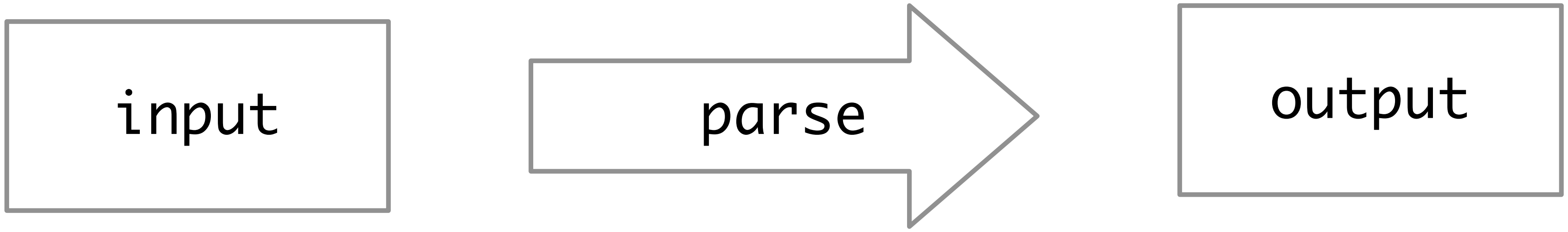
PetitParser

www.tudorgirba.com

based on the work of Lukas Renggli
www.lukas-renggli.ch

PetitParser

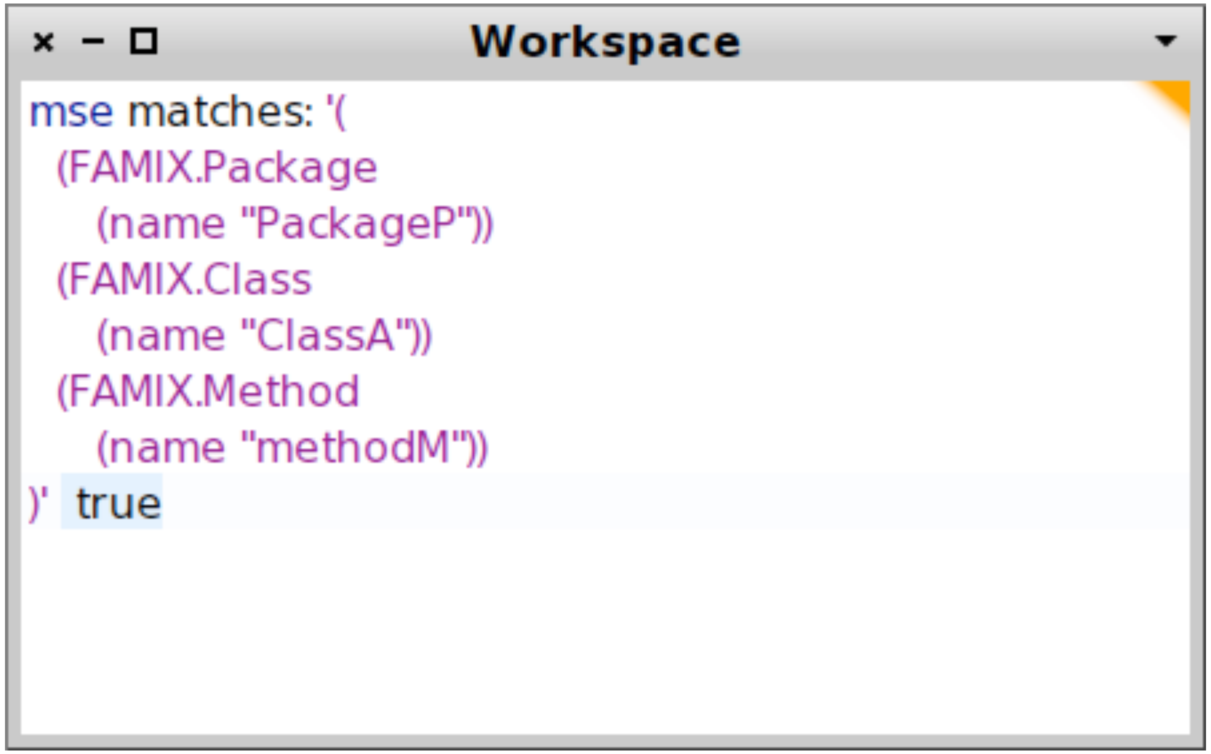
built by Lukas Renggli
deeply integrated with Smalltalk
part of the Moose Suite


input
parse
output

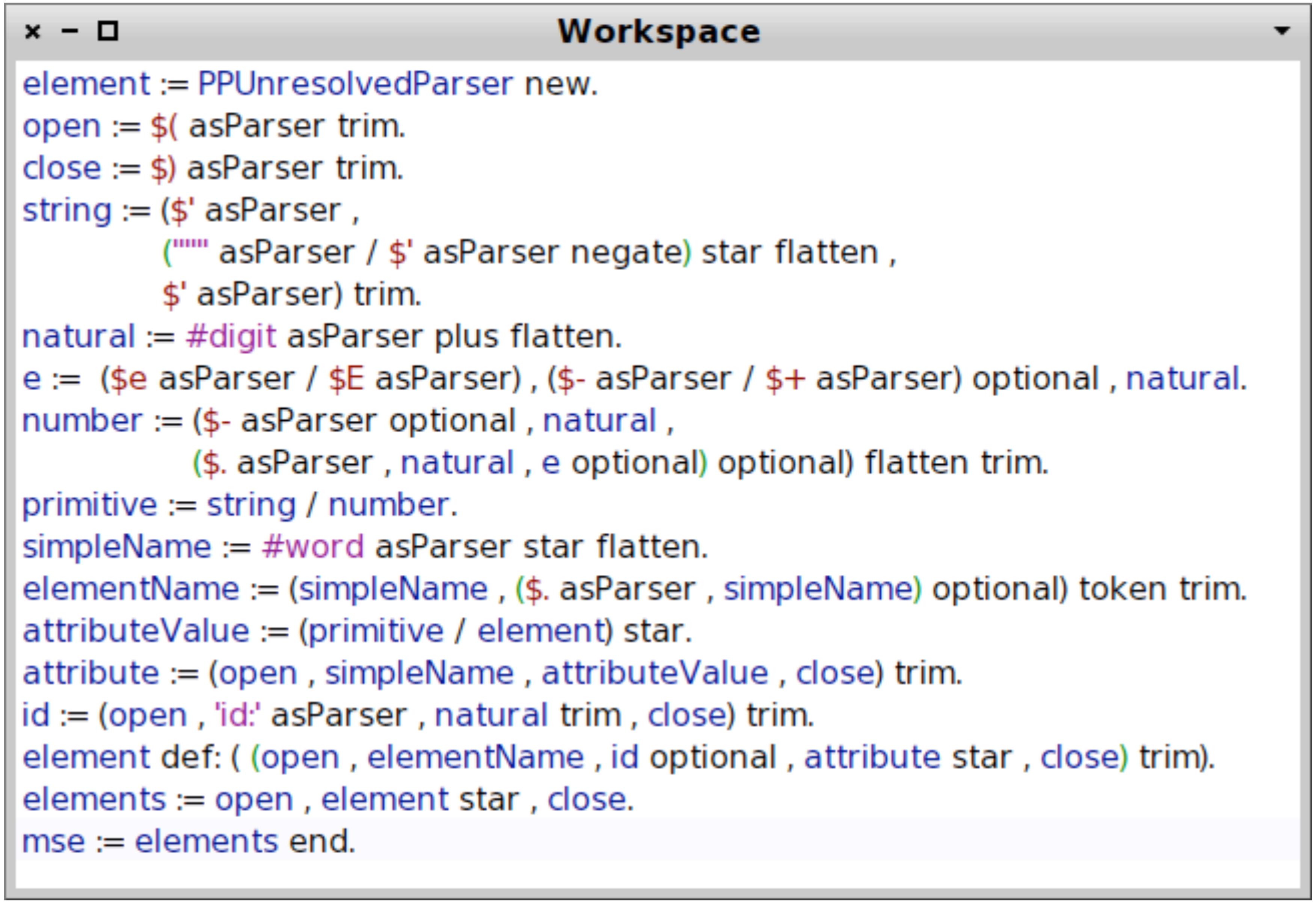
PetitParser

```
Root := Document ?
Document := OPEN ElementNode * CLOSE
ElementNode := OPEN ELEMENTNAME AttributeNode * CLOSE
AttributeNode := OPEN SIMPLENAME ValueNode * CLOSE
ValueNode := Primitive | ElementNode
Primitive := STRING | NUMBER
OPEN := "("
CLOSE := ")"
ELEMENTNAME := letter ( letter | digit ) * ( "." letter ( letter | digit ) ) *
SIMPLENAME := letter ( letter | digit ) *
NUMBER := "-" ? digit + ( "." digit + ) ? ( ( "e" | "E" ) ( "-" | "+" ) ? digit + ) ?
STRING := ( "'" [^'] * "'" ) +
digit := [0-9]
letter := [a-zA-Z_]
comment := """ [^"] * """
```

