ParseTreeWriter Explained

S. Ducasse

March 12, 2024

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

Contents

Illustrations

CHAPTER 1

Checking and transforming programs with rewrite engine

Manipulating programs automatically is a really powerful tool. A refactoring tool is a typical example. Another example is a rule checking engine.

In this book, we present the engine named ParseTreeRewriter on top of which the Pharo refactoring engine and quality rule logic are built. The ParseTreeRewriter has been originally developed by J. Brant and D. Roberts.

This powerful engine allows you to define abstract expressions to match and transform code. The engine is based on an abstract syntax tree kind of unification algorithm that matches meta-variable to abstract syntax tree nodes and let you manipulate such meta-variables. We said that this is a kind of unification because it is not fully true.

In this book, we will first present the notion of abstract syntax tree, then the syntax of the parse tree rewriter expressions with practical examples.

We thanks for the following persons for their participation to this booklet from close or far: Gisela Decuzzi, Camille Teruel, John Brant, Lukas Renggli, Anne Etien, and Jean-Christophe Bach.

1.1 101 on AST and related sconcepts

An Abstract Syntax Tree (AST) is a tree data structure that represents source code as a tree of nodes. While source code is just a string of characters, an AST is a tree whose nodes correspond to the syntactic constructs of the parsed language. In Pharo, an AST node can represent a method, a message, a variable, a block, an assignment, etc.

Terminology point. Pharo ASTs are more concrete syntax trees in the sense that they keep information about the syntactic characters such as parentheses, square bracket and pipe for temporaries definitions. We call them AST in the following, but now you know that we are not fully correct.

Such a tree is much easier to process than a plain string. That is why this data structure is used by most tools that have to deal with code.

Let's met our first AST. Consider the following method.

```
Point >> < aPoint
  "Answer whether the receiver is above and to the left of aPoint."

  ^x < aPoint x and: [y < aPoint y]
```

You can get the AST of this method with the following expression.

```
(Point >> #<) ast
```

The root of the AST is a method node whose child is a return node. This return node has a message node and this message node has a receiver (the AST of the expression `x < aPoint x`), a selector (`and:`) and a block node as argument. Explore this AST to get familiar with its structure. You can also inspect the ASTs of other methods of the system.

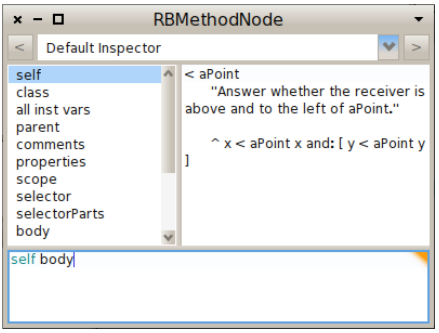

Figure 1-1 An inspector on an AST.

Sending the message body `body` to the AST returns the body of the method (as shown in Figure 1-1 and `selector` the method selector.

```
(Point >> #<) ast body
```

For example the AST of the following method is displayed in Figure **??**.

```
selector: arg1 and: arg2
  | temp1 temp2 |
  temp1 := OrderedCollection new.
  temp2 := arg1, arg2.
  ^ temp1 + temp2
```

