemented.

```
PhyscheTest >> testEvalSimpleSet
  self assert: (ph parseAndEval: '(define x2 21') equals: #undefined.
  self assert: (ph parseAndEval: '(set x2 22)') equals: #undefined.
  self assert: (ph parseAndEval: 'x2') equals: 22.
```

Now we are ready to implement `set!`.

```
Phsyche >> eval: expression in: anEnvironment
  ...
    first = #define
      ifTrue: [ ^ self evalDefineSpecialForm: expression in:
    anEnvironment ].
    first = #set
      ifTrue: [ ^ self evalSetSpecialForm: expression in:
    anEnvironment ].
    first = #lambda
      ifTrue: [ ^ self evalLambdaSpecialForm: expression in:
    anEnvironment ].
  ...
```

```
Phsyche >> evalSetSpecialForm: expression in: anEnvironment
  anEnvironment lookupAt: expression second put: (self eval:
    expression third in: anEnvironment).
  ^ #undefined
```

Our tests should pass.

6.4 Implementing begin

The special form `begin` just evaluates one after the other the expressions in the list and return the value of the last one. The following tests are super simple but makes sure that all the elements are evaluated and that the result of the last one is returned.

```
PhyscheTest >> testEvalBegin
  self assert: (ph parseAndEval: '(begin 1 2 3)') equals: 3
```

```
PhyscheTest >> testEvalBeginSet
  self assert: (ph parseAndEval:  '(begin (define x 1) (set x 2)
    x)') equals: 2
```

Now the implementation of `begin` is the following one.

```
Phsyche >> eval: expression in: anEnvironment
  ...
    first = #define
      ifTrue: [ ^ self evalDefineSpecialForm: expression in:
    anEnvironment ].
    first = #set
      ifTrue: [ ^ self evalSetSpecialForm: expression in:
    anEnvironment ].
    first = #lambda
      ifTrue: [ ^ self evalLambdaSpecialForm: expression in:
    anEnvironment ].
    first = #begin
      ifTrue: [ ^ self evalBeginSpecialForm: expression in:
    anEnvironment ].
  ...
```

```
Phsyche >> evalBeginSpecialForm: expression in: anEnvironment
  | res |
  expression allButFirst do: [ :each | res := self eval: each in:
    anEnvironment ].
  ^ res
```

A more complex test

Now we can write a more complex test showing that we can change the binding of a variable created over function application.

```
PhyscheTest >> testEvalSetAtCorrectLevel
  | proc |
  ph parseAndEval: '
  (define fy3
    ((lambda (x)
      (lambda (y)
          (begin
            (set x (+ x 2))
            (+ x y))))
        3))
  ').
  proc := ph eval: #fy3.
  self assert: (ph parseAndEval: '(fy3 5)') equals: 10
```

We are done.

6.5 Fun with closures

Finally we can develop now little objects with an internal state. We define a function whose captured environment will keep a number that can only be modified by the function.

```
(define makeAccount
  (lambda (balance)
    (lambda (amount)
      (begin
        (set! balance (+ balance amount))
        balance))))
```

Now we can create several accounts each having its own state.

```
(define ac1 (makeAccount 1000))
(ac1 -200)
>>> 800
(define ac2 (makeAccount 2000))
(ac2 300)
>>> 2300
```

6.6 Conclusion

This ends our naive implementation of a subset of Scheme. In the following chapter we revisit the implementation to reduce the complexity of the `eval:in:` method.

CHAPTER 7

Phemoo

This chapter is optional. In this alternate implementation we treat primitives and special forms the same way. Basically we create an object for each case and give the object the possibility to specify the full evaluation.

7.1 Reusing tests

To make sure that Phemoo is doing exactly the same as Phsyche we define a subclass to PhsycheTest.

```
PhsycheTest subclass: #PhemooTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Pheme-Tests'
```

```
PhemooTest >> setUp
  super setUp.
  ph := Phemoo new
```

Now using the TestRunner you will be able to run all the tests on a Phemoo instance.

