Fun with Interpreters in Pharo

Stéphane Ducasse and Guillermo Polito

January 6, 2022

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

Contents

Illustrations

CHAPTER 1

AST

The cornerstone of an AST interpreter is ASTs, short for abstract syntax trees. 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. To illustrate it, consider the piece of Pharo code below:

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

This piece of code assigns into a variable named `variable` the result of sending the `,` message to a `'constant'` string, to the result of the message `self message`.

An AST organizes the code above as a tree containing nodes representing variables, assignments, strings, and message sends.

This chapter presents ASTs by looking at the existing AST implementation in Pharo, the RBAST. RBAST is the AST implementation used currently (Pharo 9.0) 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, RBASTs 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.

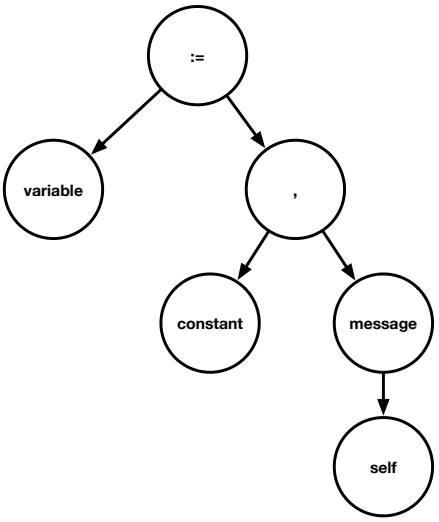

Figure 1-1 AST representing the code of `variable := 'constant' , self message`.

1.1 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 1-1 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 kind of nodes will be instances of diferent classes, forming a composite.

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 `RBParser`. The `RBParser` 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 delimieters (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.

1.2 Parsing 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 := RBParser 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 := RBParser 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.

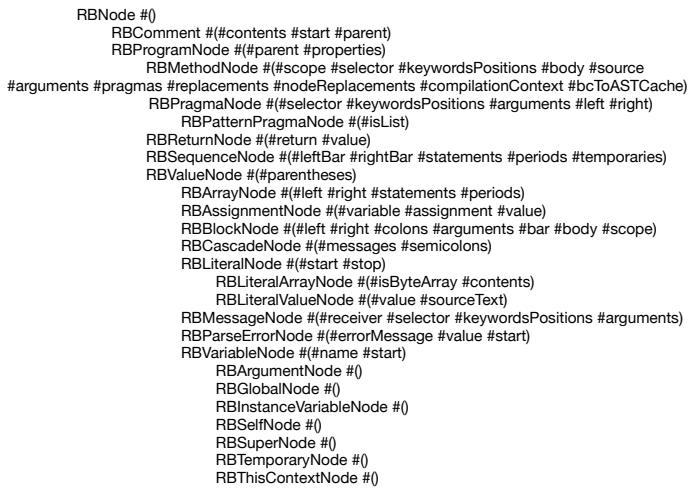

Figure 1-2 Overview of the method node hierarchy (TODO: remove RBPatternPragmaNode - Add RBSelectorNode). Indentation implies inheritance.

```
| expression |
expression := RBParser 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 hiearchy. Figure 1-2 shows the node inheritance hierarchy of Pharo rendered as a textual tree.

1.3 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 `RBLiteralValueNode`, 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 := RBParser parseExpression: '17'.
integerExpression value
>>> 17
```

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

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

A special case of literals are literal arrays, which have their own node: `RBLiteralArrayNode`. 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 := RBParser parseExpression: '#(1 2 3)'.
arrayExpression value
>>> #(1 2 3)

arrayExpression contents first
>>> RBLiteralValueNode(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 `RBVariableNode` and know their `name`.

```
variableExpression := RBParser 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 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 `RBVariableNode` objects. The only exception to this are `self`, `super` and `thisContext` special variables. Special variables are variables that are recognised at parse-time, and generating special nodes `RBSelfNode`, `RBSuperNode` and `RBThisContextNode` for them. These special nodes inherit from `RBVariableNode` and work as normal variable nodes for the purposes of this book.

1.4 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 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 `RBAssignmentNode`. 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 := RBParser parseExpression: 'var := #( 1 2 )
    size'.
assignmentExpression variable
>>> RBVariableNode(var)
```

