Fun with Interpreters in Pharo

Stéphane Ducasse and Guillermo Polito

June 30, 2025

Keepers of the lighthouse
Édition : BoD - Books on Demand,
12/14 rond-point des Champs-Élysées,75008 Paris
Impression : Books on Demand GmbH, Norderstedt, Allemagne
ISBN: XXXXXXXXXXXXXXX
Dépôt légal : Month/YEAR
Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

CHAPTER 1

Interpreting Pharo

We are going to write a programming language interpreter. And this raises the question of the language we are going to interpret. A language that is too complex would take ages to implement, the book would be too long. Think of implementing any programming language you know: how many features does it have? Complex languages have a lot of accidental complexity (think C, C++, Java).

In this book, we have made a choice: Pharo itself. Pharo is small enough to be implementable in a couple of afternoons. And it is large enough to be interesting, as we will show you next.

After an introduction to abstract syntax trees and visitors (Chapters 2, 3, and 4) this book define interpreters of increasing level of power:

- Chapter 5 defines an interpreter of basic literal objects such as integers, floats, booleans, and literal arrays. It sets a little infrastructure to support test definition.
- Chapter 6 extends the previous interpreter with support for variables self and super.
- Chapter 7 extends the previous interpreter to support instance variables accesses. It introduces the concept of scopes. This concept will be further extended to support method execution.
- Chapter 8 introduces the infrastructures to support sending messages: a stack as well as a method scope.
- Chapter 9 extends the previous one to introduce support for parameters and temporaries.

CHAPTER 2

Code Representation with Abstract Syntax Trees

To execute actual code on top of our ObjVlisp kernel, we need code to execute. And since ObjVlisp does not force a way to represent code, we need to choose one oneselves.

Probably the simplest way to represent code we can think about are strings. That is actually what we write in editors: strings. However, strings are not the easiest to *manipulate* when we want to execute code. Instead of manipulating strings, we are going to transform strings into a more practical data structure using a parser. However, in this book we are not interested in getting into the problems of parsing (lots of books do a very fine job on that already). We will use an already existing parser and we will borrow a syntax to not define our own: Pharo's parser and Pharo's syntax.

Now that we have decided what syntax we will have at the surface of our language, we need to choose a data structure to represent our code. One fancy way to represent code is using abstract syntax trees, or in short, ASTs. An abstract syntax tree is a tree data structure that represents a program from the syntax point of view. In other words, each node in the tree represents an element that is written in a program such as variables, assignments, strings, and message sends. To illustrate it, consider the piece of Pharo code below that assigns into a variable named `variable` the result of sending the `,` message to a `'constant'` string, with `self message` as argument.

```
variable := 'constant' , self message
```

This chapter presents ASTs by looking at the existing AST implementation in

Pharo used currently by many tools in Pharo's tool-chain, such as the compiler, the syntax-highlighter, the auto-completion, the code quality engine, and the refactoring engine. As so, it's an interesting piece of engineering, and we will find it provides most of what we will need for our journey to have fun with interpreters.

In the following chapter, we will study (or re-study, for those who already know it) the Visitor design pattern. To be usable by the many tools named before, ASTs implement a visitor interface. Tools performing complex operations on ASTs may then define visitor classes with their algorithms. As we will see in the chapters after this one, one such tool is an interpreter, thus mastering ASTs and visitors is essential.

2.1 Pharo Abstract Syntax Trees

An abstract syntax tree is a tree data structure that represents a program from the syntax point of view. In the tree, nodes represent the syntactic elements of the program. The edges in the tree represent how those nodes are related. To make it concrete, Figure 2-1 shows the AST that represents the code of our previous example: `variable := 'constant' , self message`. As we will see later, each node in the tree is represented by an object, and different kinds of nodes will be instances of different classes, forming a composite.

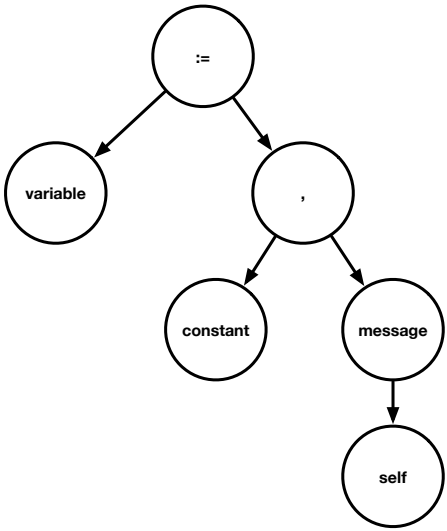

Figure 2-1 AST representing the code of `variable := 'constant' , self message`.

The Pharo standard distribution comes with a pretty complete AST implementation that is used by many tools. To get our hands over an AST, we could build it ourselves manually, or as we will do in this chapter, we ask a parser to parse some text and build an AST for us. Fortunately, Pharo also includes a parser that does exactly this: the `Parser`. The `Parser` class implements a parser for Pharo code. It has two main modes of working: parsing expressions and parsing methods.

For the Purists: Abstract vs Concrete Trees

People tend to make the distinction between abstract and concrete syntax trees. The difference is the following: an abstract syntax tree does not contain information about syntactic elements per se. For example, an abstract syntax does not contain information about parentheses since the structure of the tree itself reflects it. This is similar for variable definition delimiters (pipes) or statement delimiters (periods) in Pharo. A concrete tree on the other hand keeps such information because tools may need it. From that perspective, the Pharo AST is in between both. The tree structure contains no information about the concrete elements of the syntax, but these informations are remembered by the nodes so the source code can be rebuilt as similar as the original code as possible. However, we make a bit of language abuse and we refer to them as ASTs.

2.2 Parse Expressions

Expressions are constructs that can be evaluated to a value. For example, the program `17 max: 42` is the message-send `max:` to receiver `17` with argument `42`, and can be evaluated to the value `42` (since it is bigger than `17`).

```
| expression |
expression := OCParser parseExpression: '17 max: 42'.
expression receiver formattedCode
>>> 17

expression selector
>>> #max

expression arguments first formattedCode
>>> 42
```

Expressions are a natural instances of the composite pattern, where expressions can be combined to build more complex expressions. In the following example, the expression `17 max: 42` is used as the receiver of another message expression, the message `asString` with no arguments.

```
| expression |
expression := OCParser parseExpression: '(17 max: 42) asString'.
expression receiver formattedCode
>>> (17 max: 42)

expression selector
>>> #asString

expression arguments
>>> #()
```

Of course, message sends are not the only kind of expressions we have in Pharo. Another kind of expression that appeared already in the examples above are literal objects such as numbers.

```
| expression |
expression := OCParser parseExpression: '17'.
expression formattedCode
>>> 17
```

Pharo is a simple language, the number of different nodes that can compose the method ASTs is structured in a class hierarchy. Figure 2-2 shows the node inheritance hierarchy of Pharo rendered as a textual tree.

2.3 Literal Nodes

Literal nodes represent literal objects. A literal object is an object that is not created by sending the `new` message to a class. Instead, the developer writes directly in the source code the value of that object, and the object is created automatically from it (could be at parse time, at compile time, or at runtime, depending on the implementation). Literal objects in Pharo are strings, symbols, numbers, characters, booleans (`true` and `false`), `nil` and literal arrays (`#()`).

Literal nodes in Pharo are instances of the `ASTLiteralValueNode`, and understand the message `value` which returns the value of the object. In other words, literal objects in Pharo are resolved at parse time. Notice that the `value` message does not return a string representation of the literal object, but the literal object itself.

From now on we will omit the declaration of temporaries in the code snippets for the sake of space.

```
integerExpression := OCParser parseExpression: '17'.
integerExpression value
>>> 17
```

```
trueExpression := OCParser parseExpression: 'true'.
trueExpression value
>>> true
```

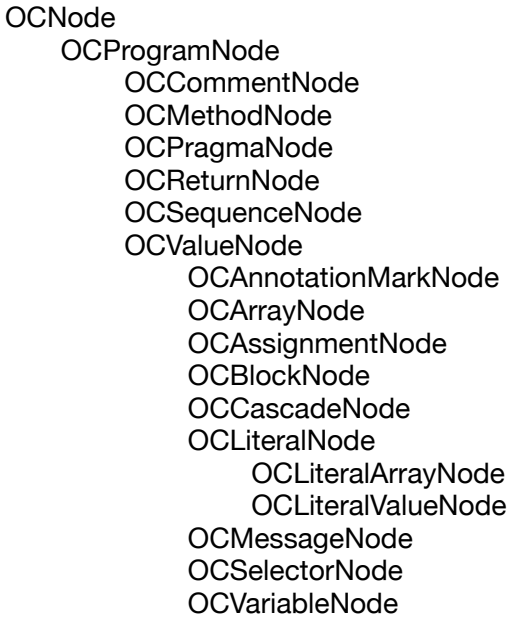

Figure 2-2 Overview of the method node hierarchy. Indentation implies inheritance.

```
"Remember, strings need to be escaped"
stringExpression := OCParser parseExpression: '''a string'''.
stringExpression value
>>> 'a string'
```

A special case of literals are literal arrays, which have their own node: `OCLiteralArrayNode`. Literal array nodes understand the message `value` as any other literal, returning the literal array instance. However, it allows us to access the sub collection of literals using the message `contents`.

```
arrayExpression := OCParser parseExpression: '#(1 2 3)'.
arrayExpression value
>>> #(1 2 3)

arrayExpression contents first
>>> OCLiteralValueNode(1)
```

In addition to messages and literals, Pharo programs can contain variables.

The Variable Node, Self, and Super Nodes

Variable nodes in the AST tree are used when variables are used or assigned to. Variables are instances of `ASTVariableNode` and know their `name`.

```
variableExpression := OCParser parseExpression: 'aVariable'.
variableExpression name
>>> 'aVariable'
```

Variable nodes are used to equally denote temporary, argument, instance, class or global variables. That is because at parse-time, the parser cannot differentiate when a variable is of one kind or another. This is especially true when we talk about instance, class and global variables, because the context to distinguish them has not been made available. Instead of complexifying the parser with this kind of information, the Pharo toolchain does it in a pipelined fashion, leaving the tools using the AST to decide on how to proceed. The parser generates a simple AST, later tools annotate the AST with semantic information from a context if required. An example of this kind of treatment is the compiler, which requires such contextual information to produce the correct final code.

For the matter of this book, we will not consider nor use semantic analysis, and we will stick with normal `OCVariableNode` objects. Later in the book we may do an optimization phase to get a richer tree that will simplify our interpretation.

2.4 Assignment Nodes

Assignment nodes in the AST represent assignment expressions using the `:=` operator. In Pharo, following Smalltalk design, assignments are expressions: their value is the value of the variable after the assignment. This allows one to chain assignments. We will see in the next chapter, when implementing an evaluator, why this is important.

An assignment node is an instance of `ASTAssignmentNode`. If we send it the `variable` message, it answers the variable it assigns to. The message `value` returns the expression at the right of the assignment.

```
assignmentExpression := OCParser parseExpression: 'var := #( 1 2 ) size'.
assignmentExpression variable
>>> OCVariableNode(var)
```

```
assignmentExpression value
>>> OCMessageNode(#(1 2) size)
```

2.5 Message Nodes

Message nodes are the core of Pharo programs, and they are the most commonly composed expression nodes we find in the AST. Messages are instances of `ASTMessageNode` and they have a receiver, a selector, and a collection of arguments, obtained through the `receiver`, `selector` and `arguments` messages. We say that message nodes are composed expressions because the `receiver` and `arguments` of a message are expressions in themselves, which can be as simple as literals or variables, or other composed messages too.

```
messageExpression := OCParser parseExpression: '17 max: 42'.
messageExpression receiver
>>> OCLiteralValueNode(17)
```

Note that `arguments` is a normal collection of expressions - in the sense that there is no special node class to represent such a sequence.

```
messageExpression arguments
>>> an OrderedCollection(OCLiteralValueNode(42))
```

And that the message selector returns also just a symbol.

```
messageExpression selector
>>> #max:
```

A note on message nodes and precedence

For those readers that already mastered the syntax of Pharo, you remember that there exist three kinds of messages: unary, binary, and keyword messages. Besides their number of parameters, the Pharo syntax accords an order of precedence between them too, i.e., unary messages get to be evaluated before binary messages, which get to be evaluated before keyword messages. Only parentheses override this precedence. The precedence of messages in ASTs is resolved at parse-time. In other words, the output of `OCParser` is an AST respecting the precedence rules of Pharo.

Let's consider a couple of examples illustrating this, illustrated in Figure 2-3. If we feed the parser with the expression below, it will create an `OCMessageNode` as we already know it. The root of that message node is the `keyword:` message, and its first argument is the `argument + 42 unaryMessage` subexpression. That subexpression is in turn another message node with the `+` binary selector, whose first argument is the `42 unaryMessage` subexpression.

```
variable keyword: argument + 42 unaryMessage
```

Now, let's change the expression by adding extra parenthesis as in:

```
variable keyword: (argument + 42) unaryMessage
```

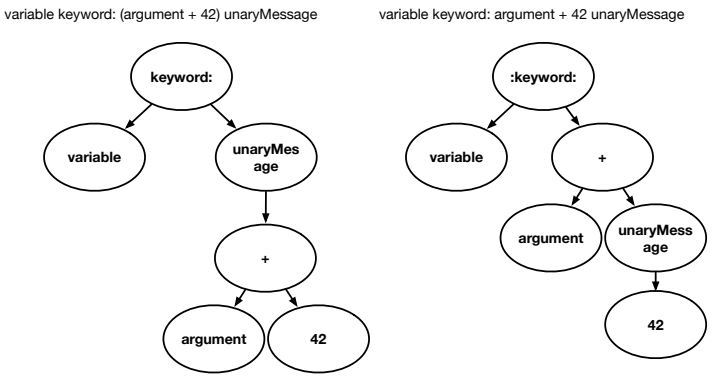

Figure 2-3 Different precedence results in different ASTs.

The resulting AST completely changed! The root is still the `keyword:` message, but now its first argument is the `unaryMessage` sent to a (now in parenthesis) `(argument + 42)` receiver.

Finally, if we modify the parenthesis again to wrap the keyword message, the root of the resulting AST has changed too. It is now the `+` binary message.

```
(variable keyword: argument) + 42 unaryMessage
```

`OCParser` is a nice tool to play with Pharo expressions and master precedence!

2.6 Cascade Nodes

Cascade nodes represent cascaded message expressions, i.e., messages sent to the same receiver. Cascaded messages are messages separated by semi-colons (`;`) such as in the following example.

```
OrderedCollection new
    add: 17;
    add: 42;
    yourself
```

This cascade is, in practical terms, equivalent to a sequence of messages to the same receiver:

```
t := OrderedCollection new.
t add: 17.
t add: 42.
t yourself
```

However, in contrast with the sequence above, cascades are expressions: their value is the value of the last message in the cascade.

A cascade node is an instance of `OCCascadeNode`. A cascade node understands the `receiver` message, returning the receiver of the cascade. It also understands the `messages` message, returning a collection with the messages in the cascade. Note that the messages inside the cascade node are normal `ASTMessageNode` and have a receiver too. They indeed share the same receiver than the cascade. In the following chapters we will have to be careful when manipulating cascade nodes, to avoid to wrongly manipulate twice the same receiver.

```
cascadeExpression := OCParser parseExpression: 'var msg1; msg2'.
cascadeExpression receiver
>>> OCVariableNode(var)

cascadeExpression messages
>>> an OrderedCollection(OCMessageNode(var msg1) OCMessageNode(var msg2))
```

2.7 Dynamic Literal Array Nodes

Pharo has dynamic literal arrays. A dynamic literal array differs from a literal array in that its elements are calculated at runtime instead of at parse time. To delay the execution of the elements in the dynamic array, a dynamic array node contains expressions, separated by dots.

```
{ 1 + 1 . self message . anObject doSomethingWith: anArgument + 3 }
```

Dynamic literal arrays nodes are instances of `OCArrayNode`. To access the expressions inside a dynamic array node, they understand the message `children`

```
arrayNode := OCParser parseExpression: '{
  1 + 1 .
  self message .
  anObject doSomethingWith: anArgument + 3 }'.

arrayNode children.
>>> an OrderedCollection(
  OCMessageNode((1 + 1))
  OCMessageNode(self message)
  OCMessageNode((anObject doSomethingWith: anArgument + 3)))
```

2.8 Method and Block Nodes

Now that we have studied the basic nodes representing expressions, we can build up methods from them. Methods are represented as instances of `ASTMethodNode` and need to be parsed with a variant of the parser we have used so far, a method parser. The `OCParser` class fulfills the role of a method parser when we use the message `parseMethod:` instead of `parseExpression:`. For example, the following piece of code returns an `ASTMethodNode` instance for a method named `myMethod`.

```
methodNode := OCParser parseMethod: 'myMethod
  1+1.
  self'
```

A method node differs from the expression nodes that we have seen before by the fact that method nodes can only be roots in the AST tree. Method nodes cannot be children of other nodes. This differs from other block-based programming languages in which method definitions are indeed expressions or statements that can be nested. In Pharo, method definitions are not statements: like class definitions, they are top-level elements. This is why Pharo is not a block structure language, even if it has closures (named blocks) that can be nested, passed as arguments, or stored.

Method nodes have a name or selector, accessed through the `selector` message, a list of arguments, accessed through the `arguments` message, and as we will see in the next section they also contain a body with the list of statements in the method.

```
methodNode selector
>>> #myMethod
```

2.9 Sequence Nodes

Method nodes have a body, represented as an `ASTSequenceNode`. A sequence node is a sequence of instructions or statements. All expressions are statements, including all nodes we have already seen such as literals, variables, arrays, assignments and message sends. We will introduce later two more kinds of nodes that can be included as part of a sequence node: block nodes and return nodes. Block nodes are expressions that are syntactically and thus structurally similar to methods. Return nodes, representing the return instruction ^, are statement nodes but not expression nodes, i.e., they can only be children of sequence nodes.

If we take the previous example, we can access the sequence node body of our method with the `body` message.

```
methodNode := OCParser parseMethod: 'myMethod
  1+1.
  self'.

methodNode body
>>> OCSequenceNode(1 + 1. self)
```

And we can access and iterate the instructions in the sequence by asking it its `statements`.

```
methodNode body statements.
>>> an OrderedCollection(OCMessageNode(1 + 1) OCSelfNode(self))
```

Besides the instructions, sequence nodes also are the ones defining temporary variables. Consider for example the following method defining a temporary.

```
myMethod
  | temporary |
  1+1.
  self'
```

In an AST, temporary variables are defined as part of the sequence node and not the method node. This is because temporary variables can be defined inside a block node, as we will see later. We can access the temporary variables of a sequence node by asking it for its `temporaries`.

```
methodNode := OCParser parseMethod: 'myMethod
  | temporary |
  1+1.
  self'.
methodNode body temporaries.
>>> an OrderedCollection(OCVariableNode(temporary))
```

2.10 Return Nodes

AST return nodes represent the instructions that are syntactically identified by the caret character ^. Return nodes, instances of `ASTReturnNode` are not expression nodes, i.e., they can only be found as a direct child of sequence nodes. Return nodes represent the fact of returning a value, and that value is an expression, which is accessible through the `value` message.

```
methodNode := OCParser parseMethod: 'myMethod
  1+1.
  ^ self'.

returnNode := methodNode body statements last.
>>>OCReturnNode(^ self)

returnNode value.
>>>OCSelfNode(self)
```

Note that as in Pharo return statements are not mandatory in a method, they are not mandatory in the AST either. Indeed, we can have method ASTs without return nodes. In those cases, the semantics of Pharo specifies that `self` is implicitly returned. It is interesting to note that the AST does not contain semantics but only syntax: we will give semantics to the AST when we evaluate it in a subsequent chapter. In Pharo this is the compiler that ensures that a method always returns self when return statements are absent in some execution paths.

Also, as we said before, return nodes are not expressions, meaning that we cannot write any of the following:

```
x := ^ 5
```

```
{ 1 . ^ 4 }
```

Block Nodes

Block nodes represent block closure expressions. A block closure is an object syntactically delimited by square brackets `[ ]` that contains statements and can be evaluated using the `value` message and its variants. The block node is the syntactic counterpart of a block closure: it is the expression that, when evaluated, will create the block object.

Block nodes work by most means like method nodes: they have a list of arguments and a sequence node as a body containing temporaries and statements. They differentiate from methods in two aspects: first, they do not have a selector, second, they are expressions (and thus can be parsed with `parseExpression:`). They can be stored in variables, passed as message arguments, and returned by messages.

```
blockNode := OCParser parseExpression: '[ :arg | | temp | 1 + 1. temp ]'.
blockNode arguments
>>>an OrderedCollection(OCVariableNode(arg))

blockNode body temporaries
>>>an OrderedCollection(OCVariableNode(temp))

blockNode body statements
>>>an OrderedCollection(OCMessageNode(1 + 1) OCVariableNode(temp))
```

2.11 Basic ASTs Manipulations

We have already covered all of Pharo AST nodes, and how to access the information they contain. Those knowing ASTs for other languages would have noticed that we have indeed few nodes. This is because in Pharo, control-flow statements such as conditionals or loops are expressed as messages, so no special case for them is required in the syntax. Because of this, Pharo's syntax fits in a postcard.

In this section, we will explore some core-messages of Pharo's AST, that allow common manipulation for all nodes: iterating the nodes, storing meta-data and testing methods. Most of these manipulations are rather primitive and simple. In the next chapter, we will see how the visitor pattern in conjunction with ASTs empower us, and gives us the possibility to build more complex applications such as concrete and abstract evaluators as we will see in the next chapters.

AST Iteration

ASTs are indeed trees, and we can traverse them as any other tree. ASTs provide several protocols for accessing and iterating any AST node in a generic way.