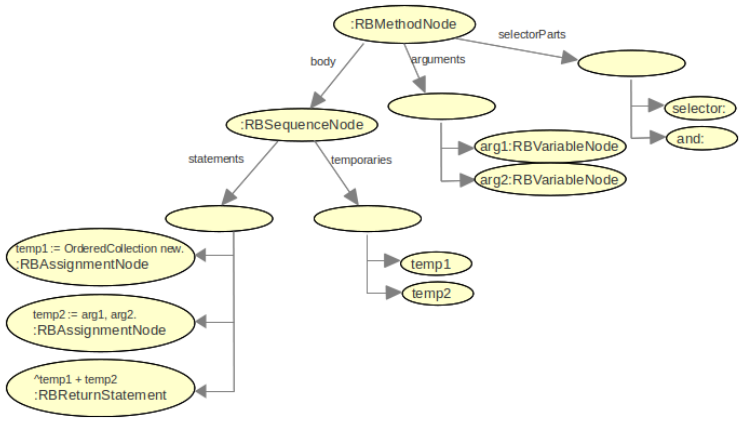

Figure 1-2 The AST of the method `selector:and:`

1.2 Basic knowledge on ProgramNode and its subclasses

Different kinds of nodes correspond to different classes. These classes form a hierarchy whose root is `RBProgramNode`. This class implements several useful methods such as: `parent`, `nodesDo:`, `isVariable`.... Browse it and its subclasses to discover the structure and the capabilities of Pharo ASTs. There is a visitor class that supports the walking through ASTs.

Here is a view on the ProgramNode hierarchy showing the classes and their instance variables. Of course, showing instance variables may be subject to change in the future. We show them to provide an overview.

```
RBProgramNode
  RBMethodNode
  RBParseErrorNode
  RBPragmaNode
  RBReturnNode
  RBSequenceNode
  RBValueNode
    RBArrayNode
    RBAssignmentNode
    RBBlockNode
    RBCascadeNode
```

```
RBLiteralNode
  RBLiteralArrayNode
  RBLiteralValueNode
RBMessageNode
RBVariableNode
  RBArgumentNode
  RBClassReference
  RBSelfNode
  RBSuperNode
  RBTemporaryNode
  RBThisContextNode
```

A SequenceNode is used by both a Block body and a MethodNode body.

1.3 About pattern matching

The pattern matching algorithm in the engine rewriter is the algorithm that will match a given variable to a subtree. This algorithm takes as input a pattern (that describes what we look for) and a tree (that could or not match the pattern). A pattern is a tree containing special nodes called meta-variables. A meta-variable is a variable whose value will be set by the pattern matching algorithm. For example, a meta-variable can be used to match a variable while another can be used to match messages.

The output of the algorithm is either false (there is no match found) or true. If it is true, that means that at least one match has been found. A match is a mapping from meta-variable to AST nodes of the input tree. These mappings correspond to the values the meta-variables of the pattern should take for the pattern to be equal to the input tree.

Let's look at a few examples. If X and Y denote meta-variables, the pattern `self X` matches `self open` with the mapping: `X -> 'open'`. The pattern `X Y` matches `self open` with the mapping: `Y -> 'self'`, `Y -> 'open'`. As a failing example, `X X` doesn't match `self open` because X cannot be at the same time 'self' and 'open'.

1.4 Pattern syntax

The ` character (backquote) creates a variable to match. Several options can be used to specify the search:

Keyword	Meaning
@	the searched node is optional or can be repeated. When applied to a variable, match any expression. When applied to a statement, m

. the searched node is a statement in a sequence node.
the searched node is a literal
{} to match nodes that satisfies the enclosed code
‘ to search recursively: When a match is found recurse into the matched node.

Let’s explain each construct by using it in a concrete context

1.5 Example

Let us start with an example and we want to make sure that our example is always correct to we will verify it using tests.

First we define a class to host

```
TestCase << #MMatcherTest
  slots: { #searcher level };
  package: 'myBecher-Matcher'
```

The instance variable `searcher` will refer to an instance of RBParseTreeSearcher and level is just an instance variable to illustrate that we can match different kind of nodes. We make sure that we want to reuse the logic as much as possible so we define a `setUp` method and some helpers.

```
MMatcherTest >> setUp
  super setUp.
  searcher := RBParseTreeSearcher new.
```

```
MMatcherTest >> search
  searcher
    executeTree: (MMatcherTest >> #level) ast
    initialAnswer: OrderedCollection new.
```

Then we define a simple method named `level` as follows:

```
level
  | current |
  level := 0.
  current := self resolve.
  [ current isRoot ] whileFalse: [
    level := level + 1.
    current := current parent ].
  ^ level
```

We are now ready to check some searches.

