Physche: A Little Scheme in Pharo

Stéphane Ducasse, Guillermo Polito

To cite this version:

Stéphane Ducasse, Guillermo Polito. Physche: A Little Scheme in Pharo. Published by the authors, 2018. hal-01900327

HAL Id: hal-01900327
https://hal.inria.fr/hal-01900327

Submitted on 21 Oct 2018

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.

Physche: A Little Scheme in Pharo

Stéphane Ducasse with Guillermo Polito

August 28, 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 Phsyche's overview 8

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 Phsyche **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 19
4.6 Adding if 20

5 Function definition and application **23**
5.1 About environments 23
5.2 Defining an environment class 24
5.3 Implement an environment class 26
5.4 Function definition 26
5.5 Function application 28

6 Adding closures to Phsyche **31**
6.1 Studying a closure 31
6.2 Implementing closure 32
6.3 Adding set! and begin 34
6.4 Implementing begin 36
6.5 Fun with closures 37
6.6 Conclusion 38

7 Phsycoo 39

7.1 Reusing tests 39
7.2 Phsycoo interpreter 39
7.3 Modeling primitives and special forms 40
7.4 The Plus primitive 41
7.5 The if special form 41
7.6 Initializing the environment 42
7.7 Reconsidering eval:in: 42
7.8 Conclusion 43

Illustrations

3-1 A naive compilation chain. 9

5-1 Function application creates an environment. 24
5-2 Two graphical representations of environments. 25

6-1 Each function application creates an environment and is evaluated in its definition environment. 33

7-1 Special forms and primitives are handled in a uniform way. 40

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, I 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.

Thanks to Clément Béra for feedback on the early version and special thanks for Quentin D. for his great feedback on the understandibility and the copy edit suggestions.

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

S. Ducasse (stephane.ducasse@inria.fr) 7 June 2018

CHAPTER 2

Scheme in a (super) nutshell

We will start with a limited version of Phsyche, our small functional programming language inspired from Scheme. In the first version, we will not support the definition of new functions (also called procedures). In the second version, we will support function definition and closures, in particular. 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 only present the parts that we will implement. Phsyche 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 simplicity we will focus for now on 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 an opening parenthesis `(` and finishes by a closing one `)`. Lists also represent function application.
- a function: functions 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 briefly present booleans, numbers and symbols. Then we focus on function application since this is a much more interesting concept.

Still there is one important point to raise: the value of an atomic expression is itself. We will see that the value of list is function application: it applies the value of the first element on the value of the rest of the list.

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

```
11
>>> 11
```

List operations

Scheme, in the path of Lisp, offers list operations such as `list` to create a list, `car` to access the first element of a list, `cdr` to access the rest of the list, `'()` to represent the empty list, and `cons` to add an element as first element of a list. Here are some examples showing their use. We do not present dotted list or pairs to keep the implementation to the bare minimum.

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

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

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

```
(cons 1 '())
>>> (1)
```

```
(cdr (list 1))
>>> ()
```

```
(null? '())
>>> #t
```

```
(null? (list 1))
>>> #f
```

Function application

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

```
(+ (* 3 2) 5)
>>> 11
```

In the code above, the function associated with the symbol `+` is looked up and the values of the arguments `(* 3 2)` and `5` are computed and passed to function.

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

So far, functions evaluate *all* their arguments before executing the function. However, we will see later on that some other forms that look like functions should not evaluate all their arguments. For example, this is the case of `define`, `lambda`, `quote` and `if`. Such functions are called special-forms and we will have to define their semantics one by one.

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 functions

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

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

To use this function, we need to apply it an argument using the form `(function 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 function 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*Phsyche 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`. Phsyche will not support the `'` notation.

List as data

To manipulate a list we can simply quote it.