- `aNode children`: returns a collection with the direct children of the node.
- `aNode allChildren`: returns a collection with all recursive children found from the node.
- `aNode nodesDo: aBlock`: iterates over all children and apply `aBlock` on each of them.
- `aNode parent`: returns the direct parent of the node.
- `aNode methodNode`: returns the method node that is the root of the tree. For consistency, expression nodes parsed using `parseExpression:` are contained within a method node too.

Property Store

Some manipulations require storing meta-data associated to AST nodes. Pharo ASTs provide a set of messages for storing arbitrary properties inside a node. Properties stored in a node are indexed by a key, following the API of Pharo dictionaries.

- `aNode propertyAt: aKey put: anObject`: inserts `anObject` at `aKey`, overriding existing values at `aKey`.
- `aNode hasProperty: aKey`: returns a boolean indicating if the node contains a property indexed by `aKey`.
- `aNode propertyAt: aKey`: returns the value associated with `aKey`. If `aKey` is not found, fails with an exception.
- `aNode propertyAt: aKey ifAbsent: aBlock`: returns the value associated with `aKey`. If `aKey` is not found, evaluate the block and return its value.
- `aNode propertyAt: aKey ifAbsentPut: aBlock`: returns the value associated with `aKey`. If `aKey` is not found, evaluate the block, insert the value of the block at `aKey`, and return the value.
- `aNode propertyAt: aKey ifPresent: aPresentBlock ifAbsent: anAbsentBlock`: Searches for the value associated with `aKey`. If `aKey` is found, evaluate `aPresentBlock` with its value. If `aKey` is not found, evaluate the block and return its value.

- `aNode removeProperty: aKey`: removes the property at `aKey`. If `aKey` is not found, fails with an exception.
- `aNode removeProperty: aKey ifAbsent: aBlock`: removes the property at `aKey`. If `aKey` is not found, evaluate the block, and return its value.

Testing Methods

ASTs provide a testing protocol that is useful for small applications and writing unit tests. All ASTs answer the messages `isXXX` with a boolean `true` or `false`. A first set of methods allow us to ask a node if it is of a specified type:

- `isLiteralNode`
- `isLiteralArray`
- `isVariable`
- `isAssignment`
- `isMessage`
- `isCascade`
- `isDynamicArray`
- `isMethod`
- `isSequence`
- `isReturn`

And we can also ask a node if it is an expression node or not:

- `isValue`

2.12 Exercises

Draw the AST of the following code, indicating what kind of node is each. You can help yourself by parsing and inspecting the expressions in Pharo.

Exercises on expressions

1. Draw the AST of expression `true`.
2. Draw the AST of expression `17`.
3. Draw the AST of expression `#( 1 2 true )`.
4. Draw the AST of expression `self yourself`.

5. Draw the AST of expression `a := b := 7`.
6. Draw the AST of expression `a + #( 1 2 3 )`.
7. Draw the AST of expression `a keyword: 'message'`.
8. Draw the AST of expression `(a max: 1) min: 17`.
9. Draw the AST of expression `a max: (1 min: 17)`.
10. Draw the AST of expression `a max: 1 min: 17`.
11. Draw the AST of expression `a asParser + b asParser parse: 'sometext' , somethingElse .`
12. Draw the AST of expression `(a asParser + b asParser) parse: ('sometext' , somethingElse) .`
13. Draw the AST of expression `(a asParser + b asParser parse: 'sometext') , somethingElse .`
14. Draw the AST of expression `((a asParser + b) asParser parse: 'sometext') , somethingElse .`

Exercises on Blocks

1. Draw the AST of block `[ 1 ]`.
2. Draw the AST of block `[ :a ]`.
3. Draw the AST of block `[ :a | a ]`.
4. Draw the AST of block `[ :a | a + b ]`.
5. Draw the AST of block `[ :a | a + b . 7 ]`.
6. Draw the AST of block `[ :a | [ b ] . 7 ]`.
7. Draw the AST of block `[ :a | | temp | [ ^ b ] . ^ 7 ]`.

Exercises on Methods

1. Draw the AST of method

```
someMethod
  "this is just a comment, ignored by the parser"
```

1. Draw the AST of method

```
unaryMethod
  self
```

1. Draw the AST of method

```
unaryMethod
  ^ self
```

1. Draw the AST of method

```
+ argument
  argument > 0 ifTrue: [ ^ argument ].
  ^ self
```

1. Draw the AST of method

```
+
  strange indentation
```

1. Draw the AST of method

```
foo: arg1 bar: arg2
  | temp |
  temp := arg1 bar: arg2.
  ^ self foo: temp
```

Exercises on Invalid Code

Explain why the following code snippets (and thus their ASTs) are invalid:

1. Explain why this expression is invalid `(a + 1) := b`.
2. Explain why this expression is invalid `a + ^ 81`.
3. Explain why this expression is invalid `a + ^ 81`.

Exercises on Control Flow

As we have seen so far, there is no special syntax for control flow statements (i.e., conditionals, loops...). Instead, Pharo uses normal message-sends for them (`ifTrue:`, `ifFalse:`, `whileTrue:` ...). This makes the ASTs simple, and also turns control flow statements into control flow expressions.

1. Give an example of an expression using a conditional and its corresponding AST
2. Give an example of an expression using a loop and its corresponding AST
3. What do control flow expressions return in Pharo?

2.13 Conclusion

In this chapter, we have studied AST, short for abstract syntax trees, an object-oriented representation of the syntactic structure of programs. We have also presented the implementation of ASTs available in Pharo. Pharo provides a

parser for Pharo methods and expressions that transforms a string into a tree representing the program. We have seen how we can manipulate those ASTs. Any other nodes follow a similar principle. You should have now the basis to understand the concept of ASTs and we can move on to the next chapter.

CHAPTER 3

Reminder: the Visitor Pattern

In this chapter, we review the visitor pattern. The main purpose of the Visitor pattern is to externalize an operation from a data structure. In addition, it supports the modular definition of operation (independent from each other) and encapsulating their own data.

3.1 A Mini Filesystem

For example, let's consider a file system implemented with the composite pattern, where nodes can be files or directories. This composite forms a tree, where file nodes are leaf nodes and directory nodes are non-leaf nodes.

```
Object << #FileNode
  slots: { #size };
  package: 'VisitorExample'
```

```
Object << #DirectoryNode
  slots: { #children};
  package: 'VisitorExample'
```

Using this tree, we can take advantage of the Composite pattern and the polymorphism between both nodes to calculate the total size of a node implementing a polymorphic `size` method in each class.

```
FileNode >> size [
  ^ size

DirectoryNode >> size
  ^ children sum: [ :each | each size ]
```

Now, let's consider the users of the file system library want to extend it with their own operations. If the users have access to the classes, they may extend them just by adding methods to them. However, chances are users do not have access to the library classes. One way to open the library classes is to implement the visitor **protocol**: each library class will implement a generic `acceptVisitor: aVisitor` method that will perform a re-dispatch on the argument giving information about the receiver. For example, when a `FileNode` receives the `acceptVisitor:` message, it will send the argument the message `visitFileNode:`, identifying itself as a file node.

```
FileNode >> acceptVisitor: aVisitor
  ^ aVisitor visitFileNode: self

DirectoryNode >> acceptVisitor: aVisitor
  ^ aVisitor visitDirectoryNode: self
```

In this way, we can re-implement a `SizeVisitor` that calculates the total size of a node in the file system as follows. When a size visitor visits a file, it asks the file for its size. When it visits a directory, it must iterate the children and sum the sizes. However, it cannot directly ask the `size` of the children, because only `FileNode` instances do understand it but directories do not. Because of this, we need to make a recursive call and re-ask the child node to accept the visitor. Then each node will again dispath on the size visitor.

```
SizeVisitor >> visitFileNode: aFileNode
  ^ aFileNode size

SizeVisitor >> visitDirectoryNode: aDirectoryNode
  ^ aDirectoryNode children sum: [ :each | each acceptVisitor: self ]
```

3.2 Exercises

Exercises on the Visitor Pattern

1. Implement mathematical expressions as a tree: define the classes `OpPlus`, `OpMinus`, and `OpNumber`.

Then model expressions like `1 + 8 / 3`, and two operations on them using the composite pattern: (a) calculate their final value, and (b) print the tree in pre-order. For example, the result of evaluating the previous expression is `3`, and printing it in pre-order yields the string `'/ + 1 8 3'`.

1. Re-implement the code above using a visitor pattern.
2. Add a new kind of node to our expressions: raised to by defining the class `OpRaiseTo`. Implement it in both the composite and visitor implementations.

3. About the difference between a composite and a visitor. What happens to each implementation if we want to add a new operation? And what happens when we want to add a new kind of node?

3.3 Conclusion

In this chapter we have reviewed the visitor design pattern first on a simple example, then on ASTs. The visitor design pattern allows us to extend tree-like structures with operations without modifying the original implementation.

CHAPTER 4

Manipulating ASTs with the Visitor Pattern

In the previous chapters we have seen how to create and manipulate AST nodes. The `Parser` class implements a parser of expressions and methods that returns AST nodes for the text given as an argument. With the AST manipulation methods we have seen before, we can already write queries on an AST. For example, counting the number of message-sends in an AST is as simple as the following loop.

```
count := 0.
aNode nodesDo: [ :n |
  n isMessage
    ifTrue: [ count := count + 1 ] ].
count
```

More complex manipulations, however, require more than an iteration and a conditional. When a different operation is required for each kind of node in the AST, potentially with special cases depending on how nodes are composed, one object-oriented alternative is to implement it using the Visitor design pattern.

The Pharo AST supports visitor: its nodes implement `acceptVisitor:` methods on each of the nodes. This means we can introduce operations on the AST that were not foreseen by the original developers. In such cases, it is up to us to implement a visitor object with the correct `visitXXX:` methods.

4.1 Introducing AST visitors: measuring the depth of the tree

To introduce how to implement an AST visitor on RBASTs, let's implement a visitor that returns the max depth of the tree. That is, a tree with a single node has a depth of 1. A node with children has a depth of 1 + the maximum depth amongst all its children. Let's call that visitor `DepthCalculatorVisitor`.

```
Object << #DepthCalculatorVisitor
  package: 'VisitorExample'
```

Pharo's AST nodes implement already the visitor pattern. They have an `acceptVisitor:` method that will dispatch to the visitor with corresponding visit methods.

This means we can already use our visitor but we will have to define some methods else it will break on a visit.

4.2 Visiting message nodes

Let's start by calculating the depth of the expression `1+1`. This expression is made of a message node, and two literal nodes.

```
expression := OCParser parseExpression: '1+1'.
expression acceptVisitor: DepthCalculatorVisitor new.
>>> Exception! DepthCalculatorVisitor does not understand visitMessageNode:
```

If we execute the example above, we get a debugger because `DepthCalculatorVisitor` does not understand `visitMessageNode:`. We can then proceed to introduce that method in the debugger using the button **create** or by creating it in the browser. We can implement the visit method as follows, by iterating the children to calculate the maximum depth amongst the children and then adding 1 to it.

```
DepthCalculatorVisitor >> visitMessageNode: aMessageNode
  ^ 1 + (aMessageNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

4.3 Visiting literal nodes

As soon as we restart the example, it will stop again with an exception again, but this time because our visitor does not know how to visit literal nodes. We know that literal nodes have no children, so we can implement the visit method as just returning one.

```
DepthCalculatorVisitor >> visitLiteralValueNode: aRBLiteralValueNode
  ^ 1
```

4.4 Calculating the depth of a method

A method node contains a set of statements. Statements are either expressions or return statements. The example that follows parses a method with two statements whose maximum depth is 3. The first statement, as we have seen above, has a depth of 2. The second statement, however, has depth of three, because the receiver of the + message is a message itself. The final depth of the method is then 5: 1 for the method node, 1 for the sequence node, and 3 for the statements.

```
method := OCParser parseMethod: 'method
  1+1.
  self factorial + 2'.
method acceptVisitor: DepthCalculatorVisitor new.
>>> Exception! DepthCalculatorVisitor does not understand visitMethodNode:
```

To calculate the above, we need to implement three other visiting methods: `visitMethodNode:`, `visitSequenceNode:` and `visitSelfNode:`. Since for the first two kind of nodes we have to iterate over all children in the same way, let's implement these similarly to our `visitMessageNode:`. Self nodes are variables, so they are leafs in our tree, and can be implemented as similarly to literals.

```
DepthCalculatorVisitor >> visitMethodNode: aMethodNode
  ^ 1 + (aMethodNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

```
DepthCalculatorVisitor >> visitSequenceNode: aSequenceNode
  ^ 1 + (aSequenceNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

```
DepthCalculatorVisitor >> visitVariableNode: aSelfNode
  ^ 1
```

4.5 Refactoring the implementation

This simple AST visitor does not actually require a different implementation for each of its nodes. We have seen above that we can differentiate the nodes between two kinds: leaf nodes that do not have children, and internal nodes that have children. A first refactoring to avoid the repeated code in our solution may extract the repeated methods into common ones: `visitNodeWithChildren:` and `visitLeafNode:`.

```
DepthCalculatorVisitor >> visitNodeWithChildren: aNode
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

```
DepthCalculatorVisitor >> visitMessageNode: aMessageNode
  ^ self visitNodeWithChildren: aMessageNode


DepthCalculatorVisitor >> visitMethodNode: aMethodNode
  ^ self visitNodeWithChildren: aMethodNode
```

```
DepthCalculatorVisitor >> visitSequenceNode: aSequenceNode
  ^ self visitNodeWithChildren: aSequenceNode
```

```
DepthCalculatorVisitor >> visitLeafNode: aSelfNode
  ^ 1
```

```
DepthCalculatorVisitor >> visitVariableNode: aSelfNode
  ^ self visitLeafNode: aSelfNode
```

```
DepthCalculatorVisitor >> visitLiteralValueNode: aLiteralValueNode
  ^ self visitLeafNode: aLiteralValueNode
```

4.6 Second Refactoring

As a second step, we can refactor further by taking into account a simple intuition: leaf nodes do never have children. This means that `aNode children` always yields an empty collection for leaf nodes, and thus the result of the following expression is always a program that zero:

```
(aNode children
  inject: 0
  into: [ :max :node | max max: (node acceptVisitor: self) ])
```

In other words, we can reuse the implementation of `visitNodeWithChildren:` for both nodes with and without children, to get rid of the duplicated `1+`.

Let's then rename the method `visitNodeWithChildren:` into `visitNode:` and make all visit methods delegate to it. This will allow us also to remove the, now unused, `visitLeafNode:`.

```
DepthCalculatorVisitor >> visitNode: aNode
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

```
DepthCalculatorVisitor >> visitMessageNode: aMessageNode
  ^ self visitNode: aMessageNode
```

```
DepthCalculatorVisitor >> visitMethodNode: aMethodNode
  ^ self visitNode: aMethodNode
```

```
DepthCalculatorVisitor >> visitSequenceNode: aSequenceNode
  ^ self visitNode: aSequenceNode
```

```
DepthCalculatorVisitor >> visitVariableNode: aSelfNode
  ^ self visitNode: aSelfNode
```

```
DepthCalculatorVisitor >> visitLiteralValueNode: aLiteralValueNode
  ^ self visitNode: aLiteralValueNode
```

4.7 Refactoring: A common Visitor superclass

If we take a look at our visitor above, we see a common structure has appeared. We have a lot of little visit methods per kind of node where we could do specific per-node treatments. For those nodes that do not do anything specific, with that node, we treat them as a more generic node with a more generic visit method. Our generic visit methods could then be moved to a common superclass named `BaseASTVisitor` defining the common structure, but making a single empty hook for the `visitNode:` method.

```
Object << #BaseASTVisitor
  package: 'VisitorExample'
```

```
BaseASTVisitor >> visitNode: aNode
  "Do nothing by default. I'm meant to be overridden by subclasses"
```

```
BaseASTVisitor >> visitMessageNode: aMessageNode
  ^ self visitNode: aMessageNode
```

```
BaseASTVisitor >> visitMethodNode: aMethodNode
  ^ self visitNode: aMethodNode
```

```
BaseASTVisitor >> visitSequenceNode: aSequenceNode
  ^ self visitNode: aSequenceNode
```

```
BaseASTVisitor >> visitVariableNode: aNode
  ^ self visitNode: aNode
```

```
BaseASTVisitor >> visitLiteralValueNode: aLiteralValueNode
  ^ self visitNode: aLiteralValueNode
```

And our `DepthCalculatorVisitor` is then redefined as a subclass of it:

```
BaseASTVisitor << #DepthCalculatorVisitor
  package: 'VisitorExample'
```

```
DepthCalculatorVisitor >> visitNode: aNode
  ^ 1 + (aNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
```

Fortunately for us, Pharo's ASTs already provide `ASTProgramNodeVisitor` a base class for our visitors, with many hooks to override in our specific subclasses.

4.8 Searching the AST for a Token

Calculating the depth of an AST is a pretty naïve example for a visitor because we do not need special treatment per node. It is however a nice example to introduce the concepts, learn some common patterns, and it furthermore forced us to do some refactorings and understand a complex visitor structure. Moreover, it was a good introduction for the `OCProgramNodeVisitor` class.

In this section, we will implement a visitor that does require a different treatment per node: a node search. Our node search will look for a node in the tree that contains a token matching a string. For the purposes of this example, we will keep it scoped to a **begins with** search, and will return all nodes it finds, in a depth-first in-order traversal. We leave as an exercise for the reader implementing variants such as fuzzy string search, traversing the AST in different order, and being able to provide a stream-like API to get only the next matching node on demand.

Let's then start to define a new visitor class `SearchVisitor`, subclass of `OCProgramNodeVisitor`. This class will have an instance variable to keep the token we are looking for. Notice that we need to keep the token as part of the state of the visitor: the visitor API implemented by Pharo's ASTs do not support additional arguments to pass around some extra state. This means that this state needs to be kept in the visitor. When the search matches the name of a variable or elements, we will store its name into a collection of matched node names.

```
OCProgramNodeVisitor << #SearchVisitor
  slots: { #token . #matchedNodeNames};
  package: 'VisitorExample'

SearchVisitor >> token: aToken
  token := aToken

SearchVisitor >> initialize
  super initialize.
  matchedNodeNames := OrderedCollection new
```

4.9 Searching in variables nodes

Let us define a little test

```
SearchVisitorTest >> testTokenInVariable

  | tree visitor |
  tree := OCParser parseMethod: 'one

| pharoVar |
pharoVar := 0.
pharoVar := pharoVar + 1.
^ pharoVar'.
```

```
  visitor := SearchVisitor new.
  visitor token: 'pharo'.
  visitor visit: tree.
  self assert: visitor matchedNodeNames first equals: 'pharoVar'
```

Implement the visit methods for variable nodes. A variable node matches the search if its name begins with the searched token.

```
SearchVisitor >> visitVariableNode: aNode

  (aNode name beginsWith: token) ifTrue: [
    matchedNodeNames add: aNode name ]
```

Searching in message nodes

Message nodes will match a search if their selector begins with the searched token. In addition, to follow the specification children of the message need to be iterated in depth first in-order. This means the receiver should be iterated first, then the message node itself, and finally the arguments.

The following test checks that we identify message selectors.

```
SearchVisitorTest >> testTokenInMessage

  | tree visitor |
  tree := OCParser parseMethod: 'one

self pharo2 pharoVar: 11.
'.
  visitor := SearchVisitor new.
  visitor token: 'pharo'.
  visitor visit: tree.
  self assert: visitor matchedNodeNames first equals: 'pharo2'.
  self assert: visitor matchedNodeNames second equals: 'pharoVar:'
```

```
SearchVisitor >> visitMessageNode: aNode

  aNode receiver acceptVisitor: self.
  (aNode selector beginsWith: token) ifTrue: [
    matchedNodeNames add: aNode selector ].
  aNode arguments do: [ :each | each acceptVisitor: self ]
```

4.10 Searching in literal nodes

Literal nodes contain literal objects such as strings, but also booleans or numbers. To search in them, we need to transform such values as string and then perform the search within that string.