7.2 Phemoo interpreter

Phemoo has the same structure and initialization than Phsyche except that primitives and special form are represented by objects, instances of their corresponding class in the PhemooPrimitives hierarchy as shown in Figure 7-1

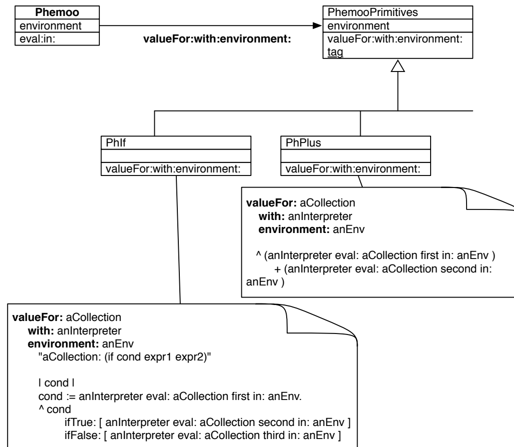


Figure 7-1 Special forms and primitives are handled in a uniform way.

```
Object subclass: #Phemoo
  instanceVariableNames: 'primitives environment'
  classVariableNames: ''
  package: 'Pheme-Phemoo'
```

7.3 Modeling primitives and special form

Primitives and special form are now expressed as subclasses of PhemooPrimitives. Each subclass should define `valueFor: aCollection with: anInterpreter environment: anEnvironment` and the class method `tag`.

```
Object subclass: #PhemooPrimitives
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Pheme-Phemoo'
```

```
PhemooPrimitives >> valueFor: aCollection with: anInterpreter
    environment: anEnvironment
  ^ self subclassResponsibility
```

```
PhemooPrimitives >> tag
  ^ self subclassResponsibility
```

Plus primitives

For example plus is expressed as follows:

```
PhemooPrimitives subclass: #PhPlus
  instanceVariableNames: ''
  classVariableNames: ''
```

```
  package: 'Pheme-Phemoo'
```

```
PhPlus class >> tag
  ^ #+
```

```
PhPlus >> valueFor: aCollection with: anInterpreter environment:
    anEnvironment
  ^  (anInterpreter eval: aCollection first in: anEnvironment )
      + (anInterpreter eval: aCollection second in: anEnvironment )
```

7.4 if special form

```
PhemooPrimitives subclass: #PhIf
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Pheme-Phemoo'
```

```
PhIf >> valueFor: aCollection with: anInterpreter environment:
    anEnvironment
  "aCollection: (if cond expr1 expr2)"

  | cond |
  cond := anInterpreter eval: aCollection first in: anEnvironment.
  ^ cond
    ifTrue: [ anInterpreter eval: aCollection second in:
    anEnvironment ]
    ifFalse: [ anInterpreter eval: aCollection third in:
    anEnvironment ]
```

```
PhIf class >> tag

  ^ #if
```

7.5 Initializing the environment

Now we specialize the initialization of the interpreter as follows:

```
Phemoo >> initializePrimitives

  ^ PhemooPrimitives allSubclasses
      do: [ :cls | primitives add: cls tag.
          environment at: cls tag put: cls new ]
```

7.6 Reconsidering eval:in:

And we are ready to have a simpler and more systematic evaluation logic.

```
Phemoo >> eval: expression in: anEnvironment
  expression = #()
    ifTrue: [ ^ expression ].
  expression isSymbol
    ifTrue: [ ^ anEnvironment at: expression ].
  expression isArray
    ifFalse: [ ^ expression ]
    ifTrue: [ (self isPrimitive: expression first)
        ifTrue: [ ^ self evaluateNonApplicativeOrder: expression in:
    anEnvironment]
        ifFalse: [ ^ self evaluateApplicativeOrder: expression in:
    anEnvironment] ]
```

This method is the same as in Phsyche.

```
Phemoo >> evaluateApplicativeOrder: expression in: anEnvironment
  "Now we have function application ((lambda (x) (+ x 3)) (+ 9 1))"
  | proc applicationEnv |
  proc := self eval: expression first in: anEnvironment.
  applicationEnv := proc
    setEnvironmentForParameters: (expression allButFirst collect: [
    :e | self eval: e in: anEnvironment])
    in: proc environment.
  ^ self eval: proc body in: applicationEnv.
```

The following method is calling each primitive to execute itself in the environment.

```
Phemoo >> evaluateNonApplicativeOrder: expression in: anEnvironment
  ^ (anEnvironment at: expression first)
    valueFor: expression allButFirst
    with: self
    environment: anEnvironment
```

We let to the use the redefinition of the primitives and special forms. There is nothing more than in Phsyche. It is just expressed differently. We let to the user the refactoring between the two interpreter to extract a common superclass.

7.7 Conclusion

We have implemented a more modular implementation. Adding a new behavior is just defining a new subclass with two methods.