The following primitive functions allow 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)
```

Program as data

What is particularly interesting is that the syntax of the language is centered on the syntax of lists and on the unification between lists as a data structure and function application.

In addition, what is really powerful is that `quote` is the bridge between function application and data structure. Indeed, the special function `quote` we can turn a program into its abstract syntax tree and we can access its elements using normal list operations such as `car` or `cdr`.

In the following example, we can access the argument of the `+` application by simply quoting the invocation. Then we can access its elements using plain list operators such as `cdr`.

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

2.1 Phsyche's overview

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

- Booleans are represented as `true` and `false` instead of `#t` and `#f`.
- Numbers are the ones 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 explain right after. The empty list is represented by `#()`.

We are now ready to implement the first version of Phsyche. We will start with a first version that only let programmers to express function use and composition without having the possibility to define their own functions. Then in a second iteration we will add closures and function definition.

CHAPTER 3

A simple parser for Phsyche

We will start to implement an 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 that 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 phase. The scanner cuts the text into a list of tokens. Then 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, and transformed by a compiler to finally generate different outputs (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 a binding for such variable in a structure (usually an environment).

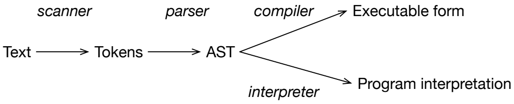


Figure 3-1 A naive compilation chain.

Note that the view depicted in Figure 3-1 is naive in sense that the compiler may also emit abstract instructions (for example bytecode) that will be interpreted by an (bytecode) interpreter and may be 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 arrays as simple abstract syntax trees.

3.2 Starting

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

```
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 really be bent and it is absolutely 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 Phsyche

In this chapter we will build *Limited Phsyche*: a simple interpreter that can evaluate atomic elements and simple function application. It will not support 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 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:` returns its argument. Quite limited and trivial so far.

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

4.2 Defining a variable

We add support for the first special form: `define`. We start by supporting variable definition.

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 holds 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 for values in a different environment than 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 ].
```

When 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 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. In addition as we mentioned in an earlier chapter, `quote` turns function application (`(prog args)`) into a manipulable data-structure: a list.

```
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

We will introduce some primitive behavior such as addition, multiplication, and list manipulation. In this implementation of Phsyche 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.

We will use a dictionary to represent the environment that keep primitives names and their respective values. But 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 the multiplication and addition as follows:

```
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 environment will contain a binding whose value is the primitive name (`#+`) and whose value will be the corresponding block.

Primitive environment initialization

To represent the environment, the interpreter defines one instance variable `environment` initialized to a dictionary. It has a `primitives` collection to keep the names of the primitives.

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

We redefine the `initialize` method to initialize the environment, the primitive name data structure (here we use an ordered collection) and to call the `initializeEnvBindings` method.

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

The `initializeEnvBindings` method automatically executes all the methods ending with `'Binding'`. We use the method `perform:` to execute a method whose name is given as argument. We add a primitive bindings to the environment and the primitive name to the primitive name list.

```
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 ]
```

The following expression returns the list of method selectors that ends with 'Binding'.

```
(self class selectors select: [ :each | each endsWith: 'Binding' ])
```

We take the opportunity to add the primitive name to the list of primitives since it will help use later during the evaluation.

Handling primitive evaluation

We are ready to implement the evaluation of a call `(primitive arg1 ... argn)`. The method `evalPrimitive:in:` finds the block closure and executes it with the values of the arguments. The method `valueWithPossibleArgs:` executes a block closure paying attention to the possible multiple number of required arguments.

```
Phsyche >> evalPrimitive: exp in: anEnvironment

  ^ (anEnvironment at: exp first)
      valueWithPossibleArgs: (exp allButFirst collect: [ :e | self
    eval: e in: anEnvironment])
```

What is interesting in this method is that this is the place that defines the semantics of the evaluation of argument calls. Here we see that the arguments are all evaluated from left to right.

We should change the `eval:in:` method to take into account that we have support for primitive 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. Another point to consider is that explicit check for primitives in the environment prevents us to overload locally their definition and this could be changed.

Some more arithmetic primitives

Here are 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

We should add some primitives to manage lists such as the elementary operations `cons`, `car`, and `cdr`.

Here are some tests to make sure that such primitives are acting as expected. Note that we consider that `cons` is only working on lists 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

We are ready to implement `if`. It has the following form: `(if condition iftrue iffalse)`. `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
```

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

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

And we can introduce this new method to the main `eval:in:` method as follows:

```
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 = #if
            ifTrue: [ ^ self evalIfSpecialForm: expression in:
    anEnvironment].
          first = #quote
            ifTrue: [ ^ expression second ]]
          ]
```

All our lovely tests are passing.