```
SearchVisitor >> testTokenInLiteral

  | tree visitor |
  tree := OCParser parseMethod: 'one

^ #(''pharoString'')'.
  visitor := SearchVisitor new.
  visitor token: 'pharo'.
  visitor visit: tree.
  self assert: visitor matchedNodeNames first equals: 'pharoString'
```

```
SearchVisitor >> visitLiteralValueNode: aNode

  (aNode value asString beginsWith: token) ifTrue: [
    matchedNodeNames add: aNode value asString ]
```

Another design would be to return the collection of matched nodes instead of storing it inside the visitor state. The visit methods should then return a collection with all matching nodes. If no matching nodes are found, an empty collection is returned.

We let you implement it as an exercise.

4.11 Exercises

Exercises on the AST Visitors

1. Implement an AST lineariser, that returns an ordered collection of all the nodes in the AST (similar to the pre-order exercise above).
2. Extend your AST lineariser to handle different linearisation orders: breadth-first, depth-first pre-order, depth-first post-order.
3. Extend the Node search exercise in the chapter to have alternative search orders. E.g., bottom-up, look not only if the strings begin with them.
4. Extend the Node search exercise in the chapter to work as a stream: asking `next` repeatedly will yield the next occurrence in the tree, or `nil` if we arrived at the end of the tree. You can use the linearisations you implemented above.

4.12 Conclusion

The visitor design pattern allows us to extend tree-like structures with operations without modifying the original implementation. The tree-like structure, in our case the AST, needs only to implement an accept-visit protocol. Opal Compiler's ASTs implement such a protocol and some handy base visitor classes.

Finally, we implemented two visitors for ASTs: a depth calculator and a node searcher. The depth calculator is a visitor that does not require special manipulation per-node, but sees all nodes through a common view. The search visitor has a common case for most nodes, and then implements special search conditions for messages, literals and variables.

In the following chapters, we will use the visitor pattern to implement AST interpreters: a program that specifies how to evaluate an AST. A normal evaluator interpreter yields the result of executing the AST. However, we will see that abstract interpreters will evaluate an AST in an abstract way, useful for code analysis.

CHAPTER 5

Implementing an Evaluator

An evaluator is a kind of interpreter that executes a program. A Pharo evaluator is an interpreter that takes as input a Pharo program and executes each one of its statements, finally returning the result of the execution.

In this chapter and the following ones, we will implement a Pharo evaluator as an AST interpreter, using the Visitor pattern we have seen before, meaning that the input of our evaluator will be AST nodes of a program to evaluate.

For presentation purposes, we will develop the evaluator in several stages, each in a different chapter. First, in this chapter, we will show how to implement a structural evaluator, i.e., an evaluator that reads and writes the structures of objects, starting the presentation from constant values.

Later chapters will incrementally add support for other language features that deserve a chapter for themselves such as messages and blocks.

This chapter is presented in a somehow relaxed TDD (Test Driven Development) style. For each new feature, we first define the scenario we want to cover. Since we are developing an evaluator, each scenario will have some code to execute and an expected result. We then define a test for the scenario and we make it pass. Before going to the next scenario, we do some refactorings to enhance the quality of our code.

During this process, we will define and refine an AST visitor. Note that we will write the visitor from scratch but we will reuse the nodes of the Pharo AST and their functionalities.

5.1 Setting Up the Stage

To start writing our Pharo evaluator in TDD style, we will start by creating out test class `CInterpreterTest`. Our class names are prefixed with `C` because we named the package of the interpreter Champollion.

```
TestCase << #CInterpreterTest
  package: 'Champollion-Tests'
```

This class will, until a change is imperative, host all our test methods.

Preparing the Scenarios

Our scenarios, made out of classes and methods to be interpreted, need to be written somewhere too. We could store those scenarios in strings in our test class, that we will then need to parse and give to our interpreter as input. We will write our scenarios as normal Pharo code, in normal Pharo classes: We will host our first scenarios as methods in a new class named `CHInterpretable`.

This solution is simple enough and versatile to support more complex situations in the future.

```
Object << #CInterpretable
  package: 'Champollion-Test'
```

5.2 Evaluating Literals: Integers

Testing our evaluator requires that we test and assert some observable behavior. In Pharo there are two main observable behaviors: either side-effects through assignments or results of methods through return statements. We have chosen to use return statements, to introduce variables and assignments later. To start, our first scenario is a method returning an integer, as in the code below:

```
CInterpretable >> returnInteger
  ^ 5
```

Executing such a method should return an integer with value 5.

5.3 Writing a Red Test

Our first test implements what our scenario defined above: executing our method should return 5. This first test specifies not only part of the behavior of our interpreter but also helps us in defining the part of its API: we want our interpreter to be able to start executing from some method's AST. Below we define a first test for it: `testReturnInteger`.

```
CInterpreterTest >> testReturnInteger
  | ast result |
  ast := OCParser parseMethod: (CInterpretable >> #returnInteger) sourceCode.
  result := self interpreter execute: ast.
  self assert: result equals: 5
```

This first test is worth one comment: since our evaluator is an AST interpreter, it requires an AST as input. In other words, we need to get the AST of the `returnInteger` method. Instead of invoking the parser to get an AST from source code, we will use Pharo's reflective API to get the AST of an already existing method.

5.4 Making the Test Pass: a First Literal Evaluator

Executing our first test fails first because our test does not understand `interpreter`, meaning we need to implement a method for it in our test class. We implement it as a factory method in our test class, returning a new instance of `CInterpreter`, and we define the class `CInterpreter` as follows.

```
CInterpreterTest >> interpreter
  ^ CInterpreter new
```

```
Object << #CInterpreter
  package: 'Champollion'
```

The class `CInterpreter` is the main entry point for our evaluator, and it will implement a visitor pattern over the Pharo method ASTs. Note that it does not inherit from the default Pharo AST Visitor. The Pharo AST visitor already implements generic versions of the `visitXXX:` methods that will do nothing instead of failing. Not inheriting from it allows us to make it clear when something is not yet implemented: we will get problems such as does not understand exceptions that we will be able to implement them step by step in the debugger. We, nevertheless, follow the same API as the default AST visitor and we use the nodes' `accept:` visiting methods.

At this point, re-executing the test fails with a new error: our `CInterpreter` instance does not understand the message `execute:`. We implement `execute:` to call the visitor main entry point, i.e., the method `visitNode:`.

```
CInterpreter >> execute: anAST
  ^ self visitNode: anAST
```

```
CInterpreter >> visitNode: aNode
  ^ aNode acceptVisitor: self
```

Since we evaluate a method AST, when we re-execute the test, the execution halts because of the missing `visitMethodNode:`. A first implementation for this method simply continues the visit on the body of the method.

```
CInterpreter >> visitMethodNode: aMethodNode
  ^ self visitNode: aMethodNode body
```

Execution then arrives at a missing `visitSequenceNode:`. Indeed, the body of a method is a sequence node containing a list of temporary variable definitions and a list of statements. Since our scenario has only a single statement with no temporary variables, a first version of `visitSequenceNode:` ignores temporary declarations and handles all the statements paying attention that the last statement value should be returned. So we visit all the statements except the last one, and we then visit the last one and return its result.

```
CInterpreter >> visitSequenceNode: aSequenceNode
  "Visit the sequence and return the result of the last statement.
  Does not look at temporary declaration for now."

  aSequenceNode statements allButLast
    do: [ :each | self visitNode: each ].
  ^ self visitNode: aSequenceNode statements last
```

Then the visitor visits the return node, for which we define the `visitReturnNode:` method. This method simply visits the contents of the return node (invoking recursively the visitor) and returns the obtained value. At this point, the value is not yet covered by the visitor.

```
CInterpreter >> visitReturnNode: aReturnNode
  ^ self visitNode: aReturnNode value
```

Finally, the contents of the return node, the integer `5` is represented as a literal value node. To handle this node, we define the method `visitLiteralValueNode:`. The implementation just returns the value of the node, which is the integer we were looking for.

```
CInterpreter >> visitLiteralValueNode: aLiteralValueNode
  ^ aLiteralValueNode value
```

Our first test is now green and we are ready to continue our journey.

5.5 Literal Evaluation: Floats

For completeness, let's implement support for literal floats. Since we already have integer constants working, let's consider next a method returning a float literal. We can see such a scenario in the code below:

```
CInterpretable >> returnFloat
  ^ 3.14
```

Executing such a method should return `3.14`.

5.6 Writing a Test

Testing this case is straightforward, we should test that evaluating our method should return 3.14. We already defined that our interpreter understands the `execute:` message, so this test can follow the implementation of our previous test.

```
CInterpreterTest >> testReturnFloat
  | ast result |
  ast := OCParser parseMethod: (CInterpretable >> #returnFloat) sourceCode.
  result := self interpreter execute: ast.
  self assert: result equals: 3.14
```

Two discussions come from writing this test. First, this test is already green, because the case of floating point constants and integer constants exercise the same code, so nothing is to be done on this side. Second, some would argue that this test is somehow repeating code from the previous scenario: we will take care of this during our refactoring step.

5.7 Refactor: Improving the Test Infrastructure

Since we will write many tests with a similar structure during this book, it comes in handy to share some logic between them. The two tests we wrote so far show a good candidate of logic to share as repeated code we can extract.

The method `executeSelector:` extracts some common logic that will make our tests easier to read and understand: it obtains the AST of a method from its selector, evaluates it, and returns the value of the execution. We also take the opportunity to set in the AST the class from which the method is originating. It will help us in the future.

```
CInterpreterTest >> executeSelector: aSymbol
  | ast |
  ast := OCParser parseMethod: (CInterpretable >> aSymbol) sourceCode.
  ast methodClass: CInterpretable.
  ^ self interpreter execute: ast
```

And we can now proceed to rewrite our first two tests as follows:

```
CInterpreterTest >> testReturnInteger
  self
    assert: (self executeSelector: #returnInteger)
    equals: 5
```

```
CInterpreterTest >> testReturnFloat
  self
    assert: (self executeSelector: #returnFloat)
    equals: 3.14
```

We are ready to write tests for the other constants efficiently.

5.8 Boolean Evaluation

Boolean literals are the `false` and `true` objects, typically used for conditionals and control flow statements. In the previous sections we implemented support for numbers, now we introduce support for returning boolean values as follows:

```
CInterpretable >> returnBoolean
  ^ false
```

Evaluating such a method should return `false`. We define a test for our boolean scenario.

```
CInterpreterTest >> testReturnBoolean

  self deny: (self executeSelector: #returnBoolean)
```

If everything goes ok, this test will be automatically green, without the need for implementing anything. This is because booleans are represented in the AST with literal value nodes, which we have already implemented.

5.9 Literal Evaluation: Arrays

Now that we support simple literals such as booleans and numbers, let's introduce literal arrays. Literal arrays are arrays that are defined inline in methods with all their elements being other literals. Literals refer to the fact that such objects are created during the parsing of the method code and now by sending a message. For this scenario, let's define two different test scenarios: an empty literal array and a literal array that has elements.

```
CInterpretable >> returnEmptyLiteralArray
  ^ #()

CInterpretable >> returnRecursiveLiteralArray
  ^ #(true 1 #('ahah'))
```

These two methods should return the respective arrays.

5.10 Writing a Red Test

Writing tests to cover these two scenarios is again straightforward:

```
CInterpreterTest >> testReturnEmptyLiteralArray
  self
    assert: (self executeSelector: #returnEmptyLiteralArray)
    equals: #()

CInterpreterTest >> testReturnRecursiveLiteralArray
  self
    assert: (self executeSelector: #returnRecursiveLiteralArray)
```

```
    equals: #(true 1 #('ahah'))
```

5.11 Make the Test Pass: visiting literal array nodes

We have to implement the method `visitLiteralArrayNode:` to visit literal arrays. Literal arrays contain an array of literal nodes, representing the elements inside the literal array. To make our tests pass, we need to evaluate a literal array node to an array where each element is the value of its corresponding literal node. Moreover, literal arrays are recursive structures: an array can contain other arrays. In other words, we should handle the visit of literal arrays recursively. Here we return the values returned by the interpretation of the elements.

```
CInterpreter >> visitLiteralArrayNode: aLiteralArrayNode
  ^ aLiteralArrayNode contents
      collect: [ :literalNode | self visitNode: literalNode ]
      as: Array
```

This makes our tests pass, and so far there is nothing else to refactor or clean. Up until now, we did not consider any form of variable and we should handle them.

5.12 Conclusion

In this chapter, we have used the Visitor pattern over AST nodes to implement the first version of a structural evaluator. This evaluator covers the basic literals: integers, floats, booleans, and arrays. Although we have not talked about it explicitly, we also implemented a first version of the visit of statements and return nodes.

We leave for the reader the exercise of extending this prototype with support for dynamic arrays (e.g., `{ self expression. 1+1 }`).

5.13 Exercises

Add tests to show that the current interpreter can support other literal objects such as strings and symbols.

CHAPTER 6

Simple variables: self and super

In the previous chapter, we wrote the first version of an evaluator managing literals: integers, floats, arrays, and booleans. This chapter adds to our evaluator the capability of evaluating variables with a specific focus on self and super. We extend this to variables in the following chapter by introducing name resolution.

6.1 Starting up with `self` and `super`

To start working with variables, we begin with a method returning `self` as the method `returnSelf` shows it.

```
CInterpretable >> returnSelf
  ^ self
```

There is a difference between this scenario and the previous one with literals. Previous scenarios always evaluated to the same value regardless how or when they were executed.

For `self`, however, evaluating this method will return a different object if the receiver of the message changes. We have to handle this case to make sure that we can properly test the interpreter.

6.2 Writing a Red Test

To properly test how `self` behaves we need to specify a receiver object, and then assert that the returned object is that same object with an identity check. For this purpose we use as receiver a new object instance of `Object`, which

implements equality as identity, guaranteeing that the object will be not only equals, but also the same. We use an explicit identity check in our test to convey our intention.

```
CInterpreterTest >> testReturnSelf
  | receiver |
  receiver := Object new.
  "Convey our intention of checking identity by using an explicit identity
     check"
  self assert: (self
      executeSelector: #returnSelf
      withReceiver: receiver)
          == receiver
```

6.3 Introducing Support to Specify a Receiver

In this section, we implement the first version of an execution context to introduce receiver objects. This first support will get us up and running to implement initial variable support, but it will run short to properly implement message sends. This setup will be extended in later chapters to support message sends and block temporaries.

This initial support requires that the evaluator gets access to the object that is the receiver in the current method execution. For now we add an instance variable in our evaluator called `receiver`.

```
Object << #CInterpreter
  slots: { #receiver };
  package: 'Champollion'
```

We define an accessor so that we can easily adapt when we will introduce a better way representation later. This instance variable is hidden behind this accessor, allowing us to change later on the implementation without breaking users.

```
CInterpreter >> receiver
  ^ receiver
```

Then we define the method `execute:withReceiver:` which allows us to specify the receiver.

```
CInterpreter >> execute: anAST withReceiver: anObject
  receiver := anObject.
  ^ self visitNode: anAST
```

And finally, we create a new helper method in our tests `executeSelector:withReceiver:` and then redefine the method `executeSelector:` to use this new method with a default receiver.

```
CInterpreterTest >> executeSelector: aSymbol withReceiver: aReceiver
  | ast |
  ast := Parse parseMethod: (CInterpretable >> aSymbol) sourceCode.
  ^ self interpreter execute: ast withReceiver: aReceiver


CInterpreterTest >> executeSelector: aSymbol
  ^ self executeSelector: aSymbol withReceiver: nil
```

We can remove the method `execute: aMethodNode` from the class `CInterpreter`, since nobody calls it anymore. We introduced the possibility to specify a message receiver, and still keep all our previous tests passing. Our new test is still red, but we are now ready to make it pass.

6.4 Making the test pass: visiting self nodes

The interpretation of `self` is done in the method `visitVariableNode:`.

The Pharo parser resolves `self` and `super` nodes statically and creates special nodes for them, avoiding the need for name resolution. Implementing it is simple, it just returns the value of the receiver stored in the interpreter. Note that this method does not access the `receiver` variable directly, but uses instead the accessor, leaving us the possibility of redefining that method without breaking the visit.

```
CInterpreter >> visitVariableNode: aNode
  aNode name = #self
    ifTrue: [ ^ self receiver ]
```

Note that in Pharo, the compiler performs a pass and enriches the variable node with an instance of the `Variable` hierarchy. It uses then another visit method to be able to avoid such an ugly condition. You can see the differences in the returned trees by comparing the results of `(CInterpretable >> #returnSelf) parseTree` and `Parser parseMethod: (CInterpretable >> #returnSelf) sourceCode`.

6.5 Introducing super

Following the same logic as for `self`, we improve our evaluator to support `super`. We start by defining a method using `super` and its companion test.

```
CInterpretable >> returnSuper
  ^ super
```

```
CInterpreterTest >> testReturnSuper
  | receiver |
  receiver := Object new.
  "Convey our intention of checking identity by using an explicit identity
     check"
```

```
  self assert: (self
      executeSelector: #returnSuper
      withReceiver: receiver)
          == receiver
```

What the interpretation of `super` shows is that this variable is also the receiver of the message. Contrary to a common and wrong belief, `super` is not the superclass or an instance of the superclass. It simply is the receiver:

```
CInterpreter >> visitVariableNode: aNode

  aNode name = #self | (aNode name = #super)
    ifTrue: [ ^ self receiver ]
```

6.6 Conclusion

Handling `self` and `super` as variables is simple. In the following chapter we will look at other variables and propose a way to manage all the different variables by the introduction of scopes.

CHAPTER 7

Scopes and Variables

Variables are, using a very general definition, storage location associated with a name. For example, a global variable in Pharo is stored in a global dictionary of key-value associations, where the association's key is the name of the variable and its value is the value of the variable respectively.

Instance variables, on their side, are storage locations inside objects. Instance variables in Pharo have each a location in the object, specified by an index, and that index corresponds to the index of the variable's name in its class' instance variable list.

In this chapter, we will study how variables are resolved. Since variable nodes contain only their name, this involves a "name resolution" step: finding where the variable is located and accessing it accordingly following the lexical scoping (or static scoping) rules. Name resolution using lexical scoping discovers variables using the nesting hierarchy of our code, where each nesting level represents a scope. For example, classes define a scope with variables visible only by their instances; the global scope defines variables that are visible everywhere in the program, and classes are nested within the global scope. Methods define another scope where their arguments and temporary variables are specific to a given method.

In this chapter, we will implement name resolution by modeling the program scopes. Scope objects will act as adapters for the real storage locations of variables, giving us a polymorphic way to read and write variables. Moreover, scopes will be organized in a chain of scopes, that will model the lexical organization of the program.

7.1 Evaluating Variables: Instance Variable Reads

The first step is to support instance variable reads. Since such variables are evaluated in the context of the receiver object, we need a receiver object that has instance variables. We then add to `CInterpretable` an instance variable and a getter and setter for it to be able to control the values in that instance variable.

```
Object << #CInterpretable
  slots: { #x };
  package: 'Champollion-Tests'

CInterpretable >> x
  ^ x

CInterpretable >> x: anInteger
  x := anInteger
```

We implement the method `returnInstanceVariableX` to follow the pattern we used until now.

```
CInterpretable >> returnInstanceVariableX
  ^ x
```

To test the correct evaluation of the instance variable read, we check that the getter returns the value in the instance variable, which we can previously set.

```
CInterpreterTest >> testReturnInstanceVariableRead
  | receiver |
  receiver := CInterpretable new.
  receiver x: 100.
  self
    assert: (self executeSelector: #returnInstanceVariableX withReceiver:
     receiver)
    equals: 100
```

Our test is failing so we are ready to make it work.

7.2 Introducing Lexical Scopes

Variables in Pharo are of different kinds:

- instance variables represent variables stored in the evaluated message receiver,
- temporaries and arguments are variables visible only within a particular method evaluation,
- shared variables group variables that are visible to a class, its instances and its subclasses,

- global variables are variables that are in the global scope and independent of the receiver.

All these ”contexts” defining variables are also named scopes because they define the reach of variable definitions. At any point during program execution, we can organize scopes in a hierarchy of scopes following a parent relationship.

- When a method evaluates, the current scope is the method scope with all the arguments and temporary variables defined in the method.
- The parent of a method scope is the instance scope that defines all the variables for an instance.
- The parent of the instance scope is the global scope that defines all the global variables.

Notice that in this scope organization, we are not explicitly talking about classes. The instance scope takes care of that: it resolves all instance variables in the hierarchy of the receiver object. A class scope could be added later to resolve shared variables and shared pools.

7.3 Creating an Instance Scope

To make our tests go green, we need to model instance scopes. We model instance scopes with a new class named 'CInstanceScope'. This class should know the receiver object, extracts the list of instance variables from it, and know how to read and write from/to it. For now we focus on the reading part.

The method `definedVariables` simply returns all the instance variables that an instance should have. Finally accessing the value of a variable is possible using the reflective method `instVarNamed:`.

```
Object <<#CInstanceScope
  slots: {#receiver};
  package: 'Champollion'

CInstanceScope >> receiver: anObject
    receiver := anObject

CInstanceScope >> definedVariables
  ^ receiver class allInstVarNames

CInstanceScope >> read: aString
  ^ receiver instVarNamed: aString
```

Now we can define the method `currentScope` in the interpreter with a first implementation. We will refine in the following sections.

```
CInterpreter >> currentScope
    ^ CInstanceScope new
      receiver: self receiver;
      yourself
```

7.4 Using the Scope

The basic structure of our lexical scope implementation introduces the method `scopeDefining:`. This method returns the scope defining the given name. The method `scopeDefining:` forwards the search to the current scope, obtained through `currentScope`. Since we only have one scope for now we do not cover the case where the variable is not in the variables of the receiver.

```
CInstanceScope >> scopeDefining: aString
  (self definedVariables includes: aString)
    ifTrue: [ ^ self ].
  Error signal: 'Variable ', aString, ' not found'
```

Now we can simply define the method scopeDefining: within the interpreter. It just delegates to the current scope.

```
CInterpreter >> scopeDefining: aName
  ^ self currentScope scopeDefining: aName
```

We should redefine the method `visitVariableNode:` to handle variables that are different from `self` and `super`.

```
CInterpreter >> visitVariableNode: aVariableNode

  ^ aVariableNode name = #self | (aVariableNode name = #super)
    ifTrue: [ self receiver ]
    ifFalse: [ (self scopeDefining: aVariableNode name) read: aVariableNode
     name ]
```

All our tests should now pass!

7.5 Refactor: Improving our Tests `setUp`

Let's simplify how receivers are managed in tests. Instead of having to explicitly create and manage a receiver object, we add the receiver as an instance variable to the `CInterpreterTest`.

```
TestCase << #CInterpreterTest
  slots: { #interpreter . #receiver};
  package: 'Champollion'
```

We can then redefine the `setUp` method to create a new instance of `CHInterpretable` and assign it to the variable `receiver`.

```
CInterpreterTest >> setUp
  super setUp.
  receiver := CInterpretable new
```

And now we use this new instance variable in `executeSelector:` as the default receiver instead of `nil`.