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

1.5 Message Nodes

Message nodes are the core of Pharo programs, and they are the most common composed expression nodes we find in the AST. Messages are instances of `RBMessageNode` 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 := RBParser parseExpression: '17 max: 42'.
messageExpression receiver
>>> RBLiteralValueNode(17)
```

Note that `arguments` is a normal collection of expressions - in the sense that there is not special node class to represent such a sequence.

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

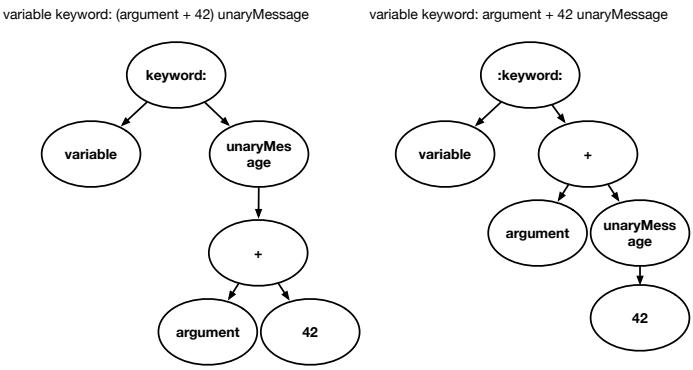

Figure 1-3 Different precedence results in different ASTs.

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 master the syntax of Pharo, you remember that there exist three kind 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. Precedence of messages in ASTs is resolved at parse-time. In other words, the output of `RBParser` is an AST respecting the precedence rules of Pharo.

Let's consider a couple of examples illustrating this, illustrated in Figure 1-3. If we feed the `RBParser` with the expression below, it will create a `RBMessageNode` 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 adding extra parenthesis as in:

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

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
```

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

1.6 Cascade Nodes

Cascade nodes represent cascaded message expressions, i.e., messages sent to the same receiver. Cascaded messages are messages separated by semicolons (`;`) 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 `RBCascadeNode`. 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 `RBMessageNode` 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 := RBParser parseExpression: 'var msg1; msg2'.
cascadeExpression receiver
>>> RBVariableNode(var)

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

1.7 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 `RBArrayNode`. To access the expressions inside a dynamic array node, they understand the message `children`

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

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

1.8 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 `RBMethodNode` and need to be parsed with a variant of the parser we have used so far, a method parser. The `RBParser` 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 a `RBMethodNode` instance for a method named `myMethod`.

```
methodNode := RBParser 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 programming languages in block 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
```

1.9 Sequence Nodes

Method nodes have a body, represented as a `RBSequenceNode`. 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 kind 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 := RBParser parseMethod: 'myMethod
  1+1.
  self'.

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

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

```
methodNode body statements.
>>> an OrderedCollection(RBMessageNode(1 + 1) RBSelfNode(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 := RBParser parseMethod: 'myMethod
  | temporary |
  1+1.
  self'.
methodNode body temporaries.
>>> an OrderedCollection(RBVariableNode(temporary))
```

1.10 Return Nodes

AST return nodes represent the instructions that are syntactically identified by the caret character ^. Return nodes, instances of `RBReturnNode` 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 we is accessible through the `value` message.

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

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

returnNode value.
>>>RBSelfNode(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 add semantics to the AST when we evaluate it in a subsequent chapter. In Pharo this is the compiler that ensures that a method always return 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 counter-part 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 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 := RBParser parseExpression: '[ :arg | | temp | 1 + 1.
    temp ]'.
blockNode arguments
>>>an OrderedCollection(RBVariableNode(arg))