As a short conclusion, our interpreter supports the execution of expression but it is really limited in the sense that we cannot add new functions. This is what we will address in the next chapter as well as the definition of closures (functions which refer to the defining environment).

CHAPTER 5

Function definition and application

We will add user function (also called procedure in some languages) and function application to Phsyche. We go step by step to describe the different aspects of function definition and application. In the next chapter we will introduce closures.

5.1 About environments

Let us start with a first function definition.

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

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 `(area 10)` execution, `r` gets the value of the argument (here 10).

The following program checks that the argument value takes precedence over variables defined in outer scope.

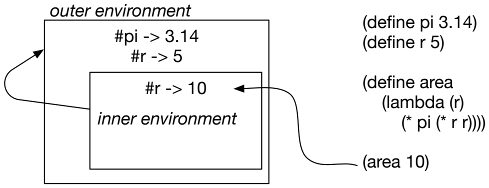


Figure 5-1 Function application creates an environment.

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

We define a variable `r` with `5` as value and the function `area` has an argument with the same name.

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

The execution of `(area 10)` shows several important points:

1. During function application, an environment should be created with the parameter and its value. The body of the function should be evaluated within the context of such an environment.
2. Second the argument value takes precedence over the value of `r` defined previously.
3. Third, when executing the body of `area` we should be able to access variable defined in *outer* scope such as `pi`.

We used a simple dictionary to represent an environment to store variables and their value. But this is not sufficient because it makes the lookup of outer variables more cumbersome. We should use a list of bindings: an environment. This way the environment created during the function application can be linked to the global as shown in Figure 5-1. With environments, we can also have function application nested into function application without any problems.

5.2 Defining an environment class

An environment is kind of linked list of bindings. It can also be seen as a dictionary that when a binding is not found locally continues the binding lookup in another environment (called its parent or outer scope). Figure 5-2 shows an example of environments.

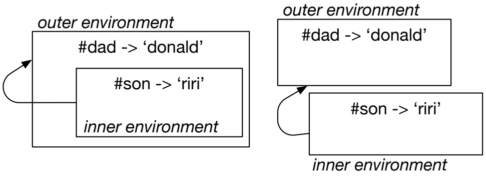


Figure 5-2 Two graphical representations of environments.

Let us define some tests first

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

Our setup makes sure that the `inner` environment is pointing to another context called `outer` here.

```
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 each scope.

```
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 scope.

```
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., when we will implement the primitive `set` for example).

5.3 Implement an environment class

There are multiple ways to implement an environment. Our implementation is just one special kind of dictionary that when it does not find the value associated to a key, looks up in its father dictionary. We define the class `PEnvironment` as a subclass of `Dictionary` as follows:

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

We need an accessor to set the outer context.

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

We redefine the method `at:` so that we first look up locally for a value. If the value is found, we return it, else when there is an outer environment we look the value in this outer environment. If 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 a function is.

5.4 Function 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.

```
PhsycheTest >> testFunctionDefinition
  | 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 define the class `PFunction` to represent functions. It is for now straightforward.

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

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

```
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 ] ] ]
```

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

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

Now Phsyche supports the *definition* of functions 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 being passed to the function.

Let us write tests to control such a behavior.

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

```
PhsycheTest >> testFunctionExecution
  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 is that while executing a function body, we have to have access to the interpreter 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 `evalApplicativeOrder:in:` is responsible for the function application: it evaluates its first element, which will return a function. It creates an new environment with the parameters and the values of the arguments as values and execute the body of the function 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 the global interpreter environment as parent. This will not be the case for closures as we will see later.

The method `setEnvironmentForParameters:in:` is a simple helper function that creates a new environment based on the function parameters and their values, and an outer environment.

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

The class method `newFromKeys:andValues:` only exists 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 Phsyche

In this chapter we add closures to Phsyche. A closure is a function capturing the environment in which it 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 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
```

This expression creates a lambda expression and apply to `7`. The execution of this lambda expression leads to the creation of another lambda expression applied to `3`.

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

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

This is why the first expression result is `10`. It does so by executing the 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 `(+ x y)` is executed, `x` is bound to `3`.