```
CInterpreterTest >> executeSelector: aSymbol
  ^ self executeSelector: aSymbol withReceiver: receiver
```

And finally we rewrite all our test methods using an explicit receiver:

```
CInterpreterTest >> testReturnSelf
  self assert: (self executeSelector: #returnSelf) == receiver

CHInterpreterTest >> testReturnSuper
  self assert: (self executeSelector: #returnSelf) == receiver

CInterpreterTest >> testReturnInstanceVariableRead
  receiver x: 100.
  self
    assert: (self executeSelector: #returnX)
    equals: 100
```

7.6 Instance Variable Writes

We have support for instance variable reads. Let us work on instance variable writes. In our scenario, a method writes a literal integer into an instance variable x and then returns the value of the assignment. This scenario has two observable behaviors that we will be tested separately.

First, such an assignment should be observable from the outside by reading that variable. Second, an assignment is an expression whose value is the assigned value, thus the return value should also be the assigned value.

```
CInterpretable >> store100IntoInstanceVariableX
  ^ x := 100
```

To test this behavior, we evaluate the method above and then we validate that effectively the instance variable was mutated. To make sure the value was modified, we set an initial value to the variable before the evaluation. After the evaluation, we should not keep that value. The case of the return is similar to our previous tests.

```
CInterpreterTest >> testStore100IntoInstanceVariableX
  receiver x: 17.
  self executeSelector: #store100IntoInstanceVariableX.
  self assert: receiver x equals: 100

CInterpreterTest >> testAssignmentReturnsAssignedValue
  self
    assert: (self executeSelector: #store100IntoInstanceVariableX)
```

```
    equals: 100
```

We should extend the instance scope to support the modification of a variable. This is what we do using the method `instVarNamed:put:`.

```
CInstanceScope >> write: aString withValue: anInteger
  receiver instVarNamed: aString put: anInteger
```

Finally, we extend the interpreter with the method `visitAssignmentNode:`. To make these tests pass, let's first see in detail our method `visitAssignmentNode`.

```
CHInterpreter >> visitAssignmentNode: anAssignmentNode
  | value |
  value := self visitNode: anAssignmentNode value.
  (self scopeDefining: anAssignmentNode variable name)
    write: anAssignmentNode variable name
    withValue: value.
  ^ value
```

Evaluating an assignment node with the form `variable := expression` requires that we evaluate the `expression` of the assignment, also called the right-hand side of the assignment, and then set that value to the left-side variable.

Our visit method then looks as follows:

- We recursively visit the right side of the assignment (got through the `value` accessor). Note that the expression can be complex such as the result of multiple messages or other assignments.
- We set that value to the variable by delegating to the instance scope (message `write:withValue:`), and
- We finally return the stored value.

Now the tests should pass.

7.7 Evaluating Variables: Global Reads

We finish this chapter with the reading of global variables, which covers two cases: proper global variables and access to classes. It illustrates the chain of scopes where an instance scope parent is a global scope.

To better control our testing environment, we decided to not use the Pharo environment by default. Instead, the interpreter will know its global scope in an instance variable and look up globals in it using a simple API, making it possible to use the system's global environment instead if we wanted to. We also leave outside of this chapter the writing to global variables. The case of global variable writing is similar to the instance variable assignment.

7.8 Little Setup

Our first testing scenario, similar to the previous ones, is as follows: we will define a method `returnGlobal` that reads and returns the global named `Global`. Now the question is that from the CInterpretable class the global variable `Global` should be a Pharo global variable. Note that when the interpreter encounters such a variable, it uses its own private environment.

So to make sure that you can compile the code and execute your tests, we define the class method initialize as follows:

```
CInterpretable class >> initialize
  self setUpGlobal

CInterpretable class >> setUpGlobal

  self environment at: #Global put: 33
```

Execute the method initialize doing `CInterpretable initialize`. It will be executed in the future when you load the package.

Such a detail is only necessary to keep Pharo itself happy and it does not have any impact in our implementation. Remember that we are writing our own Pharo implementation in the interpreter, and we will decide what to do with `Global` ourselves.

7.9 Implementing Global reads

Define the method `returnGlobal` as follows:

```
CInterpretable >> returnGlobal
  ^ Global
```

We start by enriching the class `CInterpreterTest` with an instance variable pointing to a new interpreter.

```
CInterpreterTest >> setUp

  super setUp.
  interpreter := CInterpreter new.
  receiver := CInterpretable new
```

This implies that we should modify the getter to be

```
CInterpreterTest >> interpreter

  ^ interpreter
```

We define a test that specifies that the interpreter's environment has a binding whose key is `#Global` and value is a new object. You see here that we parametrize

the interpreter so that it has a globalEnvironment that is different from the one its class.

```
CInterpreterTest >> testReturnGlobal
  | globalObject |
  globalObject := Object new.
  interpreter globalEnvironmentAt: #Global put: globalObject.
  self assert: (self executeSelector: #returnGlobal) equals: globalObject
```

We introduce a new global scope class `CGlobalScope`,A globalDictionary is basically a dictionary.

```
Object << #CGlobalScope
  slots: { #globalDictionary };
  package: 'Champollion'

CGlobalScope >> initialize
  super initialize.
  globalDictionary := Dictionary new

CGlobalScope >> globalDictionary: anObject
  globalDictionary := anObject

CGlobalScope >> at: aKey ifAbsent: aBlock
  ^ globalDictionary at: aKey ifAbsent: aBlock
```

We add an instance variable named `globalScope` in the class `CInterpreter` initialized to a global scope.

```
Object << #CInterpreter
  slots: { #receiver . #globalScope };
  package: 'Champollion2

CInterpreter >> initialize
  super initialize.
  globalScope := CGlobalScope new

CInterpreter >> globalEnvironment
  ^ globalScope
```

We define the method `globalEnvironmentAt:put:` to easily add new globals from our test and make sure that we can set the value of a global in the global scope.

```
CInterpreter >> globalEnvironmentAt: aSymbol put: anObject
  globalScope at: aSymbol put: anObject

CGlobalScope >> at: aKey put: aValue
  globalsDictionary at: aKey put: aValue
```

7.10 Introduce scopeDefining:

We define the methods `scopeDefining:` and a `read:` method in our global scope.

```
CGlobalScope >> scopeDefining: aString
  "I'm the root scope..."
  ^ self

CGlobalScope >> read: aString
  ^ globalDictionary at: aString
```

7.11 Supporting parentScope

We add support to represent parent scope to the class `CinstanceScope`. We add the instance variable `parentScope` to the class `CInstanceScope`.

```
Object <<#CInstanceScope
  slots: {#receiver . #parentScope};
  package: 'Champollion'

CInstanceScope >> parentScope: anObject
  parentScope := anObject

CInstanceScope >> parentScope
  ^ parentScope
```

We can define the scope chain by defining that an instance scope has (for now) a global scope as parent. In this method we see clearly that an instance scope lives in the context of a global one and we defined previously that the global scope is the final scope root (see Method 'CGlobalScope » #scopeDefining: above).

```
CInterpreter >> currentScope
  ^ CInstanceScope new
    receiver: self receiver;
    parentScope: globalScope;
    yourself
```

We redefine `scopeDefining:` so that if the variable is not defined in the instance variables of the instances, it looks in the parent scope.

```
CInstanceScope >> scopeDefining: aString
  (self definedVariables includes: aString)
    ifTrue: [ ^ self ].

  ^ self parentScope scopeDefining: aString
```

Now all the tests should be green.

7.12 Conclusion

In this chapter, we introduced support for variables and name resolution to support instance variables and global variables. This first evaluator serves as a basis for implementing the execution of the main operations in Pharo: messages.

The following chapters will start by decomposing message resolution into method-lookup and apply operations, introduce the execution stack, the management of temporary variables and argument, and 'super' message-sends.

We further invite the reader to explore changing the language semantics by modifying this simple evaluator: How could we implement read-only objects? Log all variable reads and writes?

CHAPTER 8

Implementing Message Sends: The Calling Infrastructure

In the previous chapters, we focused on structural evaluation: reading literal objects and reading and writing values from objects and globals. However, the key abstraction in object-oriented programming, and in Pharo in particular is *message-sending*. The work we did in the previous chapter is nevertheless important to set up the stage: we have a better taste of the visitor pattern, we started a first testing infrastructure, and eventually, message-sends need to carry out some work by using literal objects or reading and writing variables.

8.1 Message Concerns

Message-sends deserve a chapter on their own because they introduce many different concerns. On the one hand, each message-send is resolved in two steps:

- first the method-lookup searches in the receiver's hierarchy the method to be executed, and
- second that method is applied on the receiver (i.e., it is evaluated with self-bound to the receiver).

On the other hand, each method application needs to set up an execution context to store the receiver, arguments, and temporary variables for that specific method execution. These execution contexts form the *execution stack* or *call stack*. Sending a message pushes a new context in the call stack, and returning

from a method pops a context from the call stack. This are the mechanics that we will cover in this chapter so that in the following chapter we can implement logic and support late-binding.

8.2 Introduction to Stack Management

The way we managed the receiver so far is overly simplistic. Indeed, each time a program sends a message to another object, we should change the receiver and when a method execution ends, we should restore the previous receiver. Moreover, the same happens with method arguments and temporaries as we will see later. Therefore to introduce the notion of message-send we need a stack: each element in the stack needs to capture all the execution state required to come back to it later on when a message-send will return.

Each element in the call stack is usually named a *stack frame*, an activation record, or in Pharo's terminology a *context*. For the rest of this book we will refer to them as frames, for shortness, and to distinguish them from the reified contexts from Pharo.

Figure 8-1 presents a call stack with two methods. The first method in the stack (at its bottom) is method `foo`. Method `foo` calls method `bar` and thus it follows it in the stack. In addition, the message `foo` is sent to self so both frame points to the same object receiving the message. The current method executing is the one on the top of the stack. When a method returns, we can restore all the state of the previous method just by *popping* the top of the stack.

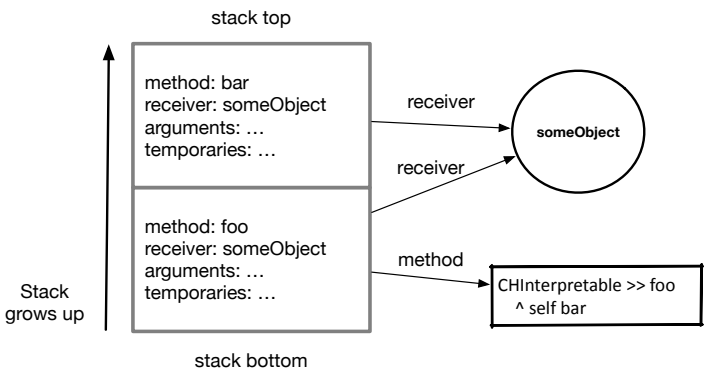

Figure 8-1 A call-stack with two frames, executing the method `foo` which sends the message `self bar`.

Listing 8-2 Replace the receiver instance variable by a stack

```
Object << #CInterpreter
  slots: { #stack . #globalScope };
  package: 'Champollion'

CInterpreter >> initialize
  super initialize.
  globalScope := CGlobalScope new.
  stack := CTStack new.
```

8.3 Method Scope

Since methods define a scope with their temporary variables and arguments, we represent frames using a new kind of scope: a method scope. For now, the method scope will store the current receiver, and later its parent scope, and a set of key-value pairs representing the variables defined in the current method execution: the arguments and temporaries (see Chapter 9).

```
Object << #CMethodScope
  slots: { #receiver };
  package: 'Champollion'

CMethodScope >> receiver: aCInterpretable
  receiver := aCInterpretable

CMethodScope >> receiver
  ^ receiver
```

8.4 Put in Place the Stack

We will use the stack implementation available at github://pharo-containers/.

```
Metacello new
  baseline: 'ContainersStack';
  repository: 'github://pharo-containers/Containers-Stack:v1.0/src';
  load.
```

A first step to introduce stack management without breaking all our previous tests is to replace the single `receiver` instance variable with a stack that will be initialized when the evaluator is created. The top of the stack will represent the current execution, and thus we will take the current receiver at each moment from the stack top. Moreover, each time we tell our interpreter to execute something we need to initialize our stack with a single frame.

With this new schema, we can now rewrite the access to the receiver to just access the value of `#self` of the top frame.

```
CInterpreter >> topFrame
  ^ stack top

CInterpreter >> receiver
  ^ self topFrame receiver
```

The final step is to set up a frame when the execution starts, which happened so far in our method `execute:withReceiver:`. We redefine the `execute:withReceiver:` to create a new frame and define the receiver as `#self` in the top frames before start the evaluation.

```
CInterpreter >> execute: anAST withReceiver: anObject
  self pushNewMethodFrame.
  self topFrame receiver: anObject.
  ^ self visitNode: anAST
```

The last piece in the puzzle is the method `pushNewMethodFrame`, which creates a new frame and pushes it on the top of the stack.

```
CInterpreter >> pushNewMethodFrame
  | newTop |
  newTop := CMethodScope new.
  stack push: newTop.
  ^ newTop
```

This refactor kept all the test green, and opened the path to introduce message-sends. As the reader may have observed, this stack can only grow. We will take care of popping frames from the stack later when we revisit method returns.

SD: currentScope is not connected to the stack and this is strange. Because it is defined as.

```
currentScope
  ^ CInstanceScope new
    receiver: self receiver;
    parentScope: globalScope;
    yourself
```

Note that we will have to improve the situation because the method `currentScope` (which is creating an instance scope) is not connected with the frame and this will be a problem.

8.5 A First Message Send Evaluation

Let's start as usual by defining a new method exhibiting the scenario we want to work on. In this case, we want to start by extending our evaluator to correctly evaluate return values of message sends.

Our scenario method `sendMessageReturnX` does a self-send message and returns the value returned by this message. The scenario stresses two points:

- On the one hand, the receiver of both messages is the same.

- On the other hand, the message is correctly evaluated as a return of the value of the activated method.

```
CInterpretable >> sendMessageReturnX
  ^ self returnInstanceVariableX
```

Notice that our method `sendMessageReturnX` and `returnInstanceVariableX` are defined in the same class. This means that in this first scenario, we can concentrate on the stack management and return value of the message sends, without caring too much about the details of the method lookup algorithm. For this first version, we will define a simple and incomplete yet useful method lookup algorithm.

Let us define a test named `testSelfSend`

```
CInterpreterTest >> testSelfSend
  receiver x: 100.
  self
    assert: (self executeSelector: #sendMessageReturnX)
    equals: 100
```

In this test we want to ensure that in a `self` message-send, the receiver of both the called and callee methods is the same. One way to do that is with a side-effect: if we write the instance variable `x` in one method and we access that value from the other method, we should get the same value for `x`. This will show that the object represented by `self` is the same and that we did not push for example `nil`.

Conceptually evaluating a message node requires recursively evaluating the receiver node, which may be a literal node or a complex expression such as another message-send. From such an evaluation we obtain the actual receiver object. Starting from the receiver, we should look up the method with the same selector as the message-send and execute the found method and return the result.

To make this test green, we implement the method `visitMessageNode:`.

In the first implementation proposed hereafter, we just fetch the desired method's AST from the receiver's class. Finally, we can activate this method with the receiver using `execute:withReceiver:` the activation will push a new frame to the call-stack with the given receiver, evaluate the method, and eventually return with a value.

```
CInterpreter >> visitMessageNode: aMessageNode
  | newReceiver method ast |
  newReceiver := self visitNode: aMessageNode receiver.
  method := newReceiver class compiledMethodAt: aMessageNode selector.
  ast := OCParser parseMethod: method sourceCode.
  ^ self execute: ast withReceiver: newReceiver
```

All our tests should pass and in particular `testSelfSend`.

8.6 AST Access Logic Consolidation

Pharo provides a way to get an AST from a compiled method, but we do not want to use because the AST it returns is different from the one we want for this book (the variables are resolved based on a semantical analysis). This is why we use `OCParser parseMethod: method sourceCode.`

To encapsulate such a decision we define the method `astOf:` and use it. In addition we take the opportunity to set the class from which the method AST is originating.

```
astOf: aCompiledMethod
  | ast |
  ast := OCParser parseMethod: aCompiledMethod sourceCode.
  ast methodClass: aCompiledMethod methodClass.
  ^ ast
```

```
CInterpreter >> visitMessageNode: aMessageNode
  | newReceiver method |
  newReceiver := self visitNode: aMessageNode receiver.
  method := newReceiver class compiledMethodAt: aMessageNode selector.
  ^ self execute: (self astOf: method) withReceiver: newReceiver
```

8.7 Balance the Stack

We mentioned earlier that when the execution of a method is finished and the execution returns to its caller method, its frame should be also discarded from the stack. The current implementation clearly does not do it. Indeed, we also said that our initial implementation of the stack only grows: it is clear by reading our code that we never pop frames from the stack.

To solve this issue, let us write a test showing the problem first. The idea of this test is that upon return, the frame of the caller method should be restored and with its receiver. If we make that the caller and callee methods have different receiver instances, then this test can be expressed by calling some other method, ignoring its value and then returning something that depends only on the receiver. In other words, this test will fail if calling a method on some other object modifies the caller!

The following code snippet shows a scenario that fulfills these requirements:

- it sets an instance variable with some value,
- sends a message to an object other than `self` and
- upon its return, it accesses its instance variable again before returning it.

Assuming the collaborator object does not modify `self`, then the result of evaluating this message should be that 1000 is returned. When the interpret will

return the value of x it will look for the receiver in the stack frame and should return the one holding 1000 and not the collaborator.

```
CInterpretable >> setXAndMessage
    x := 1000.
    collaborator returnInstanceVariableX.
    ^ x
```

Our test `testBalancingStack` executes the message `setXAndMessage` that should return 1000.

```
CInterpreterTest >> testBalancingStack
  self
    assert: (self executeSelector: #setXAndMessage)
    equals: 1000
```

We then finish our setup by extending `CInterpretable` to support delegating to a collaborator object. We add a `collaborator` instance variable to the class `CInterpretable` with its companion accessors. This way we will be able to test that the correct object is set and passed around in the example.

```
Object << #CInterpretable
  slots: { #x . #collaborator };
  package: 'Champollion'

CInterpretable >> collaborator
  ^ collaborator

CInterpretable >> collaborator: anObject
  collaborator := anObject
```

And in the `setUp` method we pass a collaborator to our initial receiver.

```
CInterpreterTest >> setUp

  super setUp.
  interpreter := CInterpreter new.
  receiver := CInterpretable new.
  receiver collaborator: CInterpretable new
```

Make the Test Pass

Executing this test breaks because the access to the instance variable x returns nil, showing the limits of our current implementation. This is due to the fact that evaluating message send `returnInstanceVariableX` creates a new frame with the collaborator as receiver, and since that frame is not popped from of the stack, when the method returns, the access to the x instance variable accesses the one of the uninitialized collaborator instead of the caller object.

To solve this problem, we should pop the frame when the activation method finishes. This way the stack is balanced. This is what the new implementation of `execute:withReceiver:` does.

```
CInterpreter >> execute: anAST withReceiver: anObject
  | result |
  self pushNewMethodFrame.
  self topFrame receiver: anObject.
  result := self visitNode: anAST.
  self popFrame.
  ^ result

CInterpreter >> popFrame
  stack pop
```

8.8 Extra Test for Receiver

Our previous tests ensure that messages return the correct value, activate the correct methods, and that the stack grows and shrinks. We, however, did not ensure yet that the receiver changes correctly on a message send, and since we do not lose any opportunity to strengthen our trust in our implementation with a new test, let's write a test for it.

The scenario, illustrated in `changeCollaboratorInstanceVariableX`, will ask the collaborator to `store100IntoX`, implemented previously. In this scenario, we must ensure that the state of the receiver and the collaborator are indeed separate and that changing the collaborator will not affect the initial receiver's state.

```
CInterpretable >> changeCollaboratorInstanceVariableX
  collaborator store100IntoInstanceVariableX
```

Our test for this scenario is as follows: If we give some value to the receiver and collaborator, executing our method should change the collaborator but not the initial receiver.

```
CInterpreterTest >>
    testInstanceVariableStoreInMethodActivationDoesNotChangeSender
  receiver x: 200.
  collaborator x: 300.

  "changeCollaboratorInstanceVariableX will replace collaborator's x but not
     the receiver's"
  self executeSelector: #changeCollaboratorInstanceVariableX.

  self assert: receiver x equals: 200.
  self assert: collaborator x equals: 100
```

To make our test run, we will store as a convenience the collaborator object in an instance variable of the test too and modify the `setUp` method.

```
TestCase << #CInterpreterTest
  slots: { #receiver . #collaborator };
  package: 'Champollion'

CInterpreterTest >> setUp
```

```
super setUp.
interpreter := CInterpreter new.
receiver := CInterpretable new.
collaborator := CInterpretable new.
receiver collaborator: collaborator
```

This test passes, meaning that our implementation already covered correctly this case. We are ready to continue our journey in message-sends.

8.9 Conclusion

In this chapter, we set the infrastructure to support message execution. We introduced the important notion of stack frames whose elements represent a given execution. We perform a rudimentary method lookup: we just look in the class of the receiver. The interpreter supports simple messages sent between different objects of the same class.

CHAPTER 9

Message Arguments and Temporaries

In the previous chapter, we extended the interpreter to support simple messages sent between different instances of the same class. In this chapter, we will extend this message passing implementation to support parameters and temporaries. We will take advantage of the

9.1 Message Argument Evaluation

So far we have worked only with unary messages. Unary messages have no arguments, so the number of programs we can express with them only is limited. The next step towards having a full-blown interpreter is to support message arguments, which will open the door to support binary and keyword messages. From the evaluator's point of view, as well as from the AST point of view, we will not distinguish between unary, binary, and keyword messages. The parser already takes care of distinguishing them and handling their precedence. Indeed, message nodes in the AST are the same for all kinds of messages, they have a selector and a collection of argument nodes. Precedence is then modeled as relationships between the AST nodes.

In addition to simply passing arguments, the evaluator needs to care about evaluation order too. This is particularly important because Pharo is an imperative language where messages can trigger side effects. Evaluating two messages in one order may not have the same result as evaluating them in a different order. Arguments in Pharo are evaluated eagerly after evaluating

the receiver expression, but before evaluating the message, from left to right. Once all expressions are evaluated, the resulting objects are sent as part of the message-send.

Initial Argument Setup

To implement some initial support for arguments, our first scenario is to simply send a message with an argument. We already one message with an argument: the `x:` setter. We define the method `changeCollaboratorWithArgument` which uses it.