target

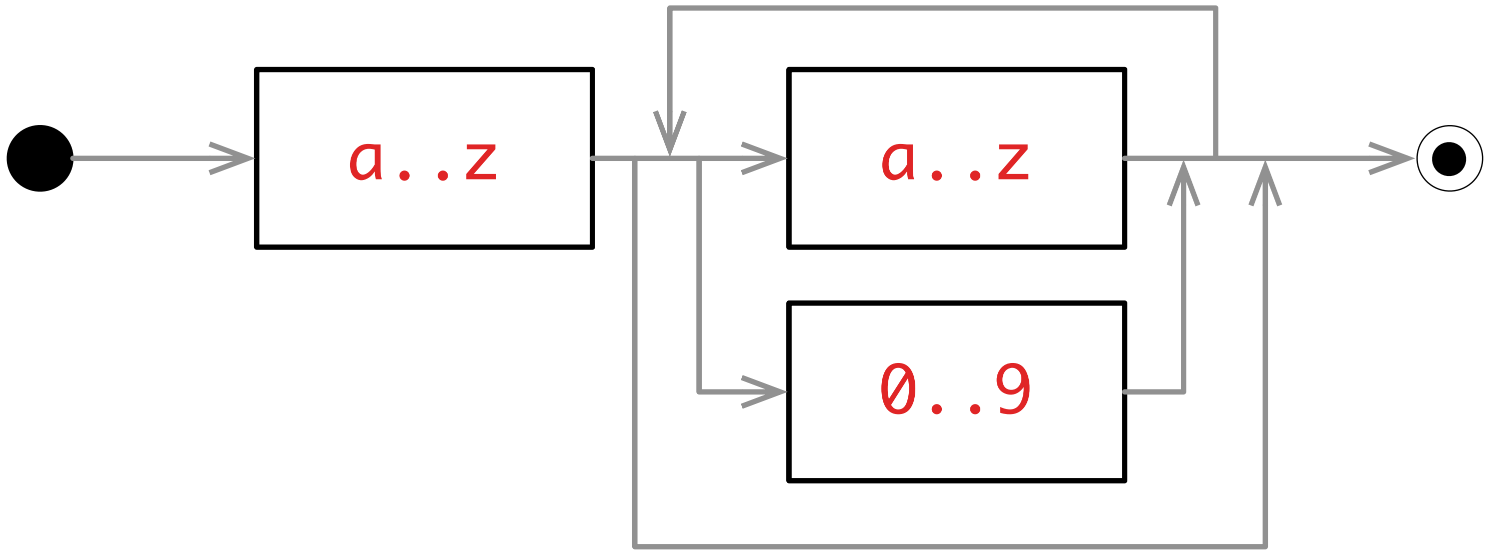


```
mse matches: '(
 (FAMIX.Package
    (name "PackageP"))
 (FAMIX.Class
    (name "ClassA"))
 (FAMIX.Method
    (name "methodM"))
)'  true
```

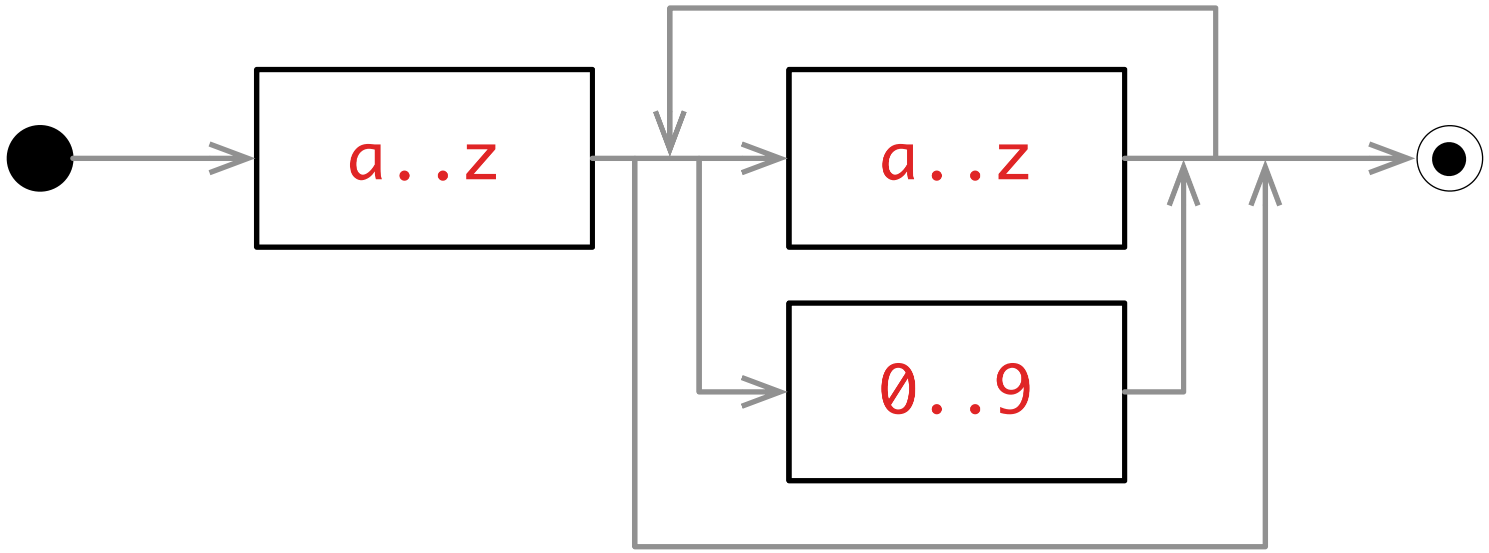


```
element := PPUnresolvedParser new.
open := $( asParser trim.
close := $) asParser trim.
string := ($' asParser ,
            ('''' asParser / $' asParser negate) star flatten ,
            $' asParser) trim.
natural := #digit asParser plus flatten.
e :=  ($e asParser / $E asParser) , ($- asParser / $+ asParser) optional , natural.
number := ($- asParser optional , natural ,
              ($. asParser , natural , e optional) optional) flatten trim.
primitive := string / number.
simpleName := #word asParser star flatten.
elementName := (simpleName , ($. asParser , simpleName) optional) token trim.
attributeValue := (primitive / element) star.
attribute := (open , simpleName , attributeValue , close) trim.
id := (open , 'id:' asParser , natural trim , close) trim.
element def: ( (open , elementName , id optional , attribute star , close) trim).
elements := open , element star , close.
mse := elements end.
```

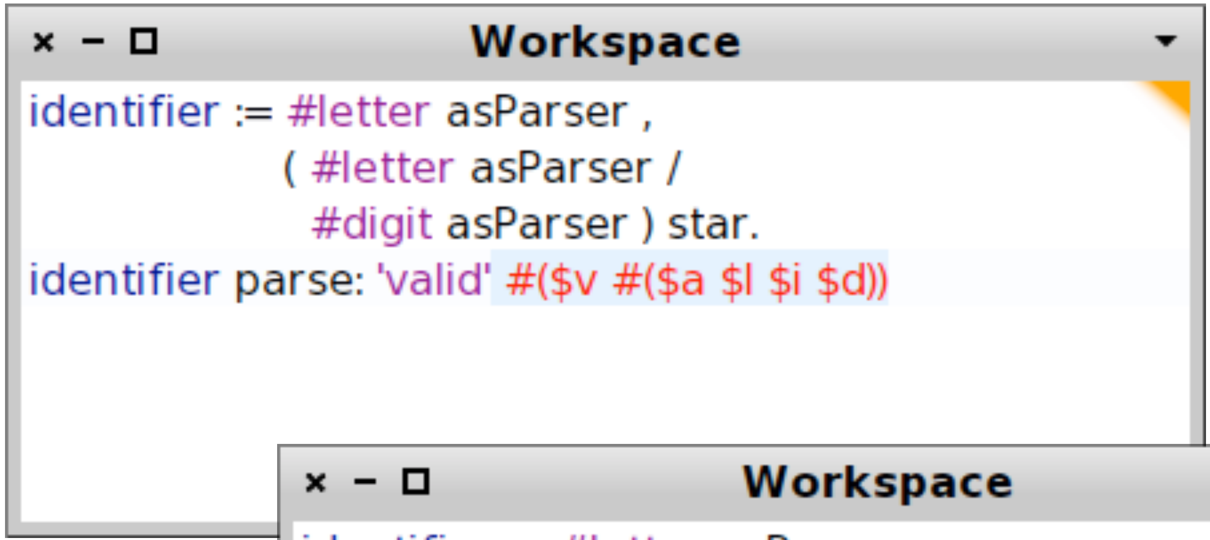
```
IDENTIFIER ::= letter
               ( letter |
                 digit ) *
```