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

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

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

About let

In fact defining local environments 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

To support closures, the first point is that we should change the `lambda` special form to refer to the environment in which it is defined. Indeed at the time the function is created, it holds a reference to its defining environment.

To test that `lambda` effectively creates an environment, we use this expression:

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

(fy3 7)
```

Here the nested function (with y as parameter) is simply 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.

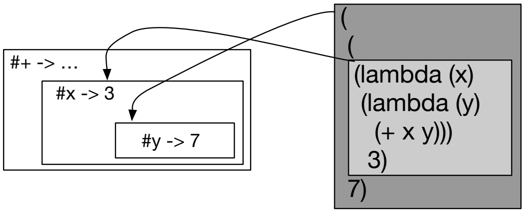


Figure 6-1 Each function application creates an environment and is evaluated in its definition environment.

Note that it is difficult to test the fact that a function declaration defines a *new* environment without using function application. This is why we are using basic introspection to access the value of the variable.

```
PhsycheTest >> testSimpleClosureIntrospection

  | proc |
  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.
```

The following test uses function application to verify the creation of the environment.

```
PhsycheTest >> 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 `PFunction` and set the current environment of the interpreter when creating the function in during execution of the lambda special form.

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

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

Now we should revisit function evaluation to use the function 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 the function environment as expressed by `proc environment`. What is important to see with closure application is that the execution environment has as outer environment the one of the function.

```
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 of a binding using the `set!` special form and performing a sequence of instructions using the `begin` special form.

The following tests specify that any modification should happen in the environment defining the existing binding. Notice that when a binding is not 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:` as follows:

```
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 ]
```

The following simple test checks that we can change the value of a binding. We will add more complex tests once we get the `begin` primitives implemented.

```
PhsycheTest >> 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.
```

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 exp1 exp2 ... expn)` evaluates one after another 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.

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

```
PhsycheTest >> 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

The following test is a bit more complex. It shows that we can change the binding of a variable created over function application. The expression `(set x (+ x 2))` modifies the value of `x` created by the function application. This is why even if the first value of `x` is `3`, it is then latter sets to `5`. Then when the function `fy` is applied to `5`, it returns `10`.

```
PhsycheTest >> 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

In this section, we will develop super naive and little account objects with an internal state, their balance. For this, we define a function whose captured environment will keep some state (a number) that can only be modified by an internal function.

The function `makeAccount` is a function with one parameter (`balance`). This function will return a function that has in its scope the `balance` variable and will be able to modify it.

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

We can create several accounts each having its own state.

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

Using the same principle you can create objects having multiple functions accessing their internal state. For this the returned function should act as a case based and execute corresponding function. In fact we prefer to use real objects such as the ones in Pharo instead of simulating them.

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

Phsycoo

This chapter is optional. In this alternate implementation we treat primitives and special forms the same way. Doing so we reduce the explicit switching logic of the `eval:in:` function. We show that the core of an interpreter can take advantage of dynamic dispatch. 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 Phsycoo is implementing the same functionality than Phsyche, we define `PhsycooTest` as a subclass of `PhsycheTest`.

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

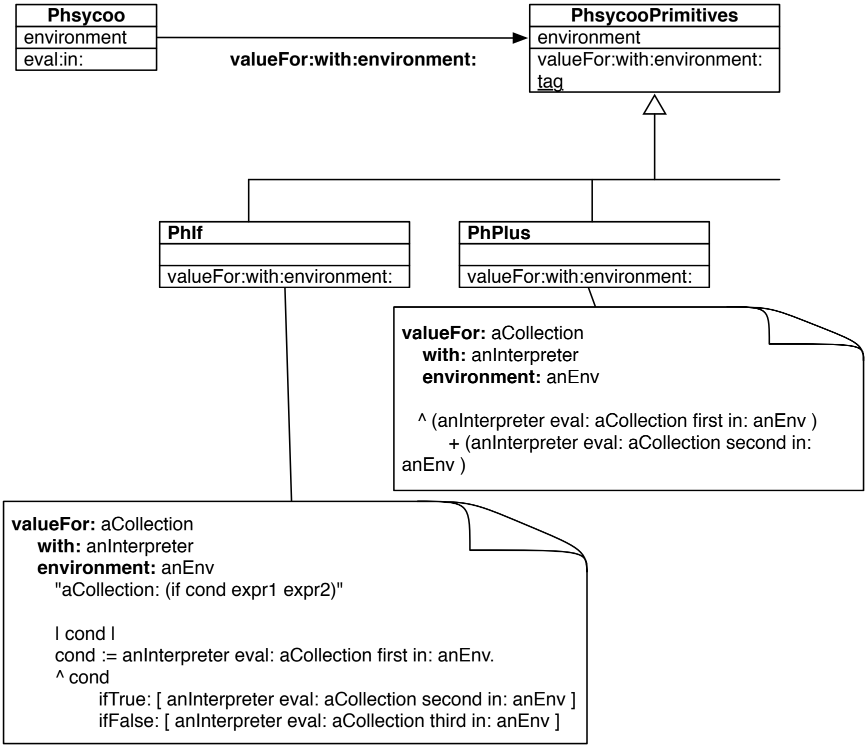
In the setup we create an instance of the Phsycoo instead of Phsyche.