```
CInterpretable >> changeCollaboratorWithArgument
  collaborator x: 500
```

In the following test, we verify that the method evaluation effectively modifies the collaborator object as written in `changeCollaboratorWithArgument`, and not the initial receiver object.

```
CInterpreterTest >> testArgumentAccess

  receiver x: 200.
  collaborator x: 300.

  self executeSelector: #changeCollaboratorWithArgument.

  self assert: receiver x equals: 200.
  self assert: collaborator x equals: 500
```

Since we have not implemented any support for arguments yet, this test should fail.

9.2 Enhance Method Scope

The current method scope is limited to managing the receiver. It is not enough. Method scopes should support variables as well as parsent scope.

We add the `parentScope:` instance variable and its accessors as well as

```
Object << #CMethodScope
  slots: { #receiver . #parentScope . #variables };
  package: 'Champollion'
```

To support variables, we add a dictionary to hold them and some utility methods.

```
CMethodScope >>initialize

  super initialize.
  variables := Dictionary new.

CMethodScope >> at: aKey
```

```
  ^ variables at: aKey

CMethodScope >> at: aKey put: aValue
  variables at: aKey put: aValue
```

In addition, we add support for identifying the scope. We define the method `scopeDefining:` as follows: It checks whether the looked up name is a variable one.

```
CMethodScope >> scopeDefining: aString
  (variables includesKey: aString)
    ifTrue: [ ^ self ].

  ^ self parentScope scopeDefining: aString
```

Note that the `scopeDefining:` delegates to its parent scope when the variable is not locally found.

Finally we add a `read:` method using the variable implementation logic.

```
CMethodScope >> read: aString
  ^ variables at: aString
```

9.3 Support for Parameters/Arguments

Implementing argument support requires two main changes:

- On the caller side, we need to evaluate the arguments in the context of the caller method and then store those values in the new frame.
- On the callee side, when an argument access is evaluated, those accesses will not re-evaluate the expressions in the caller. Instead, argument access will just read the variables pre-stored in the current frame.

Let's start with the second step, the callee side, and since all variable reads are concentrated on the scope lookup, we need to add the method scope in the scope chain.

Previous Situation

Previously the method `execute:withReceiver:` was defined as follows:

```
execute: anAST withReceiver: anObject

  | result |
  self pushNewMethodFrame.
  self topFrame receiver: anObject.
  result := self visitNode: anAST.
  self popFrame.
  ^ result
```

Where `pushNewMethodFrame` was pushing a method scope instance on the stack.

```
pushNewMethodFrame
  | newTop |
  newTop := CMethodScope new.
  stack push: newTop.
  ^ newTop
```

Improved Version

The new version is the following one:

```
CInterpreter >> execute: anAST withReceiver: anObject
  | result |
  self pushNewMethodFrame.
  self topFrame parentScope: (CInstanceScope new
    receiver: anObject;
    parentScope: globalScope;
    yourself).

  self topFrame receiver: anObject.
  result := self visitNode: anAST.
  self popFrame.
  ^ result
```

After pushing to the stack a new frame representing the method execution, we should make sure that the parent scope of the method scope is an instance scope. This is what we were doing in the `currentScope` method.

Also notice that the instance scope and the method scope have both a receiver. SD: When can they be different? And why the instance one is not enough since it is in the parent scope of the method.

We have still to make sure that `currentScope` refers to the top frame of the interpreter. This is what the following redefinition expresses:

```
CInterpreter >> currentScope
  ^ self topFrame
```

Then we need to update `visitMessageNode:` to compute the arguments by doing a recursive evaluation, and then use those values during the new method activation.

```
CInterpreter >> visitMessageNode: aMessageNode
  | newReceiver method args |
  args := aMessageNode arguments collect: [ :each | self visitNode: each ].
  newReceiver := self visitNode: aMessageNode receiver.
  method := newReceiver class compiledMethodAt: aMessageNode selector.
  ^ self executeMethod: (self astOf: method) withReceiver: newReceiver
     andArguments: args
```

To include arguments in the method activation, we add a new `arguments` parameter to our method `execute:withReceiver:` to get `execute:withReceiver:withArguments:`.

In addition to adding the receiver to the new frame representing the execution, we add a binding for each parameter (called unfortunately arguments in Pharo AST) with their corresponding value in the argument collection. This binding is added to the variables of the top frame. The message `with:do:` iterates both the parameter list and actual arguments as pairs.

```
CInterpreter >> execute: anAST withReceiver: anObject andArguments: aCollection
  | result |
  self pushNewMethodFrame.
  self topFrame parentScope: (CInstanceScope new
    receiver: anObject;
    parentScope: globalScope;
    yourself).

  self topFrame receiver: anObject.
  anAST arguments
    with: aCollection
    do: [ :arg :value | self topFrame at: arg name put: value ].
  result := self visitNode: anAST.
  self popFrame.
  ^ result
```

Instead of just removing the old `executeMethod:withReceiver:` method, we redefine it calling the new one with a default empty collection of arguments. This method was used by our tests and is part of our public API, so keeping it will avoid migrating extra code and an empty collection of arguments is a sensible and practical default value.

```
CInterpreter >> executeMethod: anAST withReceiver: anObject
  ^ self
    executeMethod: anAST
    withReceiver: anObject
    andArguments: #()
```

Our tests should all pass now.

9.4 Refactoring the Terrain

Let's now refactor a bit the existing code to clean it up and expose some existing but hidden functionality. Let us extract the code that accesses `self` and the frame parameters into two other methods that make more intention revealing that we are accessing values in the current frame.

```
CInterpreter >> tempAt: aSymbol put: anInteger
  self topFrame at: aSymbol put: anInteger
```

```
CInterpreter >> execute: anAST withReceiver: anObject andArguments: aCollection
  | result |
  self pushNewMethodFrame.
  self topFrame parentScope: (CInstanceScope new
    receiver: anObject;
    parentScope: globalScope;
    yourself).

  self topFrame receiver: anObject.
  anAST arguments
    with: aCollection
    do: [ :arg :value | self tempAt: arg name put: value ].
  result := self visitNode: anAST.
  self popFrame.
  ^ result
```

9.5 Temporary Evaluation

Temporary variables, or local variables, are variables that live within the scope of a method's **execution**. Memory for such variables is allocated when a method is activated, and released when the method returns. Because of this property, temporary variables are also called automatic variables in languages like C.

The usual way to implement such temporary variables is to allocate them in the method execution's frame. This way, when the method returns, the frame is popped and all the values allocated in temporaries are discarded and can be reclaimed. In other words, we can manage temporaries the same way as we manage arguments.

Our first scenario introducing temporaries will verify the default value of temporaries. Indeed when temporaries are allocated in Pharo, the execution engine (in this case our evaluator) should make sure these variables are correctly initialized to a default value, in this case `nil`.

Notice that temporaries cannot be observed from outside the execution of a method unless we halt the evaluation of a method in the middle of the evaluation. Since our testing approach is more like a black-box approach, we need to make our scenarios visible from the outside. Because of these reasons, our tests will rely on returns again, as we did before with literal objects.

We define the method `returnUnassignedTemp` that simply returns a local variable that is just allocated.

```
CInterpretable >> returnUnassignedTemp
  | temp |
  ^ temp
```

The companion test verifies that the value of an uninitialized temporary is `nil`.

```
CInterpreterTest >> testUnassignedTempHasNilValue
  self
    assert: (self executeSelector: #returnUnassignedTemp)
    equals: nil
```

The current subset of Pharo that we interpret does not contain blocks and their local/temporary variables – We will implement blocks and more complex lexical scopes in a subsequent chapter. Therefore the temporary variable management we need to implement is simple.

To make our test pass, we modify the `execute:withReceiver:andArguments:` method to define the temporaries needed with `nil` as value.

```
CInterpreter >> executeMethod: anAST withReceiver: anObject andArguments:
     aCollection
  | result  |
  self pushNewMethodFrame.
  self topFrame parentScope: (CHInstanceScope new
    receiver: anObject;
    parentScope: globalScope;
    yourself);

  self topFrame receiver: anObject.
  anAST arguments with: aCollection do: [ :arg :value | self tempAt: arg name
     put: value ].
  anAST temporaryNames do: [ :tempName | self tempAt: tempName put: nil ].
  result := self visitNode: anAST body.
  self popFrame.
  ^ result
```

The tests should pass.

9.6 Another Refactoring

We take the opportunity to refactor a bit the method `execute:withReceiver:andArguments:`. We extract the temporary and argument management into a new method named `manageArgumentsTemps:of:`.

```
CInterpreter >> execute: anAST withReceiver: anObject andArguments: aCollection

  | result |
  self pushNewMethodFrame.
  self topFrame parentScope: (CInstanceScope new
       receiver: anObject;
       parentScope: globalScope;
       yourself).
  self topFrame receiver: anObject.
  self manageArgumentsTemps: aCollection of: anAST.
  result := self visitNode: anAST.
  self popFrame.
  ^ result
```

```
CInterpreter >> manageArgumentsTemps: aCollection of: anAST

  anAST arguments
    with: aCollection
    do: [ :arg :value | self tempAt: arg name put: value ].
  anAST temporaryNames do: [ :tempName |
    self tempAt: tempName put: nil ]
```

9.7 Temporary Variable Write Evaluation

The next aspect we have to address is temporary writing.

We test that writes to temporary variables are working too. We define our scenario method `writeTemporaryVariable`, which defines a temporary variable, assigns to it and returns it.

```
CInterpretable >> writeTemporaryVariable
  | temp |
  temp := 100.
  ^ temp
```

Remark that an optimizing compiler for this code would be smart enough to do constant propagation of the literal integer and then realize that the temporary is dead code and remove it, leaving us with a method body looking like `^ 100 .` However, since the parser does not do such optimizations by itself, we are sure that the AST we get contains both the temporary definition, the assignment, and the temporary return.

Its companion test checks that evaluating this method does effectively return 100, meaning that the temporary variable write succeeded, and that `temp` means the same variable in the assignment and in the access.

```
CInterpreterTest >> testWriteTemporaryVariable
  self
    assert: (self executeSelector: #writeTemporaryVariable)
    equals: 100
```

If you execute the test `testWriteTemporaryVariable` it should fail.

Since temporary variable name resolution is already managed by method scopes, we just need to implement `write:withValue:` in it to make all our tests pass.

```
CMethodScope >> write: aString withValue: aValue
  variables at: aString put: aValue
```

9.8 Evaluation Order

The last thing we need to make sure is that arguments are evaluated in the correct order. The evaluation order in Pharo goes as follows:

- Before evaluating a message, the receiver and all arguments are evaluated.
- The receiver is evaluated before the arguments.
- Arguments are evaluated in left-to-right order.

Testing the evaluation order in a black-box fashion, as we have been doing so far, is rather challenging with our current evaluator. Indeed, our evaluator does not yet handle arithmetics, allocations, or other kinds of primitive, so we cannot easily count! A simple approach to test is to make a counter out of Peano Axioms.

The main idea is to implement numbers as nested sets, where

- the empty set *(S())* represents the number `0`,
- the set *(S(S())* that contains the empty set (`0`) represents the number `1`,

the set *(S(S(S())))* that contains the number `1` represents the number `2`, and so on.

9.9 Peano Number Implementation

First we implement peano numbers. The code illustrating the idea follows.

First we add a counter named `currentPeanoNumber` to the interpreter and we initialize it with the peano number zero represented by an empty Set.

```
CInterpretable >> initialize
  super initialize.
  currentPeanoNumber := { } "Empty"
```

The method `nextPeanoNumber`

```
CInterpretable >> currentButComputeNextPeanoNumber
  | result |
  "Implement a stream as an increment in terms of Peano axioms.
  See https://en.wikipedia.org/wiki/Peano_axioms"
  result := currentPeanoNumber.
  "We increment the counter"
  currentPeanoNumber := { currentPeanoNumber }.
  "We return the previous result"
  ^ result
```

We define a simple method `peanoToInteger:` that converts a peano number into its corresponding integer number.

```
CInterpreterTest >> peanoToInteger: aPeanoNumber
  "Helper method to transform a peano number to a normal Pharo integer"
  ^ aPeanoNumber
    ifEmpty: [ 0 ]
    ifNotEmpty: [ 1 + (self peanoToInteger: aPeanoNumber first) ]
```

9.10 Using Peano Numbers

Using the peano encoding, we can express our evaluation order scenario and test as follows: We want to get information about the message receiver and each of its arguments: we will compute and store a peano number for each of these sitations.

Let us implement this now. The message receiving the arguments will receive as argument three generated peano values, that we will return as dynamic literal array. If the evaluation order is right, the evaluation order of the receiver should be 0, the evaluation of the first argument should be 1, and so on.

We add a new instance variable to `CInterpretable` to store the evaluation order of the receiver of a message.

```
Object << #CInterpretable
  slots: { #x . #collaborator . #currentPeanoNumber . #evaluationOrder };
  package: 'Champollion'
```

```
CInterpretable >> evaluationOrder
  ^ evaluationOrder
```

We define the method `evaluateReceiver` which will store the current peano number in the `evaluationOrder` instance variable. Notice that this method returns the receiver so that we can use it as receiver of a message with multiple arguments (to compute the peano number of arguments).

```
CInterpretable >> evaluateReceiver
  evaluationOrder := self currentButComputeNextPeanoNumber.
  ^ self
```

We then define a method `returnEvaluationOrder` as follows:

```
CInterpretable >> returnEvaluationOrder
  ^ self evaluateReceiver
    messageArg1: self currentButComputeNextPeanoNumber
    arg2: self currentButComputeNextPeanoNumber
    arg3: self currentButComputeNextPeanoNumber

CInterpretable >> messageArg1: arg1 arg2: arg2 arg3: arg3
  ^ {arg1 . arg2 . arg3}
```

This method invokes the receiver evaluation and computes the next peano number before passing it as arguments to the message.

To verify the evaluation order we define two simple tests: one checking that the receiver receives the peano number zero. And one that checks that we get an array of numbers 1, 2, and 3. This array indicates that the value of the first argument was executing before the second and that the second was executed before the third.

```
CInterpreterTest >> testEvaluationOrderOfReceiver
  self executeSelector: #returnEvaluationOrder.
  self assert: (self peanoToInteger: receiver evaluationOrder) equals: 0.

CInterpreterTest >> testEvaluationOrderOfArguments
  | argumentEvaluationOrder |
  argumentEvaluationOrder := self executeSelector: #returnEvaluationOrder.
  self
    assert: (argumentEvaluationOrder collect:
      [ :peano | self peanoToInteger: peano])
    equals: #(1 2 3)
```

To make this test green we need our interpreter to support dynamic arrays (The definition is similar to the one for literal arrays we already added).

```
CInterpreter >> visitArrayNode: anArrayNode
  ^ anArrayNode statements
    collect: [ :e | self visitNode: e ]
    as: Array
```

At this point our test will fail because the evaluation order is wrong! The receiver was evaluated 4th, after all arguments. This is solved by changing the order of evaluation in `visitMessageNode:` and executing `self visitNode: aMessageNode receiver` first.

```
CInterpreter >> visitMessageNode: aMessageNode
  | newReceiver method args |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each | self visitNode: each ].
  method := newReceiver class compiledMethodAt: aMessageNode selector.
  ^ self execute: (self astOf: method) withReceiver: newReceiver andArguments:
     args
```

9.11 About Name Conflict Resolution

Inside the scope of a method, statements have access to parameters, temporaries, instances and global variables. A name conflict appears when two variables that should be visible in a method share the same name. In a conflict scenario, the language developer needs to devise a resolution strategy for these problems, to avoid ambiguities.

For example, consider a method `m:` that has an argument named `integer` and defines a temporary variable also named `integer`. How should the values of that name be resolved? How are assignments resolved? A conflict resolution strategy provides a set of deterministic rules to answer these questions and let developers understand what their programs do in a non-ambiguous way.

A first simple strategy to avoid conflicts is preventing them at construction time. That is, the language should not allow developers to define variables if

they generate a name conflict. For example, a method should not be able to define a temporary variable with the same name as an instance variable of its class. Usually, these validations are done once at compile time, and programs that do not follow these rules are rejected.

Another strategy to solve this problem is to allow shadowing. That is, we give each variable in our program a priority, and then the actual variable to read or write is looked up using this priority system. Typically priorities in these schemas are modelled as lexical scopes. Lexical scoping divides a program into a hierarchy of scopes. Each scope defines variables and all but the top-level scope have a parent scope. For example, the top-level scope defines global variables, the class scope defines the instance variables, the method scope defines the parameters and temporaries. In this way, variable visibility can be defined in terms of a scope: the variables visible in a scope are those defined in the scope or in the parents of the scope. Moreover, scoping also gives a conflict resolution strategy: variables defined closer to the current scope in the scope hierarchy have more priority than those defined higher in the scope hierarchy.

SD: We should add something about the interpreter.

9.12 About Return

As a reader, you may wonder why we did not do anything for return expression and this is an interesting question. Up to now interpreting a return is just returning the value of the interpretation of the return expression. In fact, up until now, a method execution has a single path of execution: it means that the complete method body should be executed and that we did not introduce different condition control flow. This is when we will introduce block closure and conditional control flow and we will have to revisit the interpretation of return.

9.13 Conclusion

Supporting message sends and in particular method execution is the core of the computation in an object-oriented language and this is what this chapter covered.

Implementing messages implied modeling the call stack and keeping it balanced on method returns. We have seen that a call stack is made up of frames, each frame representing the activation of a method: it stores the method, receiver, arguments, and temporaries of the method that is executing. When a message takes place, the receiver and arguments are evaluated in order from left to right, a new frame is created and all values are stored in the frame.

CHAPTER 10

Late Binding and Method Lookup

Method lookup deserves a chapter on its own: it represents the core internal logic of late-binding. The method-lookup algorithm needs to support normal message-sends as well as 'super' message-sends. In this chapter, we will implement method lookup for messages sent to an object. Then we will present how we handle the case of messages sent to `super`.

So far we have concentrated on method evaluation and put aside method lookup. Our current solution fetches methods from the class of the receiver, without supporting inheritance. In this section, we address this problem and implement a proper method lookup algorithm.

10.1 Method Lookup Introduction

Sending a message is a two-step process as shown by Figure 10-1:

1. Method lookup: the method corresponding to the selector is looked up in the class of the receiver and its superclasses.
2. Method execution: the method is applied to the receiver. This means that `self` or `this` in the method will be bound to the receiver.

Conceptually, sending a message can be described by the following function composition:

```
sending a message (receiver argument)
   return apply (lookup (selector classof(receiver) receiver) receiver
     arguments)
```

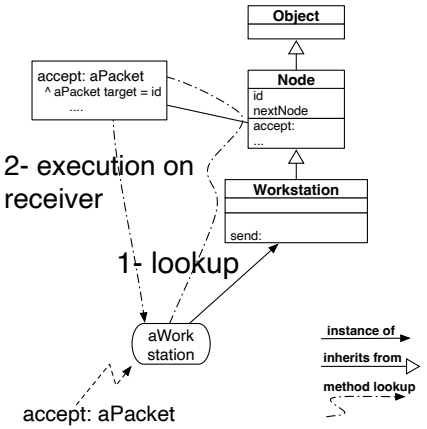

Figure 10-1 Sending a message is a two-step process: method lookup and execution.

Method lookup

Now the lookup process is conceptually defined as follows:

1. The lookup starts in the **class** of the **receiver**.
2. If the method is defined in that class (i.e., if the method is defined in the method dictionary), it is returned.
3. Otherwise the search continues in the superclass of the currently explored class.
4. If no method is found and there is no superclass to explore (if we are in the class `Object`), this is an error (i.e., the method is not defined).

The method lookup walks through the inheritance graph one class at a time using the superclass relationship. Here is a possible description of the lookup algorithm that will be used for both instance and class methods.

```
lookup (selector class receiver):
   if the method is found in class
      then return it
      else if class == Object
           then Error
           else lookup (selector superclass(class) receiver)
```

Let us implement method lookup.

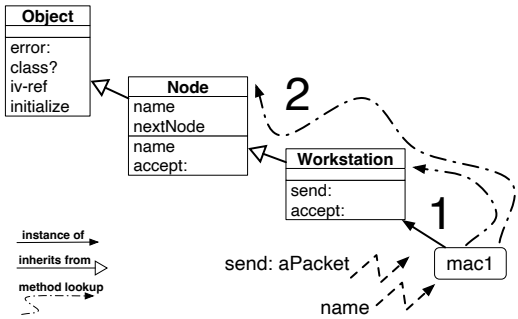

Figure 10-2 Looking for a method is a two-step process: first, go to the class of receiver then follow inheritance.

10.2 Method Lookup Scenario

To implement and test the method lookup, we should extend our scenario classes with a class hierarchy. We introduce two superclasses above `CInterpretable`: `CInterpretableRoot` and its subclass `CInterpretableSuperclass`. With this setup, we can test all interesting situations, even the ones leading to infinite loops if our method lookup is wrongly implemented.

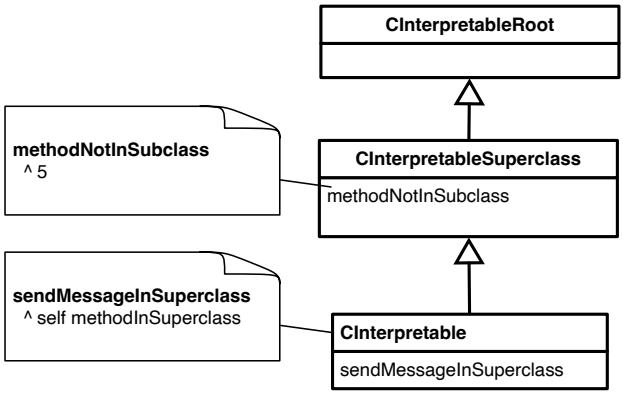

Figure 10-3 A simple hierarchy for self-send lookup testing.