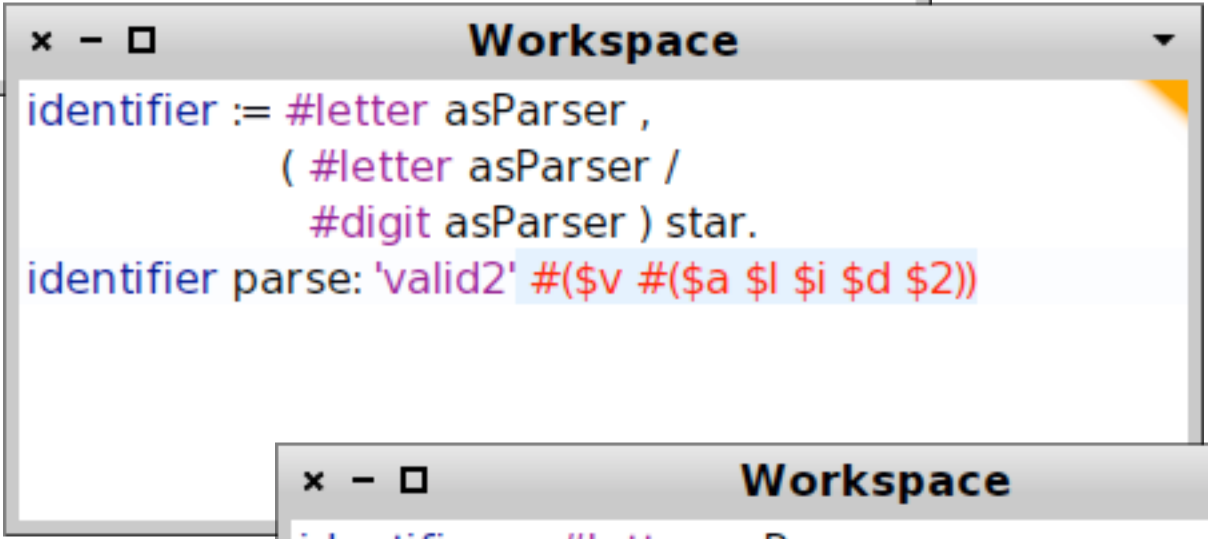
```
identifier := #letter asParser ,
              ( #letter asParser /
                #digit asParser ) star.
```



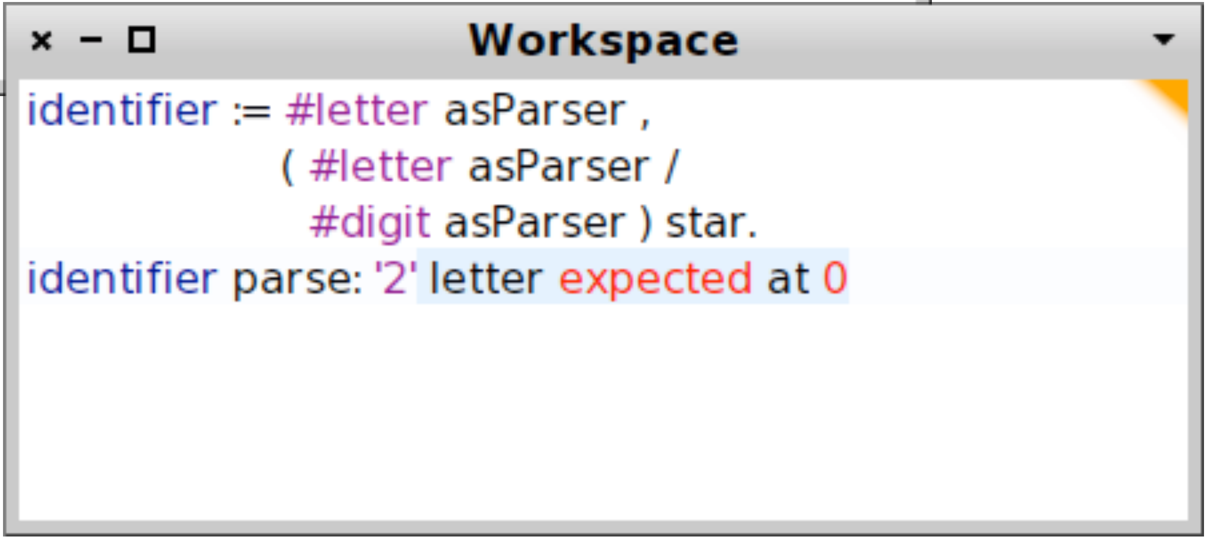
exercise



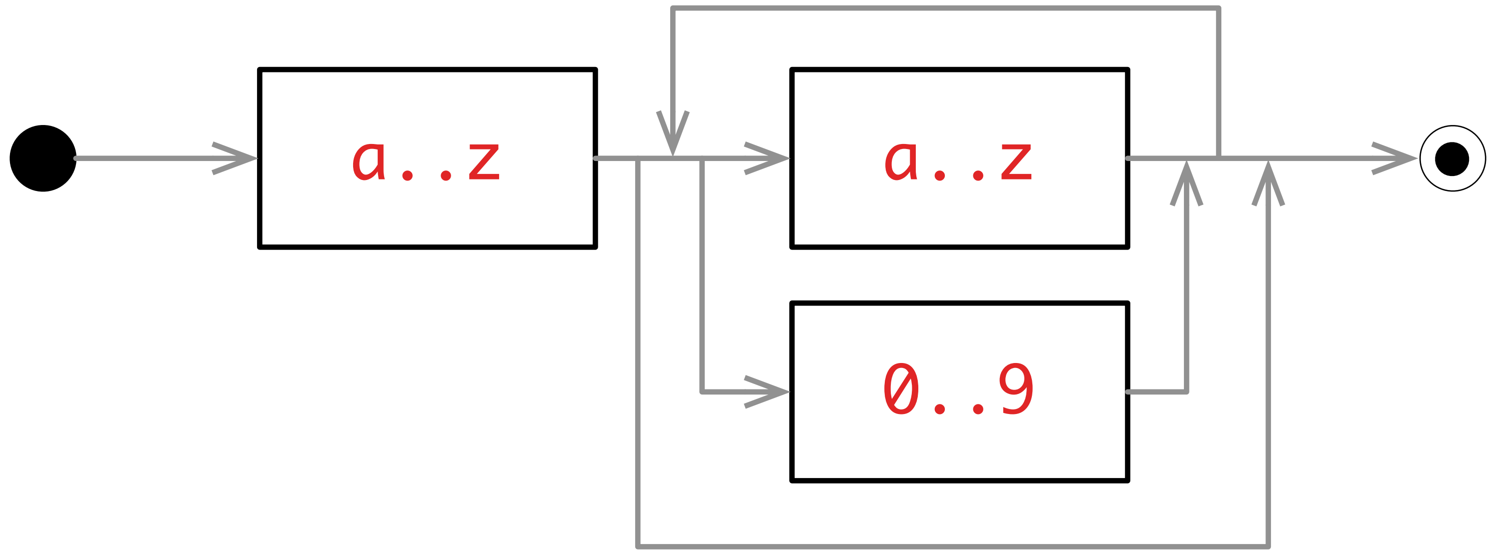
```
identifier := #letter asParser ,
              ( #letter asParser /
                #digit asParser ) star.
identifier parse: 'valid' #($v #($a $l $i $d))
```



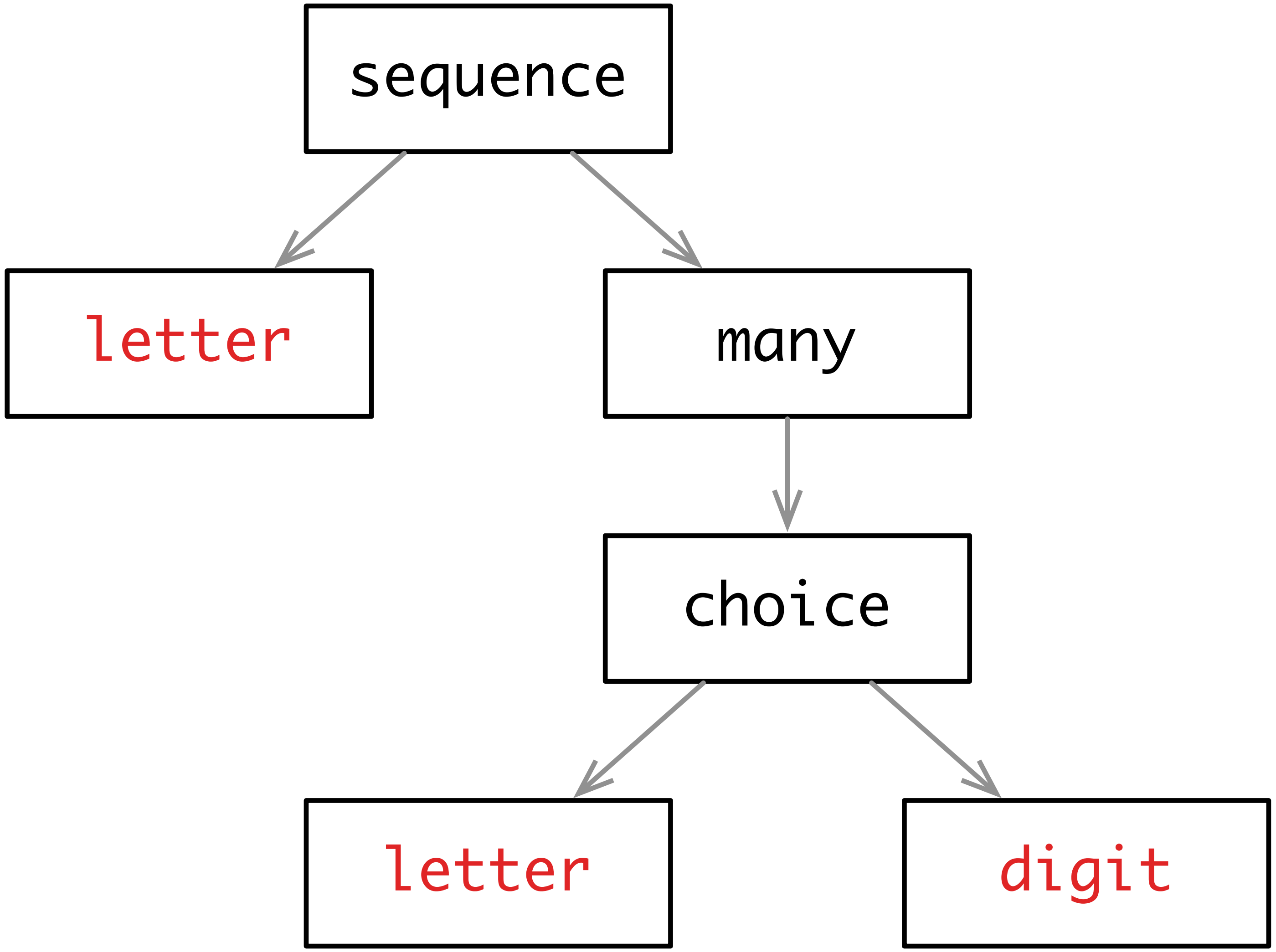
```
identifier := #letter asParser ,
              ( #letter asParser /
                #digit asParser ) star.
identifier parse: 'valid2' #($v #($a $l $i $d $2))
```