1.6 Not using meta-variables

The first question is can ask to the searcher is to find an exact node. Here we look for the node (a message node) that represents the message `current node`.

```
testNoVariable

  searcher
    matches: 'current isRoot'
    do: [ :aNode :answer | answer add: aNode ; yourself].
  self search.
  self assert: searcher answer first class equals: RBMessageNode.
  self assert: searcher answer first printString equals:
    'RBMessageNode(current isRoot)'
```

For example, the pattern `current isRoot` matches the single occurrence of the parse-tree node of current isRoot.

```
testNoVariableMultipleMatches

  searcher
    matches: 'current'
    do: [ :aNode :answer | answer add: aNode ; yourself].
  self search.
  self assert: searcher answer size equals: 4.
  self assert: (searcher answer collect: [ :each | each start ])
    asArray equals: #(34 62 116 127)
```

Looking for the nodes that matches `current` we get four different temporary nodes each with its own source locations. It shows that the temporary definition is not part of the matches.

```
testNoVariableLevelMultipleMatches

  searcher
    matches: 'level'
    do: [ :aNode :answer | answer add: aNode ; yourself].
  self search.
  self assert: searcher answer size equals: 4
```

The previous test shows that we can match instance variables.

1.7 A first meta-variable

The following test illustrates what will happen when we define a meta-variable matching unary messages sent to the `current` variable.

The pattern `current `selector` all the unary message sends to the receiver `current`. We obtain the two message nodes `current isRoot` and `current parent`.

```
testSelectorToCurrent

  searcher
    matches: 'current `selector'
    do: [ :aNode :answer | answer add: aNode ; yourself].
```

```
  self search.
  self assert: searcher answer size equals: 2.
  self
    assert: searcher answer printString
    equals: 'an OrderedCollection(RBMessageNode(current isRoot)
     RBMessageNode(current parent))'
```

To be able to match binary messages we will indicate that we can have an optional number of argument using the expression `@arg. In the following we match the message node `level + 1`.

```
testSelectorToLevel
  searcher
    matches: 'level `selector: `@arg'
    do: [ :aNode :answer | answer add: aNode ; yourself].
  self search.
  self assert: searcher answer size equals: 1.
  self
    assert: searcher answer printString
    equals: 'an OrderedCollection(RBMessageNode(level + 1))'
```

1.8 Examples of transformations

```
| `@Temps | ``@.Statements. ``@Boolean ifTrue: [^false]. ^true
| `@Temps | ``@.Statements. ^``@Boolean not
``@object not ifTrue: ``@block
``@object ifFalse: ``@block.
```

```
RBParseTreeRewriter new
  replace: '``@aDictionary at: ``@key
    ifAbsent:
      [| `@Temps |
      ``@.Statements.
      ``@aDictionary at: ``@key put: ``@value]' with:
    '``@aDictionary at: ``@key
    ifAbsentPut:
      [| `@Temps |
      ``@.Statements.
      ``@value]';
  yourself
```

```
rule := RBUnderscoreAssignmentRule new.
environment := BrowserEnvironment new forPackageNames: #('PackageA'
'PackageB' ...).
SmalllintChecker runRule: rule onEnvironment: environment.
rule open
```

1.9 Example of the globals message

Let's start it with an example, imagine that we want to identify all the places where we have `anObject globals at:`. We want to identify message as the ones in the `configurationClass` method below.

```
configurationClass
  ^ Smalltalk globals at: self configurationName asSymbol
```

First try to look for the senders of the message `globals`. You will see that you get couple hundreds of them and plenty of them are not sending `at:` but `at:ifAbsent:` or `at:put:` or even nothing to do with `at:`.

1.10 Let's write a matching pattern.

First you want any object, it could be Smalltalk or another one so we use the ` operation to create a (meta)variable that we name `anObject`. A meta variable is a variable that we will unified against the AST elements.

So we get

```
`anObject
```

Then since we want the message `globals` we add it and obtain.

```
`anObject globals
```

Note here that we do not annotate it with an operator because we do not want any message selector but just `globals`. We redo the same for `at:` so we obtain the following pattern.

```
`anObject globals at:
```

But this expression is not complete and it is missing an argument to be a full Pharo expressions. So we add another meta-variable that will match anything.

```
`anObject globals at: `anything
```

1.11 Executing the rule

Now we have the pattern but we should execute it to get the matches. For now we will use a nice extension of the Refactoring rule engine called Flamel.