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

blockNode body statements
>>>an OrderedCollection(RBMessageNode(1 + 1) RBVariableNode(temp))
```

1.11 Basic ASTs Manipulations

We have already covered all of Pharo AST nodes, and how to access the information in them. 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 post-card.

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 empowers 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.

Iterating over an AST

ASTs are indeed trees, and we can traverse them as any other tree. RBASTs 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 allChildren applying `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, expressions nodes parsed using `parseExpression:` are contained within a method node too.

Storing Properties

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 indicting 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, evaluates the block and returns its value.
- `aNode propertyAt: aKey ifAbsentPut: aBlock`: returns the value associated with `aKey`. If `aKey` is not found, evaluates the block, inserts the value of the block at `aKey` and returns the value.
- `aNode propertyAt: aKey ifPresent: aPresentBlock ifAbsent: anAbsentBlock`: Searches for the value associated with `aKey`. If `aKey` is found, evaluates `aPresentBlock` with its value. If `aKey` is not found, evaluates the block and returns 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, evaluates the block and returns its value.

Testing Methods

ASTs provide a testing protocol that can be 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`

1.12 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"
```

2. Draw the AST of method

```
unaryMethod
  self
```

3. Draw the AST of method

```
unaryMethod
  ^ self
```

4. Draw the AST of method

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

5. Draw the AST of method

```
+
  strange indentation
```

6. 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 simpler, 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?

1.13 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 seen an implementation of them: the RB ASTs. RB 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 principal. You should have now the basis to understand the concept of ASTs and we can move on to the next chapter.

CHAPTER 2

AST Visitors

In the previous Chapter we have seen how to create and manipulate AST nodes. The `RBParser` class implements a parser of expressions and methods that returns AST nodes for the text given as 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
```

However, more complex manipulations do 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.

In this section we start reviewing the visitor pattern, and we then apply it for a single task: searching a string inside the tree.

2.1 The Visitor Pattern

The Visitor pattern is one of the original design patterns from Gamma et al. [**?**], [**?**]. The main purpose of the Visitor pattern is to externalize an operation from a data structure. 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 subclass: #FileNode
  instanceVariableNames: 'size'
  package: 'VisitorExample'
```

```
Object subclass: #DirectoryNode
  instanceVariableNames: '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 ]
]
```

As the file system library forms trees that we can manipulate with a visitor, ASTs do so too. RBASTs are already extensible through visitors: 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.

2.2 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 subclass: #DepthCalculatorVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  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.

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 := RBParser 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: aRBMessageNode [
  ^ 1 + (aRBMessageNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
]
```

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
]
```

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 := RBParser 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: aRBMethodNode [
  ^ 1 + (aRBMethodNode children
      inject: 0
      into: [ :max :node | max max: (node acceptVisitor: self) ])
]
```

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

```
DepthCalculatorVisitor >> visitSelfNode: aSelfNode [
  ^ 1
]
```

2.3 Refactoring the implementation

This simple AST visitor does not actually require 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 a common ones: `visitNodeWithChildren:` and `visitLeafNode:`.

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

```
DepthCalculatorVisitor >> visitMessageNode: aRBMessageNode [
  ^ self visitNodeWithChildren: aRBMessageNode
]
[[[
DepthCalculatorVisitor >> visitMethodNode: aRBMethodNode [
  ^ self visitNodeWithChildren: aRBMethodNode
]
```

```
DepthCalculatorVisitor >> visitSequenceNode: aRBSequenceNode [
  ^ self visitNodeWithChildren: aRBSequenceNode
]
```

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

```
DepthCalculatorVisitor >> visitSelfNode: aSelfNode [
  ^ self visitLeafNode: aSelfNode
]
```

```
DepthCalculatorVisitor >> visitLiteralValueNode: aRBLiteralValueNode
    [
  ^ self visitLeafNode: aRBLiteralValueNode
]
```

2.4 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 alwaysa 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: aRBMessageNode [
  ^ self visitNode: aRBMessageNode
]
```

```
DepthCalculatorVisitor >> visitMethodNode: aRBMethodNode [
  ^ self visitNode: aRBMethodNode
]
```

```
DepthCalculatorVisitor >> visitSequenceNode: aRBSequenceNode [
  ^ self visitNode: aRBSequenceNode
]
```

```
DepthCalculatorVisitor >> visitSelfNode: aSelfNode [
  ^ self visitNode: aSelfNode
]
```

```
DepthCalculatorVisitor >> visitLiteralValueNode: aRBLiteralValueNode
    [
  ^ self visitNode: aRBLiteralValueNode
]
```