```
Object << #CInterpretableRoot
  package: 'Champollion'
```

```
CInterpretableRoot << #CInterpretableSuperclass
  package: 'Champollion'
```

```
CInterpretableSuperclass << #CInterpretable
  slots: { #x . #collaborator .  #currentPeanoNumber . #evaluationOrder };
  package: 'Champollion'
```

Our first scenario for method lookup will check that sending a message climbs up the inheritance tree when a method is not found in the receiver's class. In the code below, we define a method in `CInterpretable` that does a `self` message whose method is implemented in its superclass (`CInterpretableSuperclass`). Executing the first method should send the message, find the superclass method, and evaluate it.

```
CInterpretableSuperclass >> methodInSuperclass
  ^ 5

CInterpretable >> sendMessageInSuperclass
  ^ self methodInSuperclass

CInterpreterTest >> testLookupMessageInSuperclass
  self assert: (self executeSelector: #sendMessageInSuperclass) equals: 5
```

10.3 A First Lookup

The test should fail with our evaluator as the evaluation of the message will not find the method in the receiver's class. A first step is to refactor the method `visitMessageNode:` and extract the wrong code into a `lookup:fromClass:` method. We also take the opportunity to extract the management of arguments.

```
CInterpreter >> visitMessageNode: aMessageNode
  | newReceiver method args |
  newReceiver := self visitNode: aMessageNode receiver.
  args := self handleArgumentsOf: aMessageNode arguments.
  method := self lookup: aMessageNode selector fromClass: newReceiver class.
  ^ self execute: (self astOf: method) withReceiver: newReceiver andArguments:
     args

CInterpreter >> lookup: aSelector fromClass: aClass
  ^ aClass compiledMethodAt: aSelector

CInterpreter >> handleArgumentsOf: aMessageNode
  ^ aMessageNode arguments collect: [ :each | self visitNode: each ]
```

The method `lookup:fromClass:` is now the place to implement the method lookup algorithm:

- if the current class defines the method returns the corresponding compiled method;
- if the current class does not define the method and we are not on the top of the hierarchy, we recursively lookup in the class' superclass;

- else when we are on top of the hierarchy and the `lookup:fromClass:` returns nil to indicate that no method was found.

```
CInterpreter >> lookup: aSelector fromClass: aClass
  "Return a compiled method or nil if none is found"

  "If the class defines a method for the selector, returns the method"
  (aClass includesSelector: aSelector)
    ifTrue: [ ^ aClass compiledMethodAt: aSelector ].

  "Otherwise lookup recursively in the superclass.
  If we reach the end of the hierarchy return nil"
  ^ aClass superclass
    ifNil: [ nil ]
    ifNotNil: [ self lookup: aSelector fromClass: aClass superclass ]
```

The method `lookup:fromClass:` does not raise an error because this way the `visitMessageNode:` method will be able to send the `doesNotUnderstand:` message to the receiver, as we will see later in this chapter.

Our tests should pass.

10.4 The Case of Super

Many people get confused by the semantics of `super`. The `super` variable has two different roles in the execution of an object-oriented language. When the `super` variable is read, its value is the *receiver* of the message as we saw it in the first chapter, it has the same value as `self`.

The second role of the `super` variable is to alter the method lookup when `super` is used as the receiver of the message send. When `super` is used as the receiver of a message send, the method lookup does *not* start from the class of the receiver, but from the class where the method is installed instead, allowing it to go up higher and higher in the hierarchy.

Let us introduce a new scenario for our tests. We define two methods named `isInSuperclass` and a method `doesSuperLookupFromSuperclass` as shown below (See Figure 10-4).

It is not nice since it uses `super` when it is unnecessary, but this is for a good cause. The handling of overridden messages will present better tests.

```
CInterpretableSuperclass >> isInSuperclass
  ^ true

CInterpretable >> isInSuperclass
  ^ false

CInterpretable >> doesSuperLookupFromSuperclass
  ^ super isInSuperclass
```

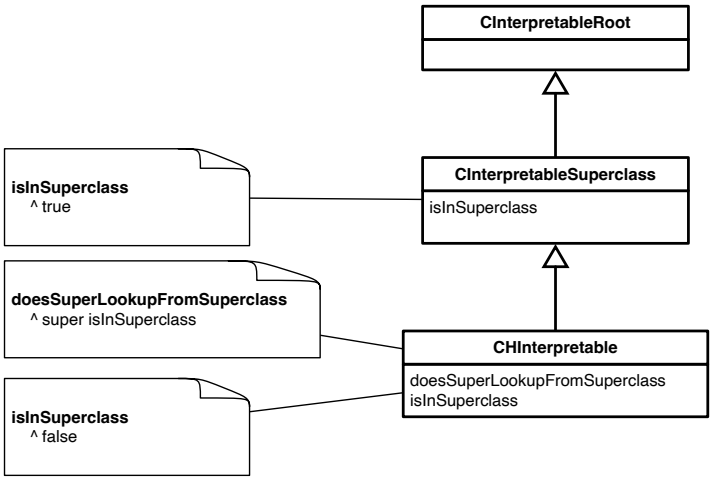

Figure 10-4 A simple hierarchy for super-send lookup testing.

Once these methods are defined, we can test that the `isInSuperclass` message activates the method in the superclass, returning `true`.

```
CInterpreterTest >> testLookupSuperMessage
  self assert: (self executeSelector: #doesSuperLookupFromSuperclass)
```

The `super` variable changes the method lookup described previously. When the receiver is `super`, the lookup does not start from the class of the receiver, but from *the superclass of the class defining the method*. This implies that we need a way to access the method that is being currently executed, and the class where it is defined.

We can store this information in the current frame during the method's activation. We add it for now as a fake temporary variable in the frame, with the name **`_method`. By prefixing the variable's name with** _ we make it less probable this fake variable creates a conflict with a real variable. If we would have just named it e.g., `method`, any method with a normal normal temporary called `method` would be broken.

```
CInterpreter >> executeMethod: anAST withReceiver: anObject andArguments:
     aCollection
  | result |
  self pushNewFrame.
  self tempAt: #___method put: anAST.
  self tempAt: #self put: anObject.
  self manageArgumentsTemps: aCollection of: anAST.
  result := self visitNode: anAST body.
  self popFrame.
  ^ result
```

We also define a convenience accessor method `currentMethod`, to get the current method stored in the current frame as well as the `tempAt:` method and its companion method in the class `CMethodScope`. In the future, if we want to change this implementation, we will have less places to change if we hide the access to the method behind an accessor.

```
CMethodScope >> at: aKey
  ^ variables at: aKey

CInterpreter >> tempAt: aSymbol
  ^ self topFrame at: aSymbol

CInterpreter >> currentMethod
  ^ self tempAt: #___method
```

Note that using the current frame to store the current method will work, even if we have several messages in sequence. When a message is sent a new frame is pushed with a new method, and on return the frame is popped along with its method. So the top frame always contains the method it executes.

Finally, we redefine the `visitMessageNode:` method to change the class where to start looking for the method.

```
CInterpreter >> visitMessageNode: aMessageNode

  | newReceiver method args lookupClass pragma |
  newReceiver := self visitNode: aMessageNode receiver.
  args := self handleArgumentsOf: aMessageNode arguments.

  lookupClass := aMessageNode isSuperSend
    ifTrue: [ self currentMethod methodClass superclass ]
    ifFalse: [ newReceiver class ].
  method := self lookup: aMessageNode selector fromClass: lookupClass.
  ^ self executeMethod: method withReceiver: newReceiver andArguments: args
```

With this last change, your tests should now all pass.

10.5 Overridden Messages

We have made sure that sending a message to `super` starts looking at methods in the superclass of the class defining the method. Now we would like to make sure that the lookup works even in the presence of overridden methods.

Let's define the method `overriddenMethod` in a superclass returning a value, and in a subclass just doing a super send with the same selector.

```
CInterpretableSuperClass >> overriddenMethod
  ^ 5

CInterpretable >> overriddenMethod
  ^ super overriddenMethod
```

If our implementation is correct, sending the message `overriddenMethod` to our test receiver should return `5`. If it is not, the test should fail, or worse, loop infinitely.

Then we check that our test returns the correct value. If the test loops infinitely the test will timeout.

```
CInterpreterTest >> testLookupRedefinedMethod
  self assert: (self executeSelector: #overriddenMethod) equals: 5
```

This test should pass.

10.6 Conclusion

In this chapter we extended the interpreter to implement method lookup. We took in particular the case of `super`. In the following chapter, we will present how to support the case where the looked up method is not found.

CHAPTER 11

Handling Unknown Messages

In the previous chapter, we presented method lookup and showed the precise semantics of the messages sent to `super`. We only took into account the case where the method we are looking up actually exists. In this chapter, we show how to handle this case. We extend the interpreter with support for the support error and the famous `doesNotUnderstand:`.

We start by revisiting the current method lookup implementation. Doing so, we will be ready to handle the case of unknown messages.

11.1 Correct Semantics Verification

To ensure that the method lookup is correctly implemented, especially in the presence of `super` messages, we need to stress our implementation with an extra scenario. Several books wrongly define that `super` messages lookup methods starting from the superclass of the class of the receiver. This is plain wrong.

This definition, illustrated in the code snippet below, is incorrect: it only works when the inheritance depth is limited to two classes, a class, and its superclass. In other cases, this definition creates an infinite loop.

```
CInterpreter >> visitMessageNode: aMessageNode

  | newReceiver method args lookupClass pragma |
  newReceiver := self visitNode: aMessageNode receiver.
  args := self handleArgumentsOf: aMessageNode arguments.

  lookupClass := aMessageNode isSuperSend
    ifTrue: [ newReceiver class superclass ]
    ifFalse: [ newReceiver class ].
```

```
  method := self lookup: aMessageNode selector fromClass: lookupClass.
  ^ self executeMethod: method withReceiver: newReceiver andArguments: args
```

A scenario showing such a problem is shown in Figure 11-1. In this scenario, our inheritance depth is of three classes and we create two methods with the same selector. In the highest class, the method returns a value. In the middle class, the first method is overridden doing a super send.

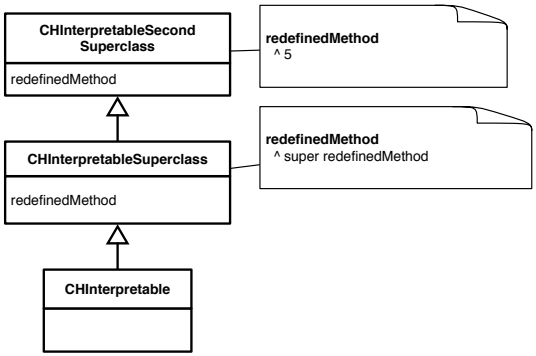

Figure 11-1 A simple situation that breaks wrongly defined super semantics: sending the message `redefinedMethod` to an instance of the class `CInterpretable` loops forever.

Let us define the situation that will loop with the wrong semantics.

```
CInterpretableRoot >> redefinedMethod
  ^ 5

CInterpretableSuperClass >> redefinedMethod
  ^ super redefinedMethod
```

To finish our scenario, we create an instance of the lower subclass in the hierarchy, and we send it a message with the offending selector.

```
CInterpreterTest >> testLookupSuperMessageNotInReceiverSuperclass
  self assert: (self executeSelector: #redefinedMethod) equals: 5
```

Before executing our new test.

With the incorrect semantics, our test will start by activating `CInterpretableSuperclass>>#redefinedMethod`.

When the interpreter finds the super send, it will start the lookup from the superclass of the receiver's class: `CInterpretableSuperclass`. Starting the lookup from this class will again find and activate `CInterpretableSuper-`

`class>>#redefinedMethod`, which will lead to activating the same method over and over again...

Coming back to our previous correct definition, it works properly, and makes our test pass:

```
CInterpreter >> visitMessageNode: aMessageNode

  | newReceiver method args lookupClass pragma |
  newReceiver := self visitNode: aMessageNode receiver.
  args := self handleArgumentsOf: aMessageNode arguments.

  lookupClass := aMessageNode isSuperSend
    ifTrue: [ self currentMethod methodClass superclass ]
    ifFalse: [ newReceiver class ].
  method := self lookup: aMessageNode selector fromClass: lookupClass.
  ^ self executeMethod: method withReceiver: newReceiver andArguments: args
```

11.2 Make the Test Pass

The test `testLookupSuperMessageNotInReceiverSuperclass` does not pass because it fails before being able to execute the method. Indeed, the method `executeSelector:withReceiver:` makes the strong assumption that the executed method is defined in the class `CInterpretable` and this clearly not always the case.

```
executeSelector: aSymbol withReceiver: aReceiver
  | ast |
  ast := OCParser parseMethod: (CInterpretable >> aSymbol) sourceCode.
  ast methodClass: CInterpretable.
  ^ self interpreter execute: ast withReceiver: aReceiver
```

When we analyze the problem we see that a method lookup phase is missing. Starting from the receiver class, we should look for the correct compiled method. Here the method `redefinedMethod` is defined in the superclas sof `Cinterpretable`. To address this limit, we introduce the following method in the interpreter: It first looks for the method in the class of the receive then executes the method.

```
CInterpreter >> send: aSelector
  receiver: newReceiver
  lookupFromClass: lookupClass
  arguments: arguments

  | method |
  method := self lookup: aSelector fromClass: lookupClass.
  ^ self
    execute: (self astOf: method)
    withReceiver: newReceiver
    andArguments: arguments
```

And we use it in the test method infrastructure. It is good because it removes the duplication of logic around getting the AST and its associated class.

```
CInterpreterTest >> executeSelector: aSymbol withReceiver: aReceive
  ^ self interpreter
    send: aSymbol
    receiver: aReceiver
    lookupFromClass: aReceiver class
    arguments: #()
```

With this change most of our tests should pass.

The following is failing, and this is obvious because there is no `returnSuper` in the class Object.

```
testReturnSuper

  receiver := Object new.
  "Convey our intention of checking identity by using an explicit identity
     check"
  self assert: (self
      executeSelector: #returnSuper
      withReceiver: receiver) == receiver
```

We update it as follows

```
testReturnSuper

  receiver := CInterpretable new.
  "Convey our intention of checking identity by using an explicit identity
     check"
  self assert: (self
      executeSelector: #returnSuper
      withReceiver: receiver) == receiver
```

Now all our tests pass.

We can refactor a bit more and make `visitMessageNode:` use the new message `send:receiver:lookupFromClass:arguments:` as follows:

```
CInterpreter >> visitMessageNode: aMessageNode

  | newReceiver args lookupClass |
  newReceiver := self visitNode: aMessageNode receiver.
  args := aMessageNode arguments collect: [ :each |
        self visitNode: each ].

  lookupClass := aMessageNode receiver isSuperVariable
          ifTrue: [ self currentMethod methodClass superclass ]
          ifFalse: [ newReceiver class ].
  ^ self
    send: aMessageNode selector
    receiver: newReceiver
    lookupFromClass: lookupClass
    arguments: args asArray
```

The astute reader should think that we are not done. Indeed we can ask ourselves about the situation where the lookup does not find the method to execute.

This is what we will see now.

11.3 Unknown Messages

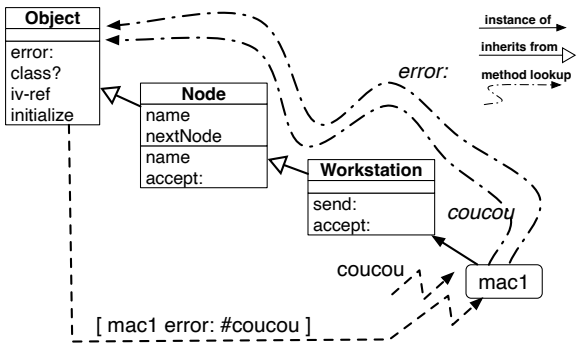

Figure 11-2 When a message is not found, another message is sent to the receiver supporting reflective operation.

11.4 Handling unknown messages

When the method is not found, the message `error` is sent as shown in Figure 11-2. Sending a message instead of simply reporting an error using a trace or an exception is a key design decision. In Pharo, this is done via the `doesNotUnderstand:` message, and it is an important reflective hook. Indeed classes can define their own implementation of the method `error` and perform specific actions to the case of messages that are not understood. For example, it is possible to implement proxies (objects representing other remote objects) or compile code on the fly by redefining such a message locally.

Here is a sketch of the lookup algorithm with error handling.

```
lookup (selector class):
   if the method is found in class
      then return it
      else if class == Object
           then return nil
           else lookup (selector superclass(class))
```

And then we redefine sending a message as follows:

```
sending a message (receiver argument)
   methodOrNil = lookup (selector classof(receiver)).
   if methodOrNil is nil
      then send the message error to the receiver
      else return apply(methodOrNil receiver arguments)
```

Before extending the interpreter to support unknow messages we should discuss

11.5 Does not understand and Message Reification

In Pharo, when an object receives a message for which the lookup does not find a corresponding method, it sends instead the `doesNotUnderstand:` message to that object, with the ”original message” as an argument. This original message is not only the selector but it comprises the arguments too.

The interpreter should take a selector and arguments to create an object representation of the message. We say the interpreter reifies the message.

About Reification.

Reification is the process of making concrete something that was not. In the case of the interpreter of a programming language, many of the operations of the language are implicit and hidden in the interpreter’s execution. For example, the implementation of message-sends and assignments are hidden to the developer in the sense that the developer cannot manipulate assignments for example to count the number of times an assignment has been used during program execution. While information hiding in interpreters is important to make languages safe and sound, the language has no way to manipulate those abstractions. Reifications enter the game to enable those manipulations: interpreter concepts are concretized as objects in the interpreted language, they are ”lifted up” from the interpreter level to the application.

Reifications are a powerful concept that allow us to manipulate implementation concerns from the language itself. In this case, the does not understand mechanism allows us to intercept the failing message-lookup algorithm and to implement in our program a strategy to handle the error. There exist in Pharo many different reifications such as classes and methods. In the scope of interpreters, we will see in the chapters that follow other kinds of reification: context objects representing execution frames.

A word is to be said about the performance implications of reifications. Reifications add levels of indirection to the execution. In addition, it allocates objects and this adds a significant overhead in the interpretation and increases the pressure in the garbage collector. Production interpreters try to minimize this cost to delay reifications as much as possible, and avoid them when they are

not necessary. This is what we will do with message reifications: we will create them when a method-lookup effectively fails and not before, penalizing only the execution of does not understand messages.

11.6 Implementing doesNotUnderstand:

To implement the does not understand feature, let's start by setting up our testing scenario: a method sending a not understood `messageIDoNotUnderstandWithArg1:withArg:2` message.

This message should be looked-up and not found, so the interpreter should send a `doesNotUnderstand:` message to the same receiver with the message reification.

For the message reification, we are going to follow Pharo's behavior and expect an instance of `Message` that should have the selector and an array with all the arguments.

The simplest implementation for the `doesNotUnderstand:` method is to simply return its argument. Notice that we define it on the class `CInterpretable`. Indeed every class can reimplement the way it handles message not understood by defining such a method.

```
CInterpretable >> doesNotUnderstand: aMessage
  ^ aMessage
```

To put in place our test scenario we define a new method sending an unknown message and a couple of tests.

```
CInterpretable >> sendMessageNotUnderstood
  ^ self messageIDoNotUnderstandWithArg1: 17 withArg2: 27
```

We define two tests covering that the implementation captures the message information.

```
CInterpreterTest >> testDoesNotUnderstandReifiesMessageWithSelector
  self
    assert: (self executeSelector: #sendMessageNotUnderstood) selector
    equals: #messageIDoNotUnderstandWithArg1:withArg2:

CInterpreterTest >> testDoesNotUnderstandReifiesMessageWithArguments
  self
    assert: (self executeSelector: #sendMessageNotUnderstood) arguments
    equals: #( 17 27 )
```

These two tests will fail in the interpreter, because the method lookup will return `nil`, which will fail during method activation. To address it, we need to handle this problem and send the `doesNotUnderstand:` message, as we said before. For this we modify the method `send:receiver:lookupFromClass:arguments:` as follows:

- it looks for the method
- if the method is not found, it creates a message, and send the message `doesNotUnderstand:` to the receiver with the message as an argument,
- else it just executes the found method.

```
CInterpreter >> send: aSelector receiver: newReceiver lookupFromClass:
     lookupClass arguments: arguments [

  | method |
  method := self lookup: aSelector fromClass: lookupClass.
  method ifNil: [ | messageReification |
    "Handle does not understand:
     - lookup the #doesNotUnderstand: selector
     - reify the message
     - activate"
    messageReification := Message
      selector: aSelector
      arguments: arguments.
    ^ self
      send: #doesNotUnderstand:
      newReceiver: receiver
      lookupFromClass: lookupClass
      arguments: { messageReification } ].