TO BE CHANGED:No more flamel

```
FlamelRule new
   matchingExpression: '`anObject globals at: `anything';
   matchExpression;
   run;
   result
```

Execute and inspect the result. It may take some minutes because it is matching all the methods of the system. You get an `RBParseTreeEnvironment` and the results are in `classSelectors` and `metaSelectors`.

Let us explain the script, once the rule object is created we specify the expression using `matchingExpression:`, then we specify that we want to match an expression (and not a complete method as we will explain in Section *MatchingMethods*, then we run the rule and ask for its results.

1.12 Refining the scope

Sometimes you do not want to look in the complete system but in a restricted environment. To do so we use the message `scope:`

TO BE CHANGED: no more flamel

```
FlamelRule new
   matchingExpression: '`anObject globals at: `anything';
   matchExpression;
   scope: (RBPackageEnvironment packageName: 'Kernel');
   run;
   result
```

Now if we want to restrict the search to a subset of the environment.

1.13 Running your transformation

```
searchPattern

  ^ '`selector
| expr source |
expr := RBParser parseExpression:  `#input.
configurationSelector := `#configuration.
`@.st1.
self assert: source equals: `#output'
```

```
targetPattern

  ^ '`selector
| source |
configurationSelector := `#configuration.
source := self formatExpression:  `#input.
self assert: source equals: `#output'! !
```

```
rewrite: aCompiledMethod
  "self new rewrite"
  | rewriter ok |
  rewriter := RBParseTreeRewriter new.
  rewriter replaceMethod: self searchPattern with: self
```

```
    targetPattern.
ok := rewriter executeTree: aCompiledMethod parseTree.
ok ifFalse: [ ^ 'did not work' ].
Author
  useAuthor: 'Refactoring'
  during: [
    aCompiledMethod origin
      compile: rewriter tree formattedCode
      classified: aCompiledMethod protocol ]
```

CHAPTER 2

Revisting the DSL

In this chapter we describe the DSL with many examples.

- single node (e.g., `` `@node ``)
- multiple nodes (e.g., `` `.@statements ``)

2.1 Single variables

- A single backquote ` introduces the most basic meta-variable.
- foo‘ only matches single variable.

Examples:

The following code enables to define a method node in which we want to find the pattern. An `RBParseTreeSearcher` enables to define a pattern containing the meta-variables. The resulting substitutions are gathered into the `answer` variable:

```
| methodNode searcher |
methodNode := OpalCompiler new parse: 'test
x bar'.
searcher := RBParseTreeSearcher new.
searcher
  matches: '``foo bar'
  do: [ :aNode :answer | answer add: aNode ; yourself].
searcher executeTree: methodNode initialAnswer: Set new.
searcher answer.
```

In the following tables, only the code corresponding to the method node is changed.

foo bar‘ **matches** the following expressions:

Input	Result	Explanation
x bar	`{ foo=x }`	x is a variable
self bar	`{ foo=self }`	`self` is a RBSelfNode, that is cor
OrderedCollection bar	`{ foo=OrderedCollection }`	`OrderedCollection` is consider
x toto ; bar	`{ foo=x }`	`x toto; bar` is a `RBCascadeNo`
x bar. y bar	`{ foo=x, foo=y }`	x and y are variables.
self x bar. y bar	`{ foo=y }`	y is variable and `self x` is not a

`` `foo bar `` **does not match** the following expressions:

Input	Explanation
bar	The pattern must contain one and only variable on which the bar selector is ca
3 bar	`3` is a literal (not a variable)
#(2 3) bar	`#(2 3)` is a literal array
{2. 3} bar	`{2. 3}` is an array
#at bar	`#at` is a literal
[true] bar	`[true]` is a block
(x:=1) bar	`(x:=1)` is a statement
self x bar	`self x` is a RBMessageNode

2.2 Literal variables

`` `#lit `` matches literals, e.g. numbers, string, literal arrays, booleans.