2.5 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 subclass: #BaseASTVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'VisitorExample'
```

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

```
BaseASTVisitor >> visitMessageNode: aRBMessageNode [
  ^ self visitNode: aRBMessageNode
]
```

```
BaseASTVisitor >> visitMethodNode: aRBMethodNode [
  ^ self visitNode: aRBMethodNode
]
```

```
BaseASTVisitor >> visitSequenceNode: aRBSequenceNode [
  ^ self visitNode: aRBSequenceNode
]
```

```
BaseASTVisitor >> visitSelfNode: aSelfNode [
  ^ self visitNode: aSelfNode
]
```

```
BaseASTVisitor >> visitLiteralValueNode: aRBLiteralValueNode [
  ^ self visitNode: aRBLiteralValueNode
]
```

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

```
BaseASTVisitor subclass: #DepthCalculatorVisitor
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'VisitorExample'
```

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

A more elaborate visitor could provide many more hooks. For example, in our example above we could have differentiated `RBSelf` nodes from `RBVariableNodes`, defining the following.

```
BaseASTVisitor >> visitSelfNode: aRBSelfNode [
  ^ self visitVariableNode: aRBSelfNode
]
```

```
BaseASTVisitor >> visitVariableNode: aRBVariableNode [
  ^ self visitNode: aRBVariableNode
]
```

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

2.6 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 understanding a complex visitor structure. Moreover, it was a good introduction for the `RBProgramNodeVisitor` class.

In this section we will implement a visitor that does require 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 `RBProgramNodeVisitor`. 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.

```
RBProgramNodeVisitor subclass: #SearchVisitor
  instanceVariableNames: 'token'
  classVariableNames: ''
  package: 'VisitorExample'

SearchVisitor >> token: aToken [
  token := aToken
]
```

The main idea of our visitor is that it will return a collection with all matching nodes. If no matching nodes are found, an empty collection is returned.

Searching in variables nodes

Let's then start implementing the visit methods for variable nodes. `RBProgramNodeVisitor` will already treat special variables as variable nodes, so a single visit method is enough for all four kind of nodes. A variable node matches the search if its name begins with the searched token.

```
SearchVisitor >> visitVariableNode: aNode [
  ^ (aNode name beginsWith: token)
      ifTrue: [ { aNode } ]
      ifFalse: [ #() ]
]
```

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, finally the arguments.

```
SearchVisitor >> visitMessageNode: aNode [
  ^ (aNode receiver acceptVisitor: self),
    ((aNode selector beginsWith: token)
      ifTrue: [ { aNode } ]
      ifFalse: [ #() ]),
    (aNode arguments gather: [ :each | each acceptVisitor: self ])
]
```

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 >> visitLiteralNode: aNode [
  ^ (aNode value asString beginsWith: token)
      ifTrue: [ { aNode } ]
      ifFalse: [ #() ]
]
```

The rest of the nodes

The rest of the nodes do not contain strings to search in them. Instead the contain children we need to search. We can then provide a common imple-

mentation for them by simply redefining the `visitNode:` method.

```
SearchVisitor >> visitNode: aNode [
  ^ aNode children gather: [ :each | each acceptVisitor: self ]
]
```

Another design would be to store the collection holding the rest inside the visitor and to avoid the temporary copies. We let you refactor your code to implement it.

2.7 Exercises

Exercises on the Visitor Pattern

1. Implement mathematical expressions as a tree, to for example 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.

1. Add a new kind of node to our expressions: raised to. Implement it in both the composite and visitor implementations.

1. 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?

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).

1. Extend your AST lineariser to handle different linearisation orders: breadth-first, depth-first pre-order, depth-first post-order.

1. 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.

1. 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 to the end of the tree. You can use the linearisations you implemented above.

2.8 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. The tree-like structure, in our case the AST, needs only to implement an accept-visit protocol. RB 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.