  ^ self execute: method withReceiver: newReceiver andArguments: arguments
```

Note that reifying does not understand requires that our interpreter knows two new things about our language: what selector is used for report the error (`#doesNotUnderstand:`), and what class is used to reify `Message`. In this case we are implementing a Pharo evaluator that runs in the same environment as the evaluated program: they share the same memory, classes, global variables. Because of this we make use of the existing selector and classes in Pharo. In contrast, implementing an evaluator that runs on a different environment than the evaluated program (e.g., a Pharo evaluator implemented in C), such dependencies need to be made explicit through a clear language-interpreter interface. This is for this reason that the Pharo virtual machine needs to know the selector of the message to be sent in case of message not understood.

All the tests should now pass.

11.7 Conclusion

In this chapter, we have shown how the `doesNotUnderstand:` feature is implemented, by handling the lookup error, and we introduced the concept of reification to concretize and lift up the failing message from our evaluator to the language.

CHAPTER 12

Primitive Operations

Our interpreter does not handle yet any essential behavior such as basic number operations. This prevents us from evaluating complex programs. This chapter introduces the concept of primitive methods as done in Pharo and extends the interpreter to evaluate them.

We will study how primitive methods work in Pharo, and how they should be properly implemented, including the evaluation of their fallback code (i.e., what happens when a primitive fails). We will then visit some of the essential primitives we need to make work to execute complex examples. This opens the door to the evaluation of more interesting programs.

12.1 The need for Primitives in Pharo

We call primitive behavior, behavior that needs to be implemented in the interpreter or evaluator because it cannot be purely expressed in the programming language, Pharo in this case.

Let's consider, for example, the operation of adding up two numbers (+). We cannot express in a pure and concise way a normal method of executing an addition.

Along with arithmetics, other examples of primitive operations are object allocation and reflective object access. Such a primitive behavior is expressed as special methods, namely **primitive methods** in Pharo, whose behavior is defined in the virtual machine.

In addition to essential behavior, primitive behavior is often used to implement performance-critical code in a much more efficient way, since primitives are

implemented in the implementation language and do not suffer the interpretation overhead.

12.2 Pharo Primitive Study

Differently, from languages such as Java or C, which express arithmetics as special operators that are compiled/interpreted differently, Pharo maintains the message send metaphor for primitive behavior. Indeed, in Pharo, + is a message that triggers a method look-up and a method activation.

This separation makes redefining operators as simple as implementing a method with the selector + in our own class, without the need for special syntax for it. The primitive should just be tagged with an id so that the Virtual machine finds its definition and executes it.

In Pharo the design of primitives is split in three different parts: *messages*, *primitive methods* (the Pharo method that is annotated), and the *primitive* itself which is provided by the interpreter - in the case of our interpreter this is a method that implements the primitive behavior. In the case of a Virtual Machine, it is a C function.

12.3 Study: Primitives invoked as Messages

The first thing to note is that in Pharo programs, primitive behavior is invoked through standard message sends. Indeed sending the message `1 + 2` is handled as a message, but the method + on `Integer` is a primitive (during Pharo execution it calls primitive functions transparently from the developer).

This management of primitives as messages allows developers to define operators such as + as non-primitives on their own classes, by just implementing methods with the corresponding selectors and without any additional syntax.

With some terminology abuse we could think of this as "operator redefinition", although it is no such, it is just a standard method re/definition. This operator redefinition feature is useful for example when creating internal Domain Specific Languages (DSLs), or to have polymorphism between integers and floats.

This is why in Pharo it is possible to define a new method + on any class as follows:

```
MyClass >> + anArgument
  "Redefine +"
  ...
```

12.4 Study: Primitive annotation

To define primitive behavior, Pharo relies on special methods called *primitive methods*: Primitive methods are normal Pharo methods with a `primitive` annotation. This annotation identifies that the method is special.

For example, let us consider the method `SmallInteger>>+` method below:

```
SmallInteger >> + aNumber
  <primitive: 1>
  ^ super + aNumber
```

This method looks like a normal method with selector `+` and with a normal method body doing `^ super + aNumber`. The only difference between this method and a normal one is that this method also has an annotation, or pragma, indicating that it is the primitive number 1.

12.5 Study: Interpreter primitives

Before diving into how *primitive methods* are executed, let us introduce the third component in place: the *interpreter primitives*. A primitive is a piece of code (another method) defined in the interpreter that will be executed when a primitive method is executed.

To make a parallel between our interpreter and the Pharo virtual machine: the virtual machine executes a C-function when a primitive method is executed.

The virtual machine interpreter defines a set of supported primitives with unique ids. We will mimic this behavior and in our interpreter, the primitive with id 1 implements the behavior that adds up two integers.

When a primitive method is activated, the body of the method is normally not executed. Instead the primitive 1 is executed. The method body is only executed, if the primitive failed.

More concretely, when a primitive method is activated,

- it first looks up what *primitive* to execute based on its primitive id number, and executes it.
- The primitive performs some validations if required, executes the corresponding behavior, and returns either with a success if everything went ok, or a failure if there was a problem.
- On success, the method execution returns with the value computed by the primitive.
- On failure, the body of the method is executed instead. Because of this, the body of a primitive method is also named the ”fall-back code”.

Note that in Pharo some primitives called essential such as the object allocation cannot be executed from Pharo. For such primitive, implementors added a method body to describe what the primitive is doing if it could be written in Pharo.

12.6 Infrastructure for Primitive Evaluation

To implement primitives in our evaluator we only need to change how methods are activated. Indeed, as we have seen above, neither a special method lookup nor dedicated nodes are required for the execution of primitives, and the AST already supports pragma nodes, from which we need to extract the method's primitive id.

Let's start by setting up our testing scenario: adding up two numbers. We make sure to work with small enough numbers in this test, to not care about primitive failures yet. Doing `1 + 5` the primitive should always be a success and return 6.

```
CInterpretable >> smallintAdd
  ^ 1 + 5

CInterpreterTests >> testSmallIntAddPrimitive
  self
    assert: (self executeSelector: #smallintAdd)
    equals: 6
```

12.7 Primitive Table Addition

We extend the method evaluation in a simple way: During the activation of a primitive method, we need to look for the primitive to execute and check for failures. Therefore we need to map primitive ids to primitive methods.

We implement such a mapping using a table with the form `<id, evaluator_selector>`. The `primitives` instance variable is initialized to a dictionary as follows:

```
CInterpreter >> initialize
  super initialize.
  stack := Stack new.
  primitives := Dictionary new.
  self initializePrimitiveTable.
```

Then we define the method `initializePrimitiveTable` to initialize the mapping between the primitive id and the Pharo method to be executed.

```
CInterpreter >> initializePrimitiveTable
  primitives at: 1 put: #primitiveSmallIntegerAdd
```

We define the primitive `primitiveSmallIntegerAdd` as follows:

```
CInterpreter >> primitiveSmallIntegerAdd
  | receiver argument |
  receiver := self receiver.
  argument := self argumentAt: 1.
  ^ receiver + argument
```

We introduce a way to access the value of an argument with the method `argumentAt:`.

```
CInterpreter >> argumentAt: anInteger
  ^ self tempAt: (self currentMethod arguments at: anInteger) name
```

12.8 Primitive Implementation

In the first iteration, we do not care about optimizing our evaluator (for which we had already and we will have tons of opportunities). To have a simple implementation to work on, we execute the primitive after the method's frame creation, in the `visitMethodNode:` method. This way the primitive has a simple way to access the receiver and the arguments by reading the frame. We leave primitive failure management for the second iteration.

Upon primitive method execution, we extract the primitive id from the pragma, get the selector of that id from the table, and use the `perform:` method on the interpreter with that selector to execute the primitive.

```
CInterpreter >> executePrimitiveMethod: anAST
  | primitiveNumber |
  primitiveNumber := anAST pragmas
    detect: [ :each | each isPrimitive ]
    ifFound: [ :aPragmaPrimitive | aPragmaPrimitive arguments first value ]
    ifNone: [ self error: 'Not a primitive method' ].
  ^ self perform: (primitives at: primitiveNumber)
```

In addition, we specialize `visitMethodNode:` so that it executes primitives when needed. At this stage, we do not support primitive failures.

```
CInterpreter >> visitMethodNode: aMethodNode
  aMethodNode isPrimitive ifTrue: [
    "Do not handle primitive failures for now"
    ^ self executePrimitiveMethod: aMethodNode ].
  ^ self visitNode: aMethodNode body
```

Our new test pass.

12.9 Primitive Failure Preparation

Let's now consider what should happen when a primitive fails. For example, following Pharo's specification, primitive 1 fails when the receiver or the argu-

ment are not small integers, or whenever their sum overflows and does not fit into a small integer anymore.

To produce one of such failing cases, we can implement primitive 1 in our `CInterpretable` class, which should fail because the receiver should be a small integer. When it fails, the fallback code should execute.

We define two methods `failingPrimitive` and `callingFailingPrimitive` to support the test of failing primitive.

```
CInterpretable >> failingPrimitive
  <primitive: 1>
  ^ 'failure'

CInterpretable >> callingFailingPrimitive
  ^ self failingPrimitive
```

We define a test to check that on failure we get the result returned by the failing primitive.

```
CInterpreterTest >> testFailingPrimitive
  self
    assert: (self executeSelector: #callingFailingPrimitive)
    equals: 'failure'
```

12.10 Primitive Failure Implementation

To add primitive failures cleanly, we introduce them as exceptions.

We define a new subclass of `Exception` named `CPrimitiveFailed`

In the primitive `primitiveSmallIntegerAdd`, if we detect a failure condition, we raise a `CPrimitiveFailed` error. Note that this primitive implementation is incomplete since we should also test that the argument and the result are small integers as shown in the following sections.

```
CInterpreter >> primitiveSmallIntegerAdd
  | receiver argument |
  receiver := self receiver.
  receiver class = SmallInteger
    ifFalse: [ CPrimitiveFailed signal ].
  argument := self argumentAt: 1.
  ^ receiver + argument
```

We then need to modify the way we evaluate methods to handle `CPrimitiveFailed` exceptions and continue evaluating the body.

```
CInterpreter >> visitMethodNode: aMethodNode

  [ aMethodNode isPrimitive ifTrue: [
    ^ self executePrimitiveMethod: aMethodNode ]]
    on: CPrimitiveFailed do: [ :err |
```

```
      "Nothing, just continue with the method body" ].

  ^ self visitNode: aMethodNode body
```

With these changes, everything should work fine now.

12.11 Typical Primitive Failure Cases

For primitives to work properly, and for Pharo to be a safe language, primitives should properly do a series of checks. This is particularly important when the interpreter fully controls all other aspects of the language, such as the memory. In such cases, primitives, as well as the other parts of the evaluator, have full power over our objects, potentially producing memory corruption.

Among the basic checks that primitives should do, they should not only verify that arguments are of the primitive's expected type, as we have shown above. In addition, a general check is that the primitive was called with the right number of arguments.

This check is particularly important because developers may wrongly define primitives. If we don't properly check the arguments trying to access them could cause an interpreter failure, while the proper behavior should be to just fail the primitive and let the fallback code carry on the execution.

In the following sections, we implement a series of essential primitives taking care of typical failure cases. With such primitives, it will be possible to execute a large range of Pharo programs.

We define the method `numberOfArguments` as follows:

```
CInterpreter >> numberOfArguments

  ^ self currentMethod numArgs
```

We define a better implementation of the addition primitive with checks: We verify that the receiver, argument, and result are small integers.

```
CInterpreter >> primitiveSmallIntegerAdd
  | receiver argument result |
  self numberOfArguments = 1
    ifFalse: [ CPrimitiveFailed signal ].

  receiver := self receiver.
  receiver class = SmallInteger
    ifFalse: [ CPrimitiveFailed signal ].

  argument := self argumentAt: 1.
  argument class = SmallInteger
    ifFalse: [ CPrimitiveFailed signal ].

  result := receiver + argument.
```

```
  result class = SmallInteger
    ifFalse: [ CPrimitiveFailed signal ].
  ^ result
```

12.12 Stepping

The `visitMethodNode:` definition above can be confusing let us take a moment to analyse it again. To be clear we need to see the three pieces together: the first definition, the *interpreter* primitive implementation and the *Pharo* primitive implementation.

```
CInterpreter >> visitMethodNode: aMethodNode

  [ aMethodNode isPrimitive ifTrue: [
    ^ self executePrimitiveMethod: aMethodNode ]]
    on: CPrimitiveFailed do: [ :err |
      "Nothing, just continue with the method body" ].

  ^ self visitNode: aMethodNode body
```

```
CInterpreter >> primitiveSmallIntegerAdd
     ...
```

```
SmallInteger >> + aNumber
  <primitive: 1>
  ^ super + aNumber
```

The argument of the `visitMethodNode:` is a Pharo method. When such a method is a *Pharo* primitive (i.e., it has the pragma annotation) the condition is basically doing a special interpretation: it executes the corresponding *interpreter* method primitive that we implemented inside the class `CInterpreter`. When such an execution raises a `CPrimitiveFailed` exception, the interpretation continues with the interpretation of the *Pharo* primitive fallback code.

What we see is that there is a go and back between the *Pharo* primitive and the definition of the corresponding primitive in the interpreter. First the *Pharo* primitive pragma is used to identify the *interpreter* primitive, then the interpreter executes the interpreter primitive. This is only on failure of the such a logic (which in a runtime is implemented in C for example) that the interpreter should execute the fallback code of *Pharo* primitive.

At the moment since the interpreter does not support conditional and block execution, we cannot execute the primitive fallback code.

12.13 Conclusion

In this chapter we showed how primitive behavior is evaluated. In particular we showed how fallback code is executed in case the primitive fails. In the fol-

lowing chapter we will describe essential primitives. With such implementation we will be able to execute more realistic programs.

CHAPTER 13

Essential Primitives

In this chapter, we will focus on implementing more primitives. We will implement more mathematical primitives but also support for comparisons, and array allocation.

13.1 Essential Primitives: Arithmetic

The basic arithmetic primitives are small integer addition, subtraction, multiplication, and division. They all require a small integer receiver and a small integer argument, and that the result is also a small integer. Division in addition fails in case the argument is `0`.

The following code snippet illustrates integer addition and division. For space reasons, we do not include subtraction and multiplication, their implementation is similar to the one of addition.

```
CInterpreter >> initializePrimitiveTable
  ...
  primitives at: 1    put: #primitiveSmallIntegerAdd.
  primitives at: 2    put: #primitiveSmallIntegerMinus.
  primitives at: 9    put: #primitiveSmallIntegerMultiply.
  primitives at: 10  put: #primitiveSmallIntegerDivide.
  ...
```

Here is the definition of the primitive for integer division.

```
CInterpreter >> primitiveSmallIntegerDivide
  | receiver argument result |
  self numberOfArguments < 1
    ifTrue: [ CPrimitiveFailed signal ].
```

```
  receiver := self receiver.
  receiver class = SmallInteger
    ifFalse: [ CPrimitiveFailed signal ].

  argument := self argumentAt: 1.
  (argument class = SmallInteger
    and: [ argument ~= 0 ])
      ifFalse: [ CPrimitiveFailed signal ].

  result := receiver / argument.
  result class = SmallInteger
    ifFalse: [ CPrimitiveFailed signal ].
  ^ result
```

We let you define integer subtraction and multiplication.

13.2 Essential Primitives: Comparison

Comparison primitives span in two different sets.

- The first set contains the primitives implementing number comparisons such as less than or greater or equals than.
- The second set contains the primitives for object identity comparison: identity equals to and identity not equals to.

All number comparisons all require a small integer receiver, a small integer argument. Identity comparisons only require that the primitive receives an argument to compare to.

The following definitions illustrate the two kinds of methods with small integer less than and object idenity equality.

```
CInterpreter >> initializePrimitiveTable
  ...
  primitives at: 3  put: #primitiveSmallIntegerLessThan.
  primitives at: 4  put: #primitiveSmallIntegerGreaterThan.
  primitives at: 5  put: #primitiveSmallIntegerLessOrEqualsThan.
  primitives at: 6  put: #primitiveSmallIntegerGreaterOrEqualsThan.

  primitives at: 7  put: #primitiveSmallIntegerEqualsThan.
  primitives at: 8  put: #primitiveSmallIntegerNotEqualsThan.

  primitives at: 110  put: #primitiveIdentical.
  primitives at: 111  put: #primitiveNotIdentical.
  ...
```

The following primitives following the same patterns and are self-explanatory.

```
CInterpreter >> primitiveSmallIntegerLessThan
  | receiver argument result |
  self numberOfArguments < 1
    ifTrue: [ CPrimitiveFailed signal ].

  receiver := self receiver.
  receiver class = SmallInteger
    ifFalse: [ CPrimitiveFailed signal ].

  argument := self argumentAt: 1.
  argument class = SmallInteger
    ifFalse: [ CPrimitiveFailed signal ].

  ^ receiver < argument
```

Following the definition of `primitiveSmallIntegerLessThan`, implement the following primitives

- `primitiveSmallIntegerGreaterThan`,
- `primitiveSmallIntegerLessOrEqualsThan`, and
- `primitiveSmallIntegerGreaterOrEqualsThan`.

```
CInterpreter >> primitiveIdentical
  self numberOfArguments < 1
    ifTrue: [ CPrimitiveFailed signal ].

  ^ self receiver == (self argumentAt: 1)
```

Define some tests to ensure that your implementation is correct.

13.3 Essential Primitives: Array Manipulation

So far our interpreter is able to manipulate only objects with instance variables, but not objects with variable size such as arrays or their variants e.g., strings. Arrays are special objects whose state is accessed with primitives, usually in methods named `at:`, `at:put:`, and `size`. Array access primitives check that the receiver is of the right kind and that the index arguments are integers within the bounds of the array. The following definition illustrates `Array` access primitives for general Arrays, and Strings.

```
CInterpreter >> initializePrimitiveTable
  ...
  primitives at: 60   put: #primitiveAt.
  primitives at: 61   put: #primitiveAtPut.
  primitives at: 62   put: #primitiveSize.
  primitives at: 63   put: #primitiveStringAt.
  primitives at: 64   put: #primitiveStringAtPut.
  ...
```

The primitive `primitiveAt` verifies that the receiver is an object supporting the notion of size and in addition that the index is an integer in the range of the size of the receiver.

```
CInterpreter >> primitiveAt

  self numberOfArguments = 1
    ifFalse: [ CPrimitiveFailed signal ].

  self receiver class classLayout isVariable
    ifFalse: [ CPrimitiveFailed signal ].

  ((self argumentAt: 1) isKindOf: SmallInteger)
    ifFalse: [ CPrimitiveFailed signal ].

  "Bounds check"
  ((self argumentAt: 1) between: 1 and: self receiver size)
    ifFalse: [ CPrimitiveFailed signal ].

  ^ self receiver basicAt: (self argumentAt: 1)
```

Here is a simple test verifying that the implementation is correct.

```
CInterpretable >> at

  ^ #(11 22 33) at: 2
```

```
CInterpreterTest >> testAt
  self assert: (self executeSelector: #at) equals: 22
```

Note that at this point, since the interpreter is not able to execute conditional and blocks, it cannot interpret failure of the primitive at: because it contains conditionals.

The primitive `primitiveStringAt` verifies that the receiver is from a class whose elements are bytes. It uses the class format information using the method `isBytes`.

```
CInterpreter >> primitiveStringAt
  self numberOfArguments = 1
    ifFalse: [ CPrimitiveFailed signal ].

  self receiver class classLayout isBytes
    ifFalse: [ CPrimitiveFailed signal ].

  ((self argumentAt: 1) isKindOf: SmallInteger)
    ifFalse: [ CPrimitiveFailed signal ].
  "Bounds check"

  ((self argumentAt: 1) between: 1 and: self receiver size)
    ifFalse: [ CPrimitiveFailed signal ].

  ^ self receiver at: (self argumentAt: 1)
```

The primitive `primitiveSize` verifies that the receiver is an object that has the notion of size. It uses the layout of the class to do so.

```
CInterpreter >> primitiveSize
  self receiver class classLayout isVariable
    ifFalse: [ CPrimitiveFailed signal ].

  ^ self receiver basicSize
```

We define a simple test as follows:

```
CInterpretable >> atSize

  ^ #(11 22 33)
```

```
CInterpreterTest >> testAtSize
  self assert: (self executeSelector: #atSize) equals: 3
```

Now we implement the primitive that supports the modification of arrays.

```
CInterpreter >> primitiveAtPut

  self numberOfArguments = 2
    ifFalse: [ CPrimitiveFailed signal ].

  self receiver class classLayout isVariable
    ifFalse: [ CPrimitiveFailed signal ].

  ((self argumentAt: 1) isKindOf: SmallInteger)
    ifFalse: [ CPrimitiveFailed signal ].

  "Bounds check"
  ((self argumentAt: 1) between: 1 and: self receiver size)
    ifFalse: [ CPrimitiveFailed signal ].

  ^ self receiver basicAt: (self argumentAt: 1) put: (self argumentAt: 2)
```

We test it as follows:

```
CInterpretable >> atPut
  | ar |
  ar := #(11 22 33).
  ar at: 2 put: 44.
  ^ ar
```

```
CInterpreterTest >> testAtPut
  self assert: (self executeSelector: #atPut) equals: #(11 44 33)
```

13.4 Essential Primitives: Object Allocation

We will implement important primitives: the primitives for object allocation. Object allocation is implemented by primitives `new` and `new:`.

- The method `new` allocates a new object from a fixed-slot class.

- The method `new:` allocates a new object from a variable-slot class such as `Array`, using the number of slots specified as argument.

Both these primitives validate that the receiver are classes of the specified kinds. In addition `new:` does check that there is an argument, it is a positive small integer.

```
CInterpreter >> initializePrimitiveTable
  ...
  primitives at: 70   put: #primitiveBasicNew.
  primitives at: 71   put: #primitiveBasicNewVariable.
  ...
```

To interpret basic new we rely on the one of Pharo because our interpreter does not its own memory management.

