CHAPTER 1

Code Representation with Abstract Syntax Trees

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.

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 kinds of nodes will be instances of different classes, forming a composite.

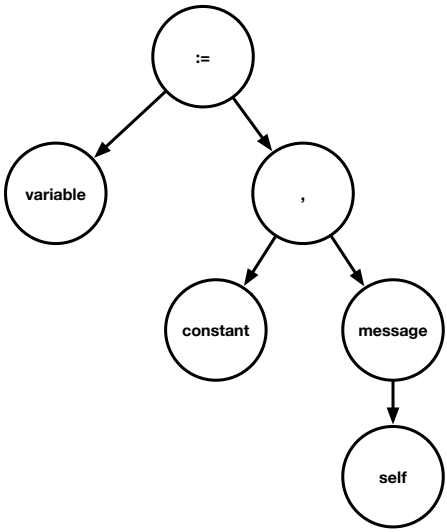

Figure 1-1 AST representing the code of `variable := 'constant' , self message`.

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.

1.2 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 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 `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
```

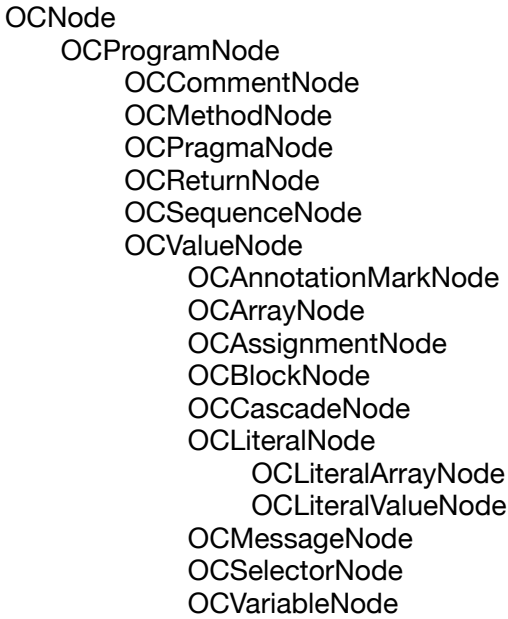

Figure 1-2 Overview of the method node hierarchy. Indentation implies inheritance.

```
"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.

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 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)
```

1.5 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 1-3. 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
```

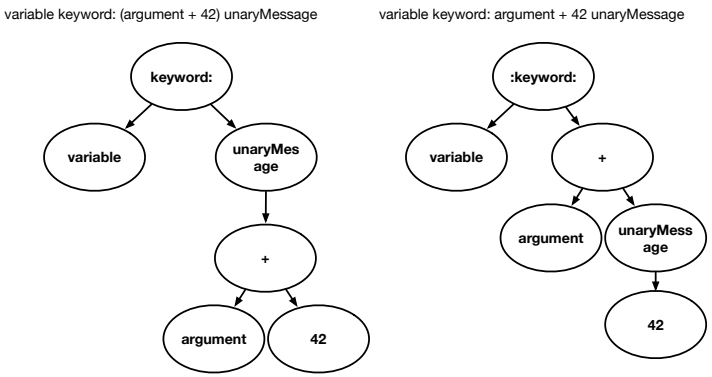

Figure 1-3 Different precedence results in different ASTs.

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!

1.6 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))
```

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 `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)))
```

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 `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
```

1.9 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))
```

1.10 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))
```

1.11 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`

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"
```

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?

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 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.