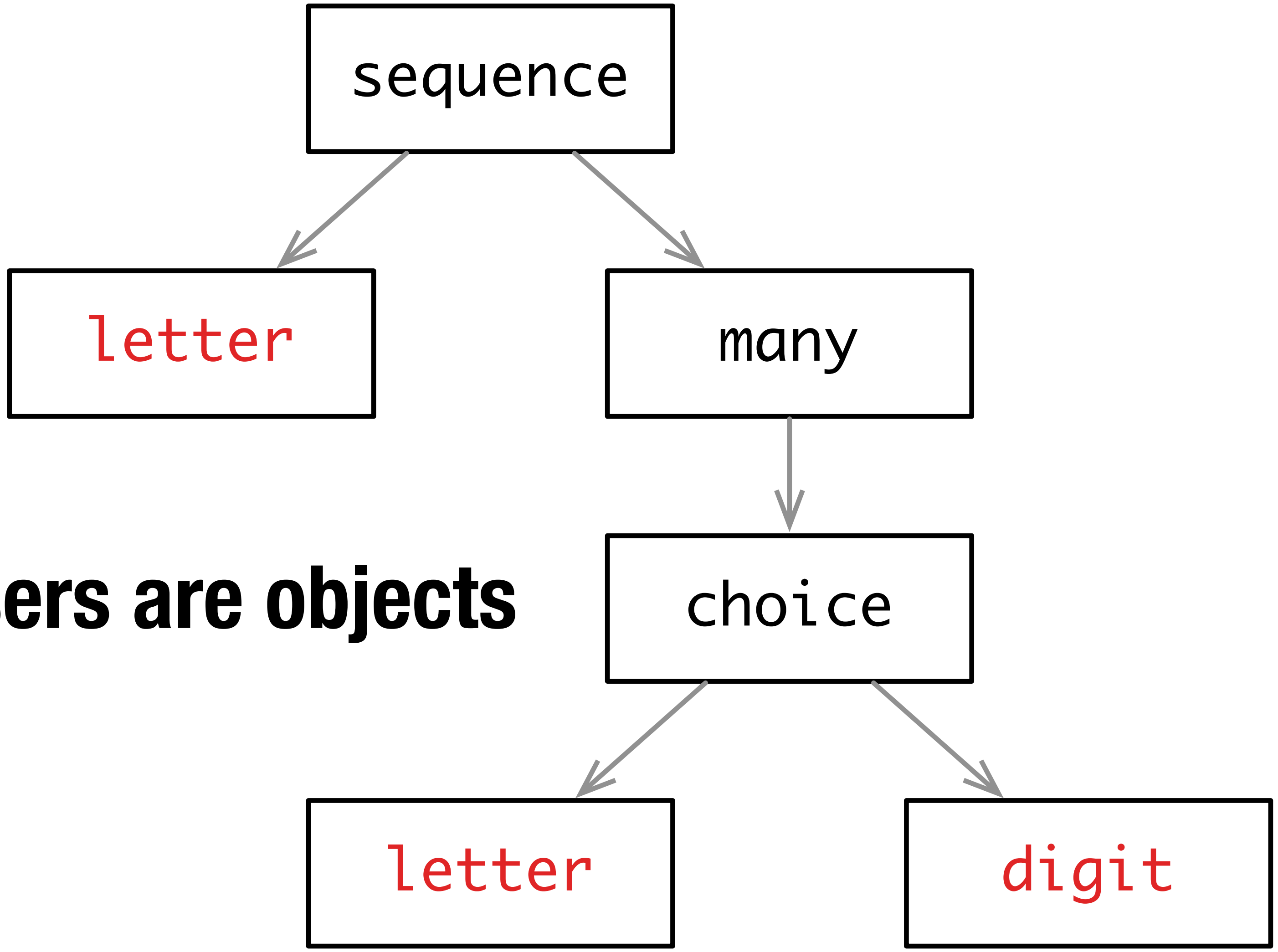
```
identifier := #letter asParser ,
              ( #letter asParser /
                #digit asParser ) star.
identifier parse: '2' letter expected at 0
identifier := #letter asParser ,
              ( #letter asParser /
                #digit asParser ) star.
```



```
identifier := #letter asParser ,
              ( #letter asParser /
                #digit asParser ) star.
```



```
identifier := #letter asParser ,
                 ( #letter asParser /
                   #digit asParser ) star.
```

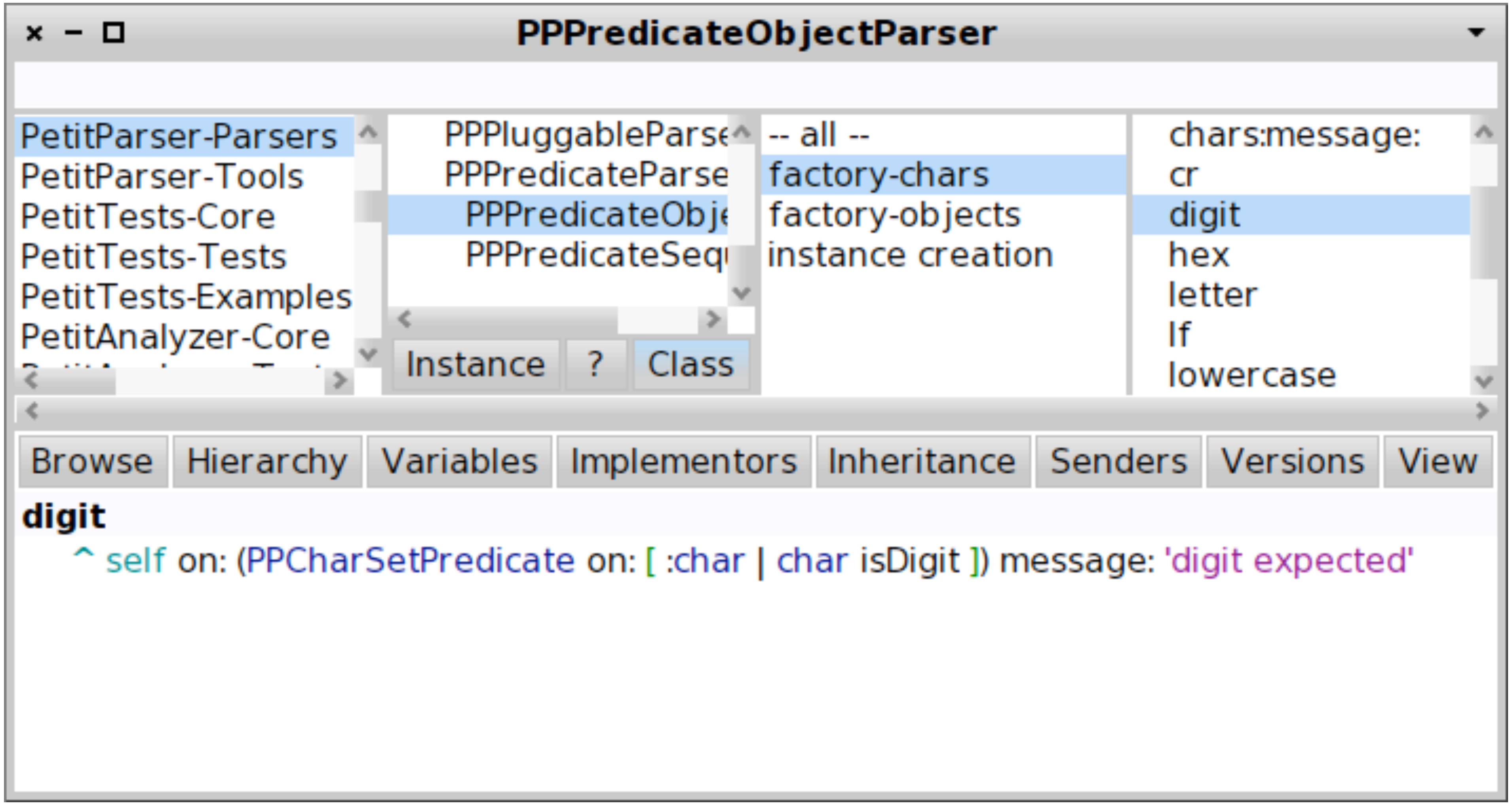


parsers are objects

terminals

`$c`	`asParser`	parse character “c”
`‘string’`	`asParser`	parse string “string”
`#any`	`asParser`	parse any character
`#digit`	`asParser`	parse one digit
`#letter`	`asParser`	parse one letter