```
CInterpreter >> primitiveBasicNew
  self receiver isClass
    ifFalse: [ CPrimitiveFailed signal ].
  ^ self receiver basicNew
```

We test it as follows:

```
CInterpretable >> newPoint

  ^ Point basicNew
```

Here we make sure that the Pharo Point class is accessible in the global scope of the interpreter.

```
CInterpreterTest >> testPointNew

  | p |
  self interpreter globalEnvironmentAt: #Point put: Point.
  p := (self executeSelector: #newPoint).
  self
    assert: p class
    equals: Point.
```

For `basicNew:` we follow the same approach as before. We validate that the class is a class supporting variable number of instance variables.

```
CInterpreter >> primitiveBasicNewVariable
  self numberOfArguments = 1
    ifFalse: [ CPrimitiveFailed signal ].

  self receiver isClass
    ifFalse: [ CPrimitiveFailed signal ].
  self receiver classLayout isVariable
    ifFalse: [ CPrimitiveFailed signal ].

  ((self argumentAt: 1) isKindOf: SmallInteger)
    ifFalse: [ CPrimitiveFailed signal ].
```

```
  ^ self receiver basicNew: (self argumentAt: 1)
```

The following test verifies that the primitive is working. We also expose Array as a global variable of the interpreter.

```
CInterpreterTest >> testArrayNew

  | p |
  self interpreter globalEnvironmentAt: #Array put: Array.
  p := (self executeSelector: #newArray).
  self
    assert: p class
    equals: Array.
  self assert: p size equals: 4
```

```
CInterpretable >> newArray

  ^ Array basicNew: 4
```

13.5 Conclusion

This chapter shows the implementation of multiple primitives behavior. The short list of essential primitives we presented are required to execute more interesting Pharo programs.

CHAPTER 14

Block Closures and Control Flow Statements

In this chapter we will extend our evaluator to manage block closures. Block closures, also named lexical closures, or just blocks in Pharo, are an important concept in most modern programming languages, including Pharo. A lexical closure is an anonymous function that captures its definition environment.

This chapter starts by explaining what blocks are and how they are evaluated. Block evaluation, being a core part of the language definition, is a service that is requested to the evaluator/interpreter through a primitive. We then dive into the lexical capture feature of blocks: when a block closure is created, it captures its defining context, namely its enclosing context (i.e., the visible variables that the block can see). This makes blocks able to read and write not only its own temporary variables but also all the variables accessible to its enclosing context and to maintain such a link even when passed around. Finally, we implement non-local returns: return instructions that return to the block *definition context* instead of the current one. Non-local returns are really important in Pharo since they are used to express early returns (the fact that the execution of a method can be stopped at a given point) a frequent language feature similar to `break` statements in other languages. Without non-local return it would difficult to quit the current execution.

14.1 Closures

Closures allow developers to abstract general algorithms from their particular details. For example, a sorting algorithm can be separated from its sorting

criteria by making the sorting criteria a block closure passed as argument to it. This allows developers to have the sorting algorithm defined and tested in a single place, and being able to reuse it with multiple criterion in different contexts.

In Pharo, blocks are *lexical* closures i.e., basically functions without a name that capture the environment in which they are defined. Lexical closures are at the center of the Pharo language, because Pharo leverages closures to define its *control-flow* instructions: conditionals, iterations, and early returns. This means that implementing block closures is enough to support all kind of control flow statements in Pharo. Moreover, Pharo libraries make usage of block closures to define library-specific control flow instructions, such as the `do:` and `select:` messages understood by collections. Pharo developers often use closures in the Domain Specific languages that they design. Developers are also encouraged to define their own control flow statements, to hide implementation details of their libraries from their users.

14.2 Representing a Block Closure

When a block expression is executed `[ 1+2 ]`, the instructions inside the block definition are not executed. Instead, a block object is created, containing those instructions. The execution of those instructions is delayed until we send the message `value` to the block object.

This means that from the evaluator point of view, the evaluation of the closure will be different from the evaluation of its execution. Evaluating a block node will return a block object, and the method `value` will require a primitive to request the interpreter the block's execution. This means that we need a way to represent a closure object in our evaluator, and that closure should store the code it is supposed to evaluate later when receiving the `value` message.

Let us define the class `CBlock` to represent a block. It has an instance variable `code` to hold the block's AST, instance of the `BlockNode` class. Notice that we do not use the existing `BlockClosure` class from Pharo, since this class is tied up with the Pharo bytecode.

For the sake of simplicity, we will not reconciliate bytecode and AST implementations, meaning that we need our own AST-based block implementation.

```
Object << #CBlock
  slots: { #code };
  package: 'Champollion-Core'
```

```
CBlock >> code: aBlockNode
  code := aBlockNode
```

```
CBlock >> code
  ^ code
```

Block Definition

When the interpreter encounters a block node, it creates a block object for it. We define the method `visitBlockNode:` as follows:

```
CInterpreter >> visitBlockNode: aBlockNode
  ^ CBlock new
    code: aBlockNode;
    yourself
```

We add a simple test to verify the correct definition of block objects.

```
CInterperter >> testBlockDefinition

  | bk |
  bk := (self executeSelector: #returnBlock).
  self
    assert: bk class
    equals: CBlock.

  self
    assert: bk code class
    equals: RBBlockNode
```

```
CInterpretable >> returnBlock
  ^ [ 1 . 5 ]
```

14.3 Block Execution

In Pharo, when a method does not have a return statement, it returns `self`. The compiler basically adds it during its compilation.

This is different for a block: a block without return statement implicitly returns the result of its last expression.

Let us write a testing scenario for this case: evaluating the following block should return `5` as it is its last expression.

```
CInterpretable >> returnBlockValue
  ^ [ 1 . 5 ] value
```

```
CInterpreterTest >> testBlockValueIsLastStatementValue
  self assert: (self executeSelector: #returnBlockValue) equals: 5
```

Closures are executed when they receive the message `value` or one of its variants such as value `value:`, `value:value:`... On the reception of such messages, their bodies should be executed. Theses messages are defined in Pharo as primitives as shown in the following:

```
BlockClosure >> value
  "Activate the receiver, creating a closure activation (MethodContext)
  whose closure is the receiver and whose caller is the sender of this
  message. Supply the copied values to the activation as its copied
  temps. Primitive. Essential."
  <primitive: 207>
  numArgs ~= 0 ifTrue:
    [self numArgsError: 0].
  ^self primitiveFailed
```

14.4 Block Execution Implementation

We follow the design of Pharo and we add a new primitive responsible for the block body execution. For this we define a method value on the CBlock and tag it as a primitive. Then we declare a new primitive in the interpreter table and finally we define a first version of the primitive corresponding to the value execution.

We define the method `value` on the class `CBlock` as a primitive number 207.

```
CBlock >> value
  <primitive: 207>
  "If the fallback code executes it means that block evaluation failed.
  Return nil for now in such case."
  ^ nil
```

We now need to implement the new primitive in the evaluator. A first version of it is to just visit the body of the block's code. Remember that primitives are executed in their own frame already, so the block's body will share the frame created for the primitive method.

```
CInterpreter >> initializePrimitiveTable
  ...
  primitives at: 207 put: #primitiveBlockValue.
  ...

CInterpreter >> primitiveBlockValue
  ^ self visitNode: self receiver code body
```

So far we implemented only a simple version of closures. We will extend it in the following sections.

14.5 Closure Temporaries

Our simplified closure implementation does not yet have support for closure temporaries. Indeed, a closure such as the following will fail with an interpreter failure because `temp` is not defined in the frame.

```
[ | temp | temp ] value
```

To solve this we need to declare all block temporaries when activating the block, as we did previously for methods. As a first attempt to make our test green, let's declare block temporaries once the block is activated:

```
CInterpreter >> primitiveBlockValue
  "Initialize all temporaries to nil"
  | blockCode |
  blockCode := self receiver code.
  blockCode temporaryNames do: [ :e | self tempAt: e put: nil ].
  ^ self visitNode: blockCode body
```

We are now able to execute the following expression

```
[ | a b |
  a := 1.
  b := 2.
  a + b ] value
```

Let us define the following test:

```
CInterpretable >> returnBlockWithVariableValue

  ^ [ | a b |
    a := 1.
    b := 2.
    a + b ] value
```

```
CInterpreterTest >> testBlockValueWithTemporariesValue
  self
    assert: (self executeSelector: #returnBlockWithVariableValue)
    equals: 3
```

14.6 Removing Logic Repetition

The handling of temporaries in `primitiveBlockValue` is very similary to a sequence of messages we wrote when activating a normal method in method `execute:withReceiver:andArguments:`. In particular in the `manageArgumentsTemps:of:` method.

```
CInterpreter >> primitiveBlockValue
  "Receiver is an instance of CBlock"

  | blockCode |
  blockCode := self receiver.
  blockCode code temporaryNames do: [ :e | self tempAt: e put: nil ].
  self receiver: blockCode definingContext receiver.
  ^ self visitNode: blockCode code body
```

```
CInterpreter >> execute: anAST withReceiver: anObject andArguments: aCollection

  ...
  self manageArgumentsTemps: aCollection of: anAST.
  ...
```

```
CInterpreter >>manageArgumentsTemps: aCollection of: anAST

  anAST arguments
    with: aCollection
    do: [ :arg :value | self tempAt: arg name put: value ].
  anAST temporaryNames do: [ :tempName |
    self tempAt: tempName put: nil ]
```

We solve this repetition by moving temporary initialization to the `visitSequenceNode:` method, since both method nodes and block nodes have sequence nodes inside them.

```
CInterpreter >> visitSequenceNode: aSequenceNode
  "Initialize all temporaries to nil"

  aSequenceNode temporaryNames do: [ :e | self tempAt: e put: nil ].

  "Visit all but the last statement without caring about the result"
  aSequenceNode statements allButLast
    do: [ :each | self visitNode: each ].
  "Return the result of visiting the last statement"
  ^ self visitNode: aSequenceNode statements last
```

We then rewrite `primitiveBlockValue` as follows:

```
CInterpreter >> primitiveBlockValue
  ^ self visitNode: self receiver code body
```

We remove the temporary management from `manageArgumentsTemps:of:` and rename it.

```
CInterpreter >>manageArguments: aCollection of: anAST

  anAST arguments
    with: aCollection
    do: [ :arg :value | self tempAt: arg name put: value ].
```

The resulting code is nicer and simpler. This is a clear indication that the refactoring was a good move.

14.7 Capturing the Defining Context

Stef Here

As we stated before, a closure is not just a function, it is a function that captures the context (set of variables that it can access) at the time of its definition. Block closures capture their *defining* context or enclosing context, i.e., the context in which they are created. Blocks are able to read and write their own temporary variables, but also all the variables accessible to its enclosing context such as a temporary variable accessible during the block definition. In this section, we evolve our closure execution infrastructure to support closure temporaries and to provide access to the enclosing environment.

The defining execution context gives the closure access to that context's receiver, arguments, and temporaries.

Pay attention, it is a common mistake to think that the captured context is the caller context, and not the defining context. In the example above the distinction is not done because the definition context was the caller one. However, as soon as we work on more complex scenarios, where blocks are sent as arguments of methods, or stored in temporary variables, this does not hold anymore.

14.8 self Capture

A first scenario to check that our block properly captures the defining context is to evaluate `self` inside a block. In our current design, the receiver specified in the block's frame is the block itself. Indeed, the expression `[ ... ] value` is a message send where the block is the message receiver and `value` is the message. However, the `self` variable should be bound to the instance of `CHInterpretable`.

```
CInterpretable >> readSelfInBlock
  ^ [ self ] value

CInterpreterTest >> testReadSelfInBlock
  self assert: (self executeSelector: #readSelfInBlock) equals: receiver
```

To make this test pass, we need to implement two different things in the evaluator.

- First we need to capture the defining context at block *definition* time in `visitBlockNode:`.
- Second we need to use *that* captured context to resolve variables.

14.9 Capture Implementation

Capturing the defining context is as simple as storing the current `topFrame` at the moment of the method creation.

We extend `CBlock` with a `definingContext` instance variable and corresponding accessors (omitted here after).

```
Object << #CBlock
  slots: { #code . #definingContext };
  package: 'Champollion'
```

Since a block is created when the block node is visited we extend the previous block creation to store the current context at this moment. Note that this is

this context that will be let block access to the temporaries and arguments it uses at the moment the block is created.

```
CInterpreter >> visitBlockNode: aRBBlockNode
  ^ CBlock new
    code: aRBBlockNode;
    definingContext: self topFrame;
    yourself
```

14.10 Accessing Captured Receiver

Resolving the block variables is a trickier case, as it can be resolved in many different ways. For now, we choose to set the correct values in the current frame upon block activation and shadow the possible ones that would be defined in the definition context.

The first variable we want to provide access to from a block is `self` which is the original receiver of the method *at the time the block was created*.

The following method is worth explaining

- First we grab the block itself. It is simple since the method `primitiveBlockValue` is executed during the evaluation of the message `value` sent to a block. Therefore `self receiver` returns the block currently executed.
- Second remember that `self` in a block refers to the receiver of the method at the time the block was created. So we need to set as receiver the receiver that we found in the context of the block creation. This is what `theBlock definingContext receiver` is returning.
- Finally we evaluate the block body.

```
CInterpreter >> primitiveBlockValue
  | theBlock |
  theBlock := self receiver.
  self receiver: theBlock definingContext receiver.
  ^ self visitNode: theBlock code body
```

```
CInterpreter >> receiver: aValue
  ^ self tempAt: #self put: aValue
```

Note that in the `primitiveBlockValue` we use the frame of message `value` execution. The evaluation of the block body uses this frame. When the evaluation is done such frame is simply popped as any other method executions (See `executeMethod:withReceiver:andArguments:`), therefore there are no worries to be made when we change the value of receiver.`receiver` is not a state of the interpreter but refers to the current frame.

Now that we can correctly resolve the receiver, instance variable reads and writes should work properly too. We leave it as an exercise for the reader to verify their correctness.

14.11 Looking up Temporaries in Lexical Contexts

A problem we have not solved yet involves the reads and writes of temporary variables that are not part of the current frame. This is the case when a block tries to access a temporary of a parent lexical scope, such as another surrounding scope, or the home method. The method `increaseEnclosingTemporary` is an example of such a situation: the block `[ temp := temp + 1 ]` will access during its execution the temporary variable that was defined outside of the block. Note that the execution of the block could happen in another method and still be block should be able to access the temporary variable `temp`.

Our next scenario checks that blocks can correctly read and write temporaries of their enclosing contexts. In our test, the enclosing environment creates a temporary. The block reads that value and increases it by one. When the block executes and returns, the value of its temporary should have been updated from 0 to 1.

```
CHInterpretable >> increaseEnclosingTemporary [
  | temp |
  temp := 0.
  [ temp := temp + 1 ] value.
  ^ temp
]

CHInterpreterTest >> testIncreaseEnclosingTemporary [
  self assert: (self executeSelector: #increaseEnclosingTemporary) equals: 1
]
```

Note should add some diagrams here

This scenario is resolved by implementing a temporary variable lookup in the block's *defining* context. Of course, a block could be defined inside another's block context, so our lookup needs to be lookup through the complete context chain. The lookup should stop when the current lookup context does not have a defining context i.e., it is a method and not a block.

To simplify temporary variable lookup we define first a helper method `lookupFrameDefiningTemporary:` that returns the frame in which a temporary is defined. This method returns a frame. It has to walk from a frame to its defining frame up to a method. However, so far the only object in our design knowing the defining frame is the block (via its instance variable `definingContext`), and we do not have any way to access a block from its frame.

One possibility is to store a block reference in its frame when it is activated, and then go from a frame to its block to its defining frame and continue the lookup. Another possibility, which we will implement, is to directly store the defining context in the frame when the block is activated.

```
CHInterpreter >> primitiveBlockValue [
  | theBlock |
  theBlock := self receiver.
  self receiver: (theBlock definingContext at: #self).
  self tempAt: #__definingContext put: theBlock definingContext.
  ^ self visitNode: theBlock code body
]
```

```
CHInterpreter >> lookupFrameDefiningTemporary: aName [
  | currentLookupFrame |
  currentLookupFrame := self topFrame.
  [ currentLookupFrame includesKey: aName ]
    whileFalse: [ currentLookupFrame := currentLookupFrame at:
     #__definingContext ].
  ^ currentLookupFrame
]
```

Note should add some diagrams here

Now we need to redefine temporary reads and writes. Temporary reads need to lookup the frame where the variable is defined and read the value from it. This is what what the method `visitTemporaryNode:` does.

```
CHInterpreter >> visitTemporaryNode: aTemporaryNode [
  | definingFrame |
  definingFrame := self lookupFrameDefiningTemporary: aTemporaryNode name.
  ^ definingFrame at: aTemporaryNode name
]
```

Temporary writes are similar to read. We need to lookup the frame where the variable is defined and write the value to it.

```
CHInterpreter >> visitAssignmentNode: aRBAssignmentNode [
  | rightSide |
  rightSide := self visitNode: aRBAssignmentNode value.
  aRBAssignmentNode variable variable isTempVariable
    ifTrue: [ | definingFrame |
      definingFrame := self
        lookupFrameDefiningTemporary: aRBAssignmentNode variable name.
      definingFrame at: aRBAssignmentNode variable name put: rightSide ]
    ifFalse: [ aRBAssignmentNode variable variable
          write: rightSide
          to: self receiver ].
  ^ rightSide
]
```

14.12 Block Non-Local Return

We have seen so far that blocks implicitly return the value of their last expression. For example the method `lastExpression` will return 43.

```
CHInterpretable >> lastExpression
  | tmp |
  tmp := 1.
  tmp := true ifTrue: [ tmp := 42. tmp := tmp + 1].
  ^ tmp
```

Now this is a complete different story when a block contains an explicit return statement. Return statements, instead, break the execution of the defining method, namely the home method, and return from it. For example, let's consider a method using `ifTrue:` to implement a guard which should stop the method execution if the guard fails:

```
CHInterpretable >> methodWithGuard
  true ifTrue: [ ^ nil ].
  ^ self doSomethingExpensive
```

Note put a figure here to show the stack, the blocks, their relationships.

When executing this method, the message `doSomethingExpensive` will never be executed. The execution of the method `methodWithGuard` will be stopped by the return statement in the block `[^ nil]`.

More precisely, the block is not activated by `methodWithGuard`. `methodWithGuard` executes the message `ifTrue:` which in turn activates the `[^ nil]`. Still, this block knows the context of `methodWithGuard` as its defining context. When the block executes, the return statement should not return `nil` to the `ifTrue:` context: it should return *from* `methodWithGuard` with the `nil` value, as if it was the return value of the method. Because of this, we call such return inside blocks "non-local returns", because they return from a non-local context, its home context.

The block may have been passed around, when the block executes a return statement, it will return from the method that created the block. We say that the execution quits the home context of the block (the context of the method that defined it).

To implement non-local returns, we will first start by defining a new helper method: `homeFrameOf:` that returns the home frame of a frame. The home frame is the frame that has a defining context. Note that the home frame of a normal method frame is itself.

```
CHInterpreter >> homeFrame [
  | currentLookupFrame |
  currentLookupFrame := self topFrame.
  [ currentLookupFrame includesKey: #__definingContext ]
    whileTrue: [ currentLookupFrame := currentLookupFrame at:
     #__definingContext ].
  ^ currentLookupFrame
]
```

Note add a diagram

A simple way to implement non-local returns in Pharo is by using exceptions: exceptions unwind automatically the call-stack, thus short-circuiting the execution of all methods automatically.

We define a new exception called `CHReturn`. It refers to the home frame and a value.

```
Error subclass: #CHReturn
  instanceVariableNames: 'value homeFrame'
  classVariableNames: ''
  package: 'Champollion-Core'
```

```
CHReturn >> homeFrame [
  ^ homeFrame
]
```

```
CHReturn >> homeFrame: aFrame [
  homeFrame := aFrame
]
```

```
CHReturn >> value [
  ^ value
]
```

```
CHReturn >> value: aValue [
  value := aValue
]
```

When we activate a method we then need to prepare ourselves to catch the exception indicating a return, and only manage it if the return is targetting the current method's context:

SD: we should explain more the `returnFrom homeFrame = thisFrame`

```
CHInterpreter >> execute: anAST withReceiver: anObject andArguments:
     aCollection [
  | result thisFrame |
  thisFrame := self pushNewFrame.

  self tempAt: #__method put: anAST.
  self tempAt: #self put: anObject.
  anAST arguments with: aCollection
    do: [ :arg :value | self tempAt: arg name put: value ].
```

```
  result := [ self visitNode: anAST ]
    on: CHReturn        "A return statement was executed"
    do: [ :return |
      return homeFrame = thisFrame
        ifTrue: [ return value ]
        ifFalse: [ return pass ] ].

  self popFrame.
  ^ result
]
```

When we visit a return we raise a return exception and we pass the context. SD: need more explanation.

```
CHInterpreter >> visitReturnNode: aReturnNode [
  CHReturn new
    value: (self visitNode: aReturnNode value);
    homeFrame: self homeFrame;
    signal
]
```

14.13 Conclusion

In this chapter we have extended our evaluator with block closures. Our block closure implementation required adding a kind of object to our runtime, `CHBlock`, to represent blocks containing some AST. Then we refined our evaluator to define a block evaluation primitive, and correctly set up the lexical context. Our lexical context implementation gives blocks access to the defining context's receiver and temporaries. We then shown a first implementation of non-local returns, using exceptions to unwind the stack.

CHAPTER 15

Todo

- make sure that methodScope does not have a separate field for receiver but use the frame in previous chapter.