In this part, the following snippet of code is used. The matching expression is changed to `` `#lit bar ``:

```
| methodNode searcher |
methodNode := OpalCompiler new parse: 'test
3 bar'.
searcher := RBParseTreeSearcher new.
searcher
  matches: '`#lit bar'
  do: [ :aNode :answer | answer add: aNode ; yourself].
searcher executeTree: methodNode initialAnswer: Set new.
searcher answer.
```

`` `#lit bar `` **matches** the following expressions:

Input	Result	Explanation
3 bar	`{ lit=3 }`	`3` is a number
'foo' bar	`{ lit='foo' }`	`'foo'` is a string
#at bar	`{lit=#at}`	`#at` is a ByteSymbol
#(a b c) bar	`{ lit=#(a b c)}`	`#(a b c)` is a literal array

true bar	`{lit=true}`	`true` is a boolean

`` `#lit bar `` **does not match** the following expressions:

Input	Explanation
{ 1. 2. 3 } bar	`{1. 2. 3}` is a collection
(1/2) bar	`(1/2)` is a message (and not considered as a number)
(1=1) bar	`(1=1)` is a message (and not considered as a boolean)

2.3 Statements

`` `.stmt `` matches a single statement.

`` `.stmt bar `` **matches** the following expressions:

! Input	! Result
3	`{ stmt=3 }`
x:=1	`{ stmt=x:=1 }`
x ifTrue: [doSomething]	`{stmt=x ifTrue: [ doSomething]}`
x:=1. ^x	`{ stmt=x:=1, stmt=^x}`

2.4 Conditional matching

`` `{ :node | ''conditionOnNode'' } `` matches a node satisfying ”conditionOnNode” which is expressed in Pharo. The parsed code being not executed, the condition concerns the characteristic of the node but cannot refer to computed values of this code.

The following snippets of code illustrate the conditional matching:

```
methodNode := OpalCompiler new parse: 'test
a := 1.
^a'.
searcher := RBParseTreeSearcher new.
searcher
  matches: '`{ :node | node name = #a}'
  do: [ :aNode :answer | answer add: aNode ; yourself].
searcher executeTree: methodNode initialAnswer: OrderedCollection
    new.
searcher answer.
```

Figure 2-1 shows the inspector on answer. Two nodes match the condition: the left-hand side of the assignment and the returned variable.

If the condition is changed as following:

```
methodNode := OpalCompiler new parse: 'test
a := 1.
^a'.
searcher := RBParseTreeSearcher new.
```

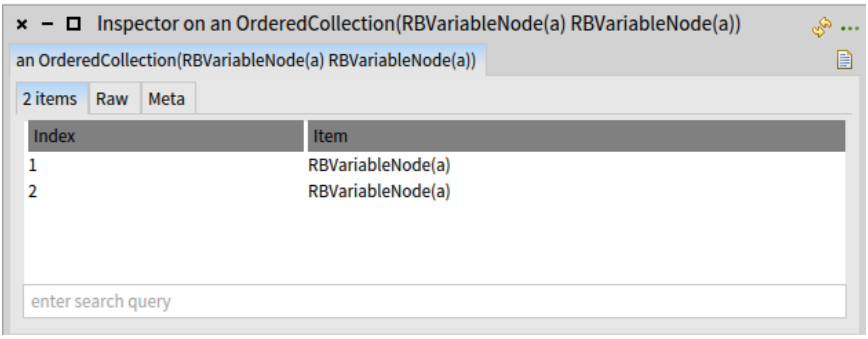

Figure 2-1 Inspector on answer

```
searcher
  matches: '`{ :node | node value = 1}'
  do: [ :aNode :answer | answer add: aNode ; yourself].
searcher executeTree: methodNode initialAnswer: OrderedCollection
    new.
searcher answer.
```

Only one node matches the condition: the right-hand side of the assignment. Since the code of the parsed method is not executed, nodes corresponding to a variable are not matched.