terminals are defined in PPPredicateObjectParser



```
digit
    ^ self on: (PPCharSetPredicate on: [ :char | char isDigit ]) message: 'digit expected'
```

exercise: browse PPPredicateObjectParser

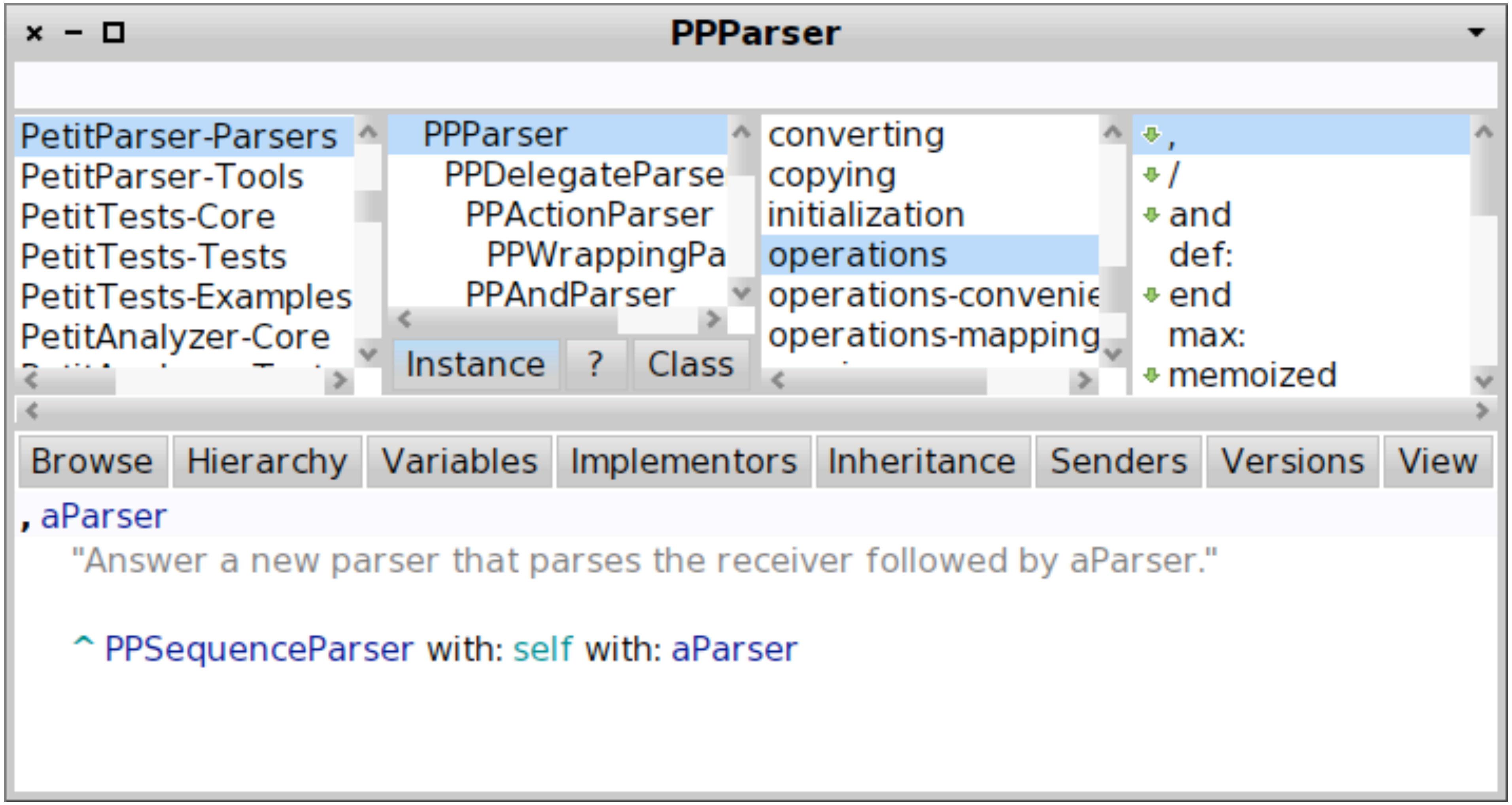
combinators

`p1 , p2`	parse p1 followed by p2 (sequence)
`p1 / p2`	parse p1, otherwise parse p2 (ordered choice)
`p star`	parse zero or more p
`p plus`	parse one or more p
`p optional`	parse p if possible

predicates

`p not`	negation (non-consuming look-ahead)
`p negate`	negation (consuming)
`p end`	end of input

PPParser is the root of all parsers

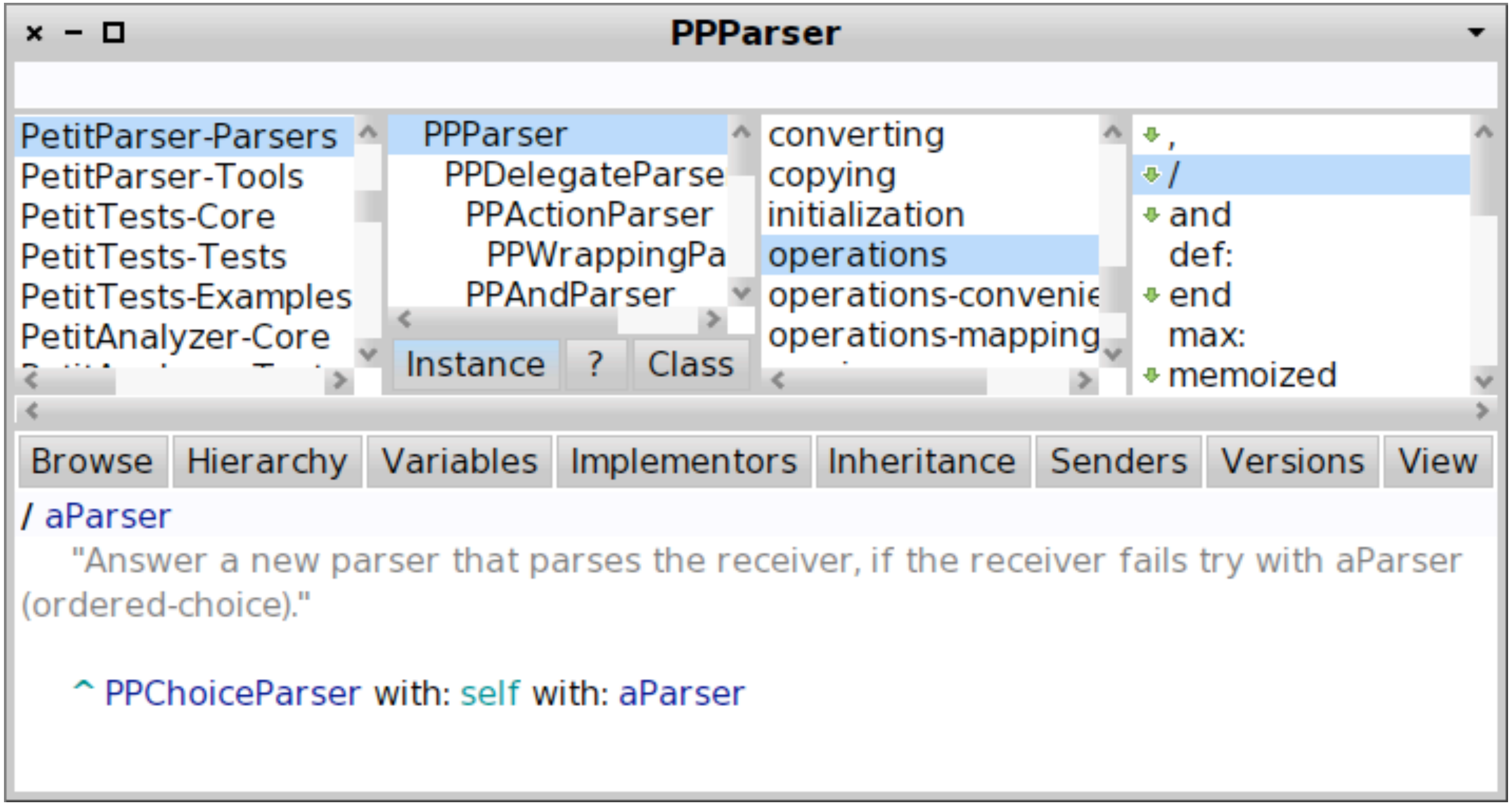


```
, aParser
	"Answer a new parser that parses the receiver followed by aParser."

	^ PPSequenceParser with: self with: aParser
```