```
PhsycooTest >> setUp
  super setUp.
  ph := Phsycoo new
```

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

7.2 Phsycoo interpreter

Phsycoo has the same structure and initialization than Phsyche except that primitives and special forms are uniformly represented by objects, instances


Phsycoo
environment
eval:in:
valueFor:with:environment:
PhsycooPrimitives
environment
valueFor:with:environment:
tag
PhIf
valueFor:with:environment:
PhPlus
valueFor:with:environment:

```
valueFor: aCollection
   with: anInterpreter
   environment: anEnv

  ^ (anInterpreter eval: aCollection first in: anEnv )
       + (anInterpreter eval: aCollection second in:
anEnv )
```

```
valueFor: aCollection
   with: anInterpreter
   environment: anEnv
     "aCollection: (if cond expr1 expr2)"

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

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

of their corresponding class in the PhsycooPrimitives hierarchy as shown in Figure 7-1

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

7.3 Modeling primitives and special forms

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

```
Object subclass: #PhsycooPrimitives
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Pheme-Phsycoo'
```

Given a collection representing a primitive or special form application, an interpreter and an environment, the method `valueFor:with:environment:` should return the corresponding value or modify the internal state of the interpreter.

```
PhsycooPrimitives >> valueFor: aCollection
  with: anInterpreter
  environment: anEnvironment

  ^ self subclassResponsibility
```

A tag is a simple identifier that will be used to define the primitive or special form in the interpreter environment. For plus it will be `#+`.

```
PhsycooPrimitives class >> tag
  ^ self subclassResponsibility
```

7.4 The Plus primitive

For example the primitive `plus` is expressed as follows:

```
PhsycooPrimitives subclass: #PhPlus
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Pheme-Phsycoo'
```

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

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

7.5 The if special form

The special form `if` is expressed as a subclass of `PhsycooPrimitives`.

```
PhsycooPrimitives subclass: #PhIf
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Pheme-Phsycoo'
```

The method `valueFor:with:environment:` defines the semantics of condition.

```
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
```

7.6 Initializing the environment

Once we have defined the primitives and special forms we should add them to the interpreter environment. For this, we specialize the initialization of the interpreter as follows:

```
Phsycoo >> initializePrimitives

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

This is where we see that the `tag` method is used to populate interpreter environment.

7.7 Reconsidering eval:in:

And we are ready to have a simpler and more systematic evaluation logic. As you can see it is much more compact and regular.

```
Phsycoo >> 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] ]
```

There are basically four cases

- we get an empty array (list), we return it.
- we get a symbol (value), we look it up and return it.

- we get self-evaluating entities such as #t, number, we return them.
- we get a function application: we distinguish between applicative order (this is a user defined or primitive and special forms execution (non application order).

The method `evaluateNonApplicativeOrder:in:` dispatches to the associate instance. Any primitive or special forms are executed via this calling point.

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

For applicative order, the method is the same as in Phsyche.

```
Phsycoo >> 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.
```

If we let aside the possibility to redefine primitives and special forms, there is nothing more than in Phsyche. It is just expressed differently.

7.8 Conclusion

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