The condition can for example be used to only select node of a specific type (`node isInstance` enables to select instance variables).

2.5 List variables

`` `@subtree `` matches any subtree. This construct can only be used in first position. For instance, `bar @subtree‘` is not a valid pattern.

`` `@subtree bar `` **matches** the following expressions:

Input	Result
self bar	`{subtree=self}`
3 bar	`{subtree=3}`
#(2 3) bar	`{subtree=#(2 3)}`
{2. 3} bar	`{subtree={2. 3}}`
#at bar	`{subtree=#at}`
[true] bar	`{subtree=[true]}`

(x:=1) bar	`{subtree=(x:=1)}`
self foo bar	`{subtree=self foo}`

`` `@subtree bar `` **does not match** the following expressions:

Input	Explanation
bar	The empty node is not a subtree

2.6 Recursive search

It is possible to elaborate more complex patterns. For instance `self `msg: `arg` matches any one-argument message sent to self:

```
self as: Array
self instVarAt: 2
```

2.7 Matching a list of zero or more nodes

`` `@nodes `` To match a list of zero or more nodes (like message args or temps).

Examples:

`` `@receiver foo `` matches

```
self foo
self size foo
(self at: 1) foo
```

`self `@msg: `@args` matches any message sent to self:

```
self class
self at: 4
self perform: #at: with: 4
```

2.8 Recursive search

`` `` supports recursive search. For example, `@receiver parent `matchesx parent parent‘ twice:

- the first time with the mapping: receiver -> x
- the second time eith the mapping: receiver -> x parent

```
methodNode := OpalCompiler new parse: 'test
x parent parent.'.
searcher := RBParseTreeSearcher new.
searcher
  matches: '``receiver parent'
  do: [ :aNode :answer | answer add: aNode ; yourself].
```

```
searcher executeTree: methodNode initialAnswer: OrderedCollection
    new.
searcher answer.
```

The following table summarizes the use of `, @ and `` with a concrete input:

Pattern	Input	Result
‘r parent	x parent	{ x }
‘@r parent	x parent	{ x }
“@r parent	x parent	{ x }
“r parent	x parent	{ x }
‘r parent	x parent parent	{ x }
‘@r parent	x parent parent	{ x parent }
“@r parent	x parent parent	{ x, x parent }
‘r parent | x parent parent	| Matches with x as r	
“r parent	x parent parent	{ x }

CHAPTER 3

Expressing and executing Rules

-

Before start you should choose if you want to: perform a search do a match

Implement your rules because all of them are abstract, probably you will end up using the Tree rules because are more automatics and powerfull than to basics.

When you are defining your rule:

- give a name for it
- define if you will use a method or an expression type pattern (use `matches:` or `matchesMethod:`)
- write your patterns and add them to the rule
- if you are in a search rule to the matcher
- if you are in a transformation rule rewriteRule
- define what to do with a result
- run your rule

An example

We want to match, so let's create an object that extends: `RBParseTreeLintRule`:

```
RBParseTreeLintRule << #SearchGlobalsAtUsage
    package: 'Blog-example'
```

We have to implement the abstract methods:

```
SearchGlobalsAtUsage >> name
  ^ 'Find all potential wrong usage in with globals'
```

I want to match an expression type because I do not care about the rest of the method, I want everything that contains the messages: #globals #at:, we also have to say what to do with the matching node in this example I will open an inspector:

```
SearchGlobalsAtUsage >> initialize
  super initialize.
  self matcher
    matches: '``@lotOfStuffBefore globals at: ``@lotOfStuffAfter'
    do: [ :theMatch :theOwner | theMatch inspect ].
```

Now we should run the rule: WARNING: this can take a time because you will check the whole system.

```
SearchGlobalsAtUsage new run.
```

To avoid this you can restrict the environment for your rule, an example:

```
rule := SearchGlobalsAtUsage new.
environment := RBClassEnvironment class: Result.
RBSmalllintChecker runRule: rule onEnvironment: (environment).
```

If you have matches then you will see the inspector.