all operations are defined in this class

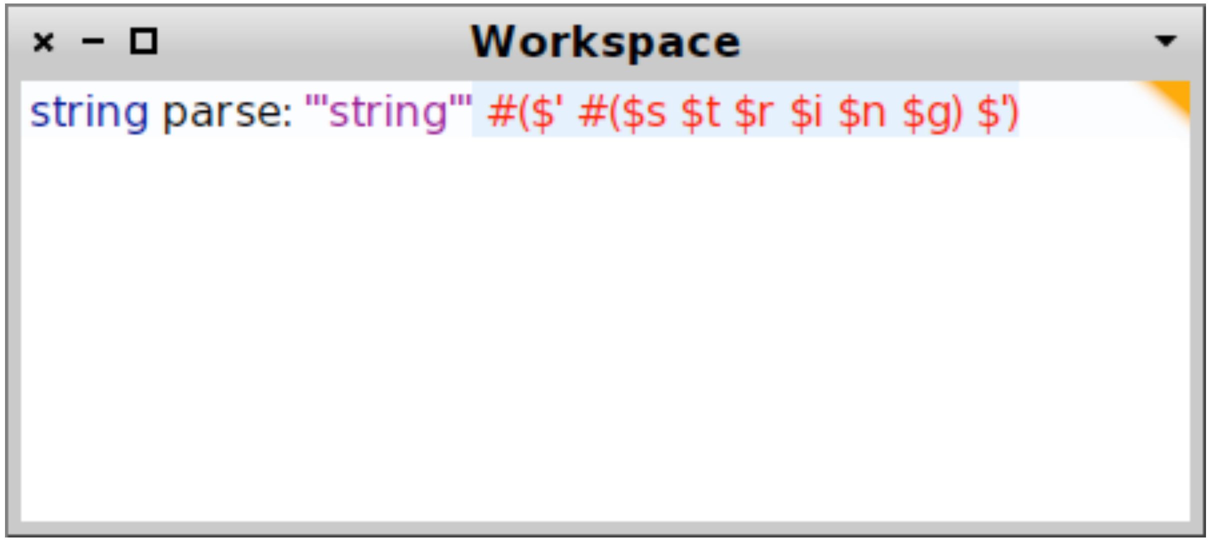
exercise: browse PPParser operations



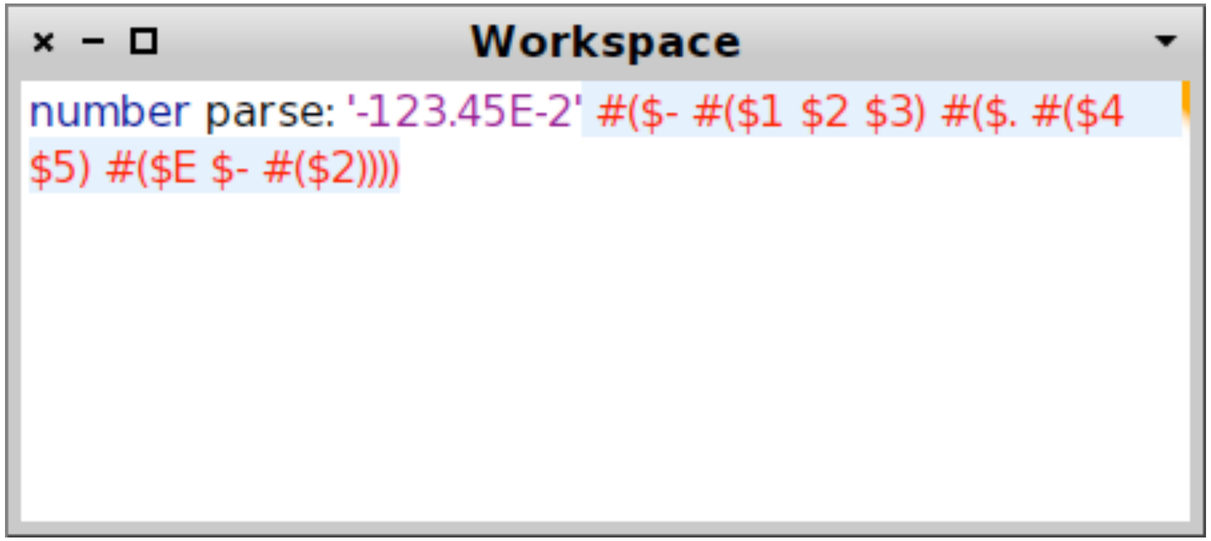
```
/ aParser
	"Answer a new parser that parses the receiver, if the receiver fails try with aParser (ordered-choice)."

	^ PPChoiceParser with: self with: aParser
```

exercise



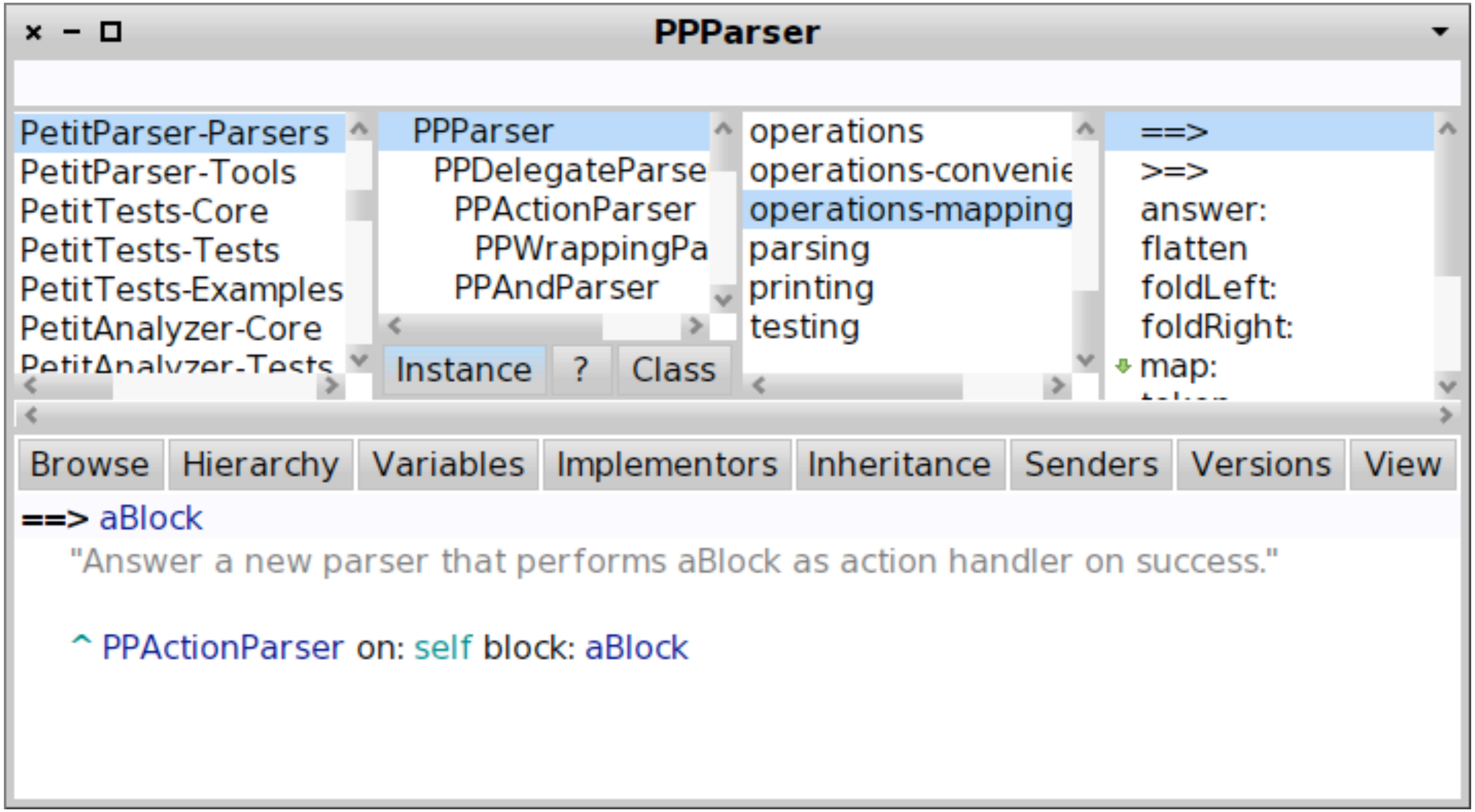
```
string parse: '''string''' #($' #($s $t $r $i $n $g) $')
```



```
number parse: '-123.45E-2' #($- #($1 $2 $3) #($. #($4
$5) #($E $- #($2))))
```

actions

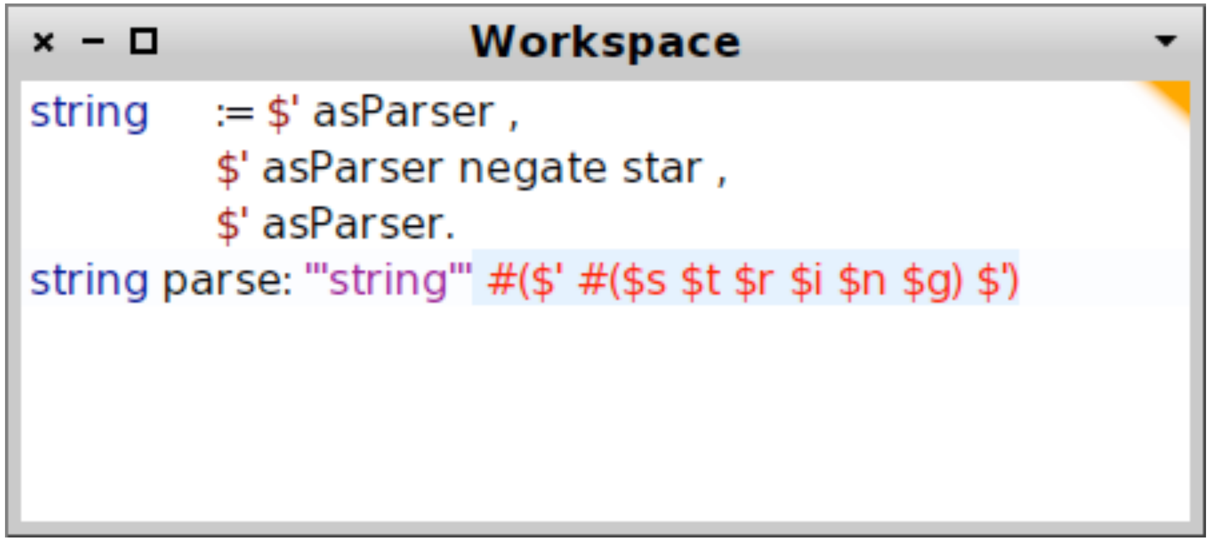
`p ==> aBlock`	Transforms the result of p through aBlock.
`p flatten`	Creates a string from the result of p.
`p token`	Creates a token from the result of p.
`p trim`	Trims whitespaces before and after p.



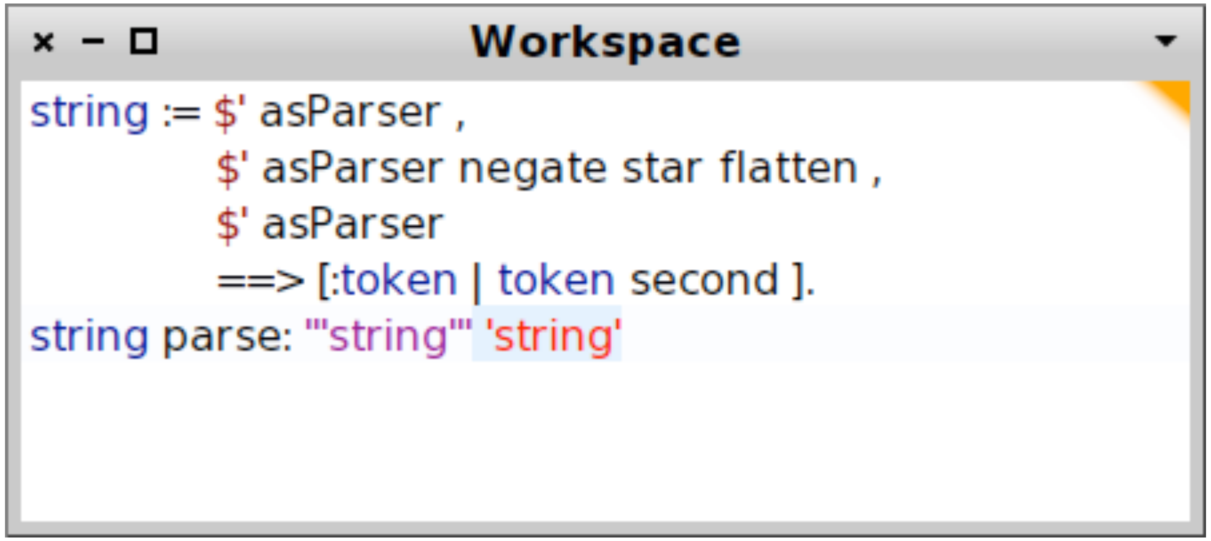
```
==> aBlock
    "Answer a new parser that performs aBlock as action handler on success."

    ^ PPActionParser on: self block: aBlock
```

```
string      := $' asParser ,
               $' asParser negate star ,
               $' asParser.
```



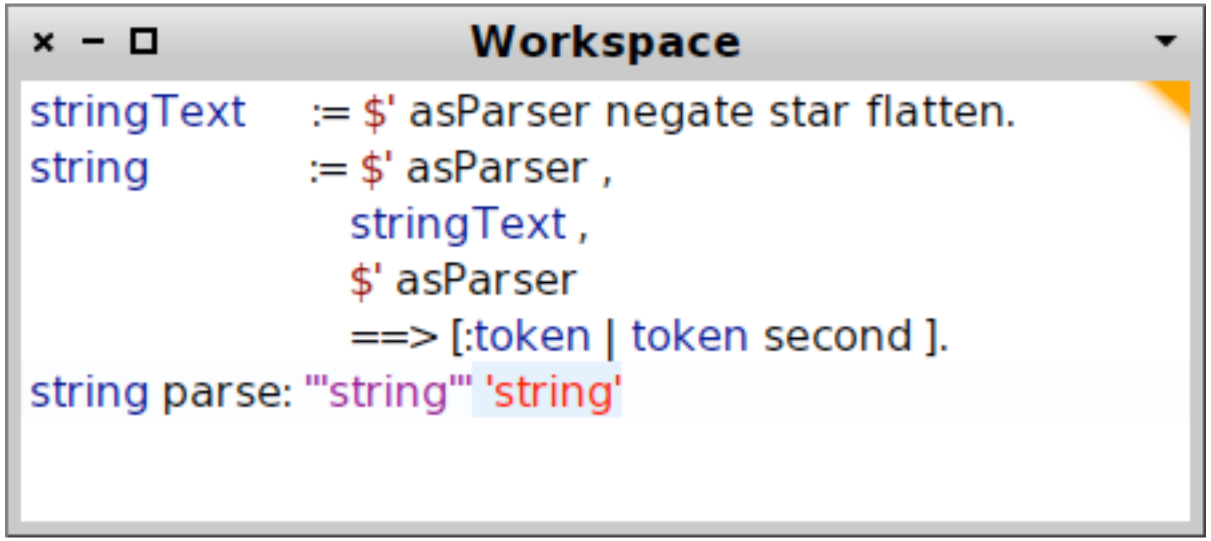
```
string    := $' asParser ,
             $' asParser negate star ,
             $' asParser.
string parse: '''string''' #($' #($s $t $r $i $n $g) $')
string      := $' asParser ,
             $' asParser negate star flatten ,
             $' asParser
             ==> [:token | token second ].
```



Workspace

```
string := $' asParser ,
          $' asParser negate star flatten ,
          $' asParser
          ==> [:token | token second ].
string parse: '''string''' 'string'
```

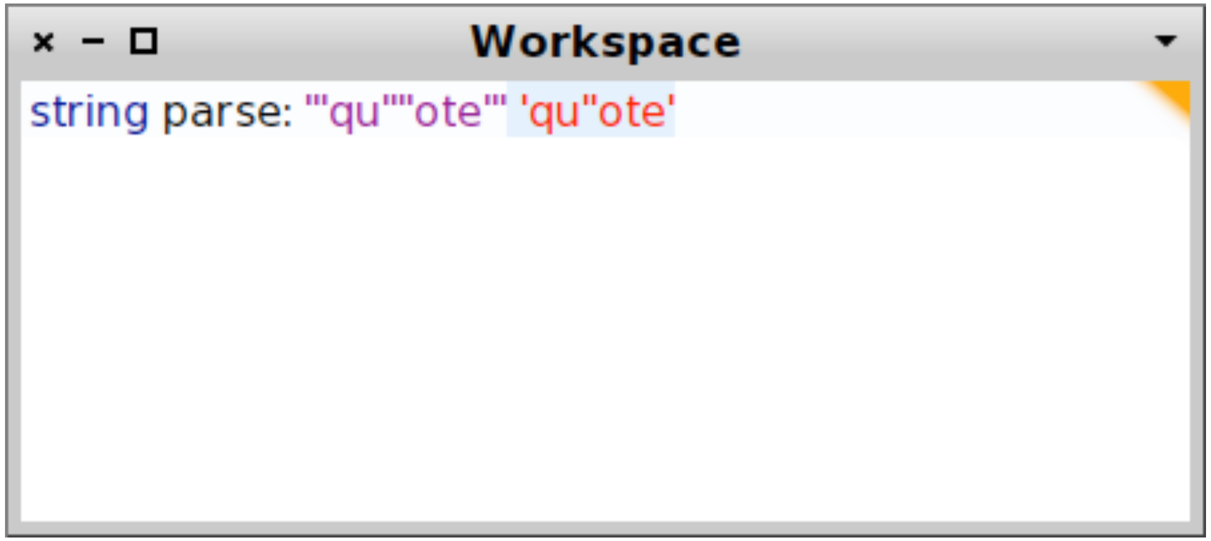
```
stringText := $' asParser negate star flatten.
string     := $' asParser ,
              stringText ,
              $' asParser
              ==> [:token | token second ].
```



Workspace

```
stringText    := $' asParser negate star flatten.
string        := $' asParser ,
                 stringText ,
                 $' asParser
                 ==> [:token | token second ].
string parse: '''string''' 'string'
```

exercise

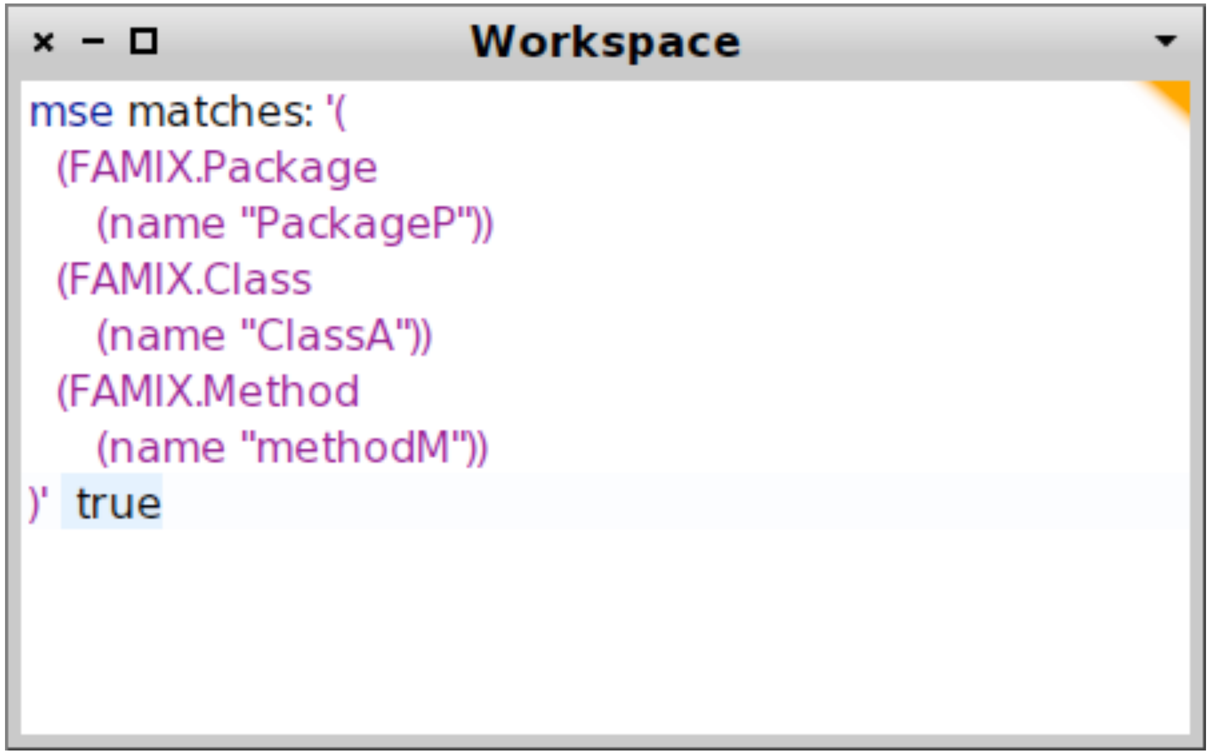


```
string parse: '''qu''''ote''' 'qu''ote'
```

exercise

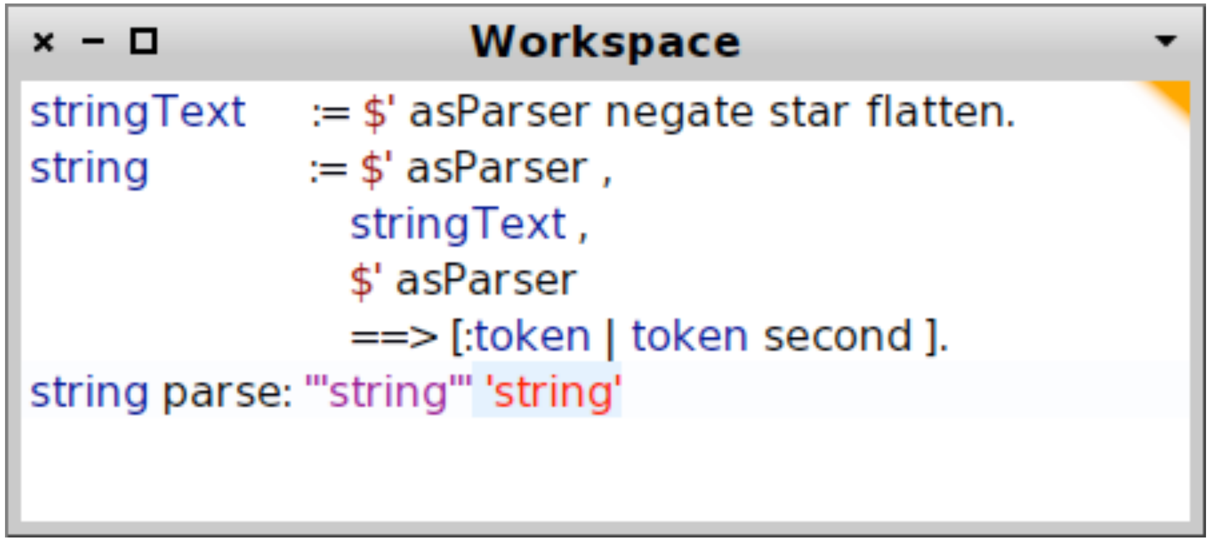
```
Root := Document ?
Document := OPEN ElementNode * CLOSE
ElementNode := OPEN ELEMENTNAME AttributeNode * CLOSE
AttributeNode := OPEN SIMPLENAME ValueNode * CLOSE
ValueNode := Primitive | ElementNode
Primitive := STRING | NUMBER
OPEN := "("
CLOSE := ")"
ELEMENTNAME := letter ( letter | digit ) * ( "." letter ( letter | digit ) *)
SIMPLENAME := letter ( letter | digit ) *
NUMBER := "-" ? digit + ( "." digit + ) ? ( ( "e" | "E" ) ( "-" | "+" ) ? digit + ) ?
STRING := ( "'" [^'] * "'" ) +
digit := [0-9]
letter := [a-zA-Z_]
comment := """ [^"] * """
```

```
(
  (FAMIX.Package
      (name 'PackageP'))
  (FAMIX.Class
      (name 'ClassA'))
  (FAMIX.Method
      (name 'methodM'))
)
```



```
mse matches: '(
  (FAMIX.Package
     (name "PackageP"))
  (FAMIX.Class
     (name "ClassA"))
  (FAMIX.Method
     (name "methodM"))
)' true
```

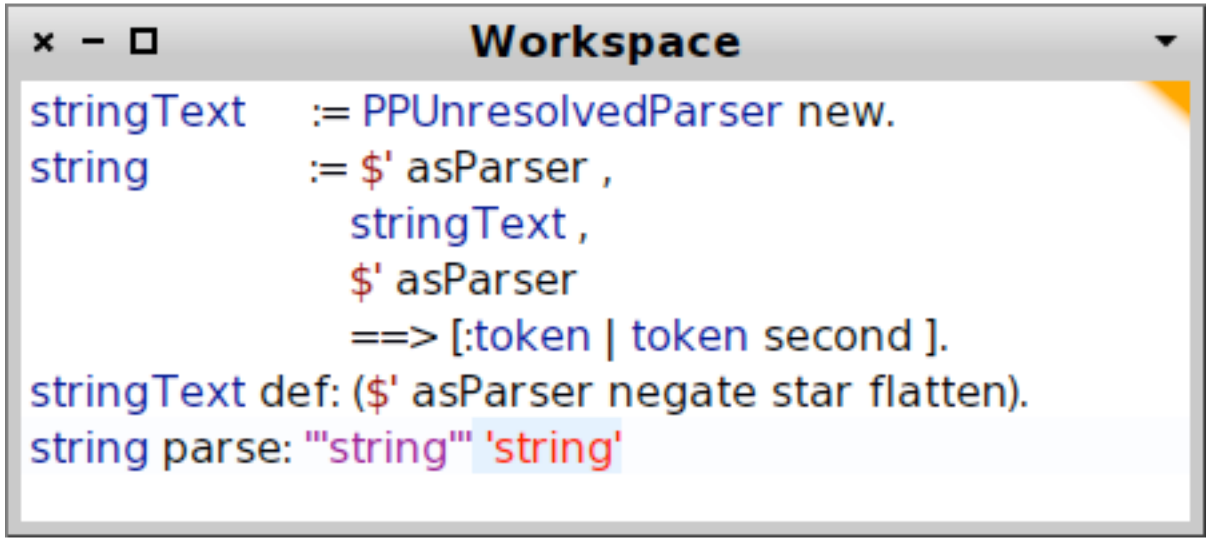
```
stringText    := $' asParser negate star flatten.
string        := $' asParser ,
                 stringText ,
                 $' asParser
                 ==> [:token | token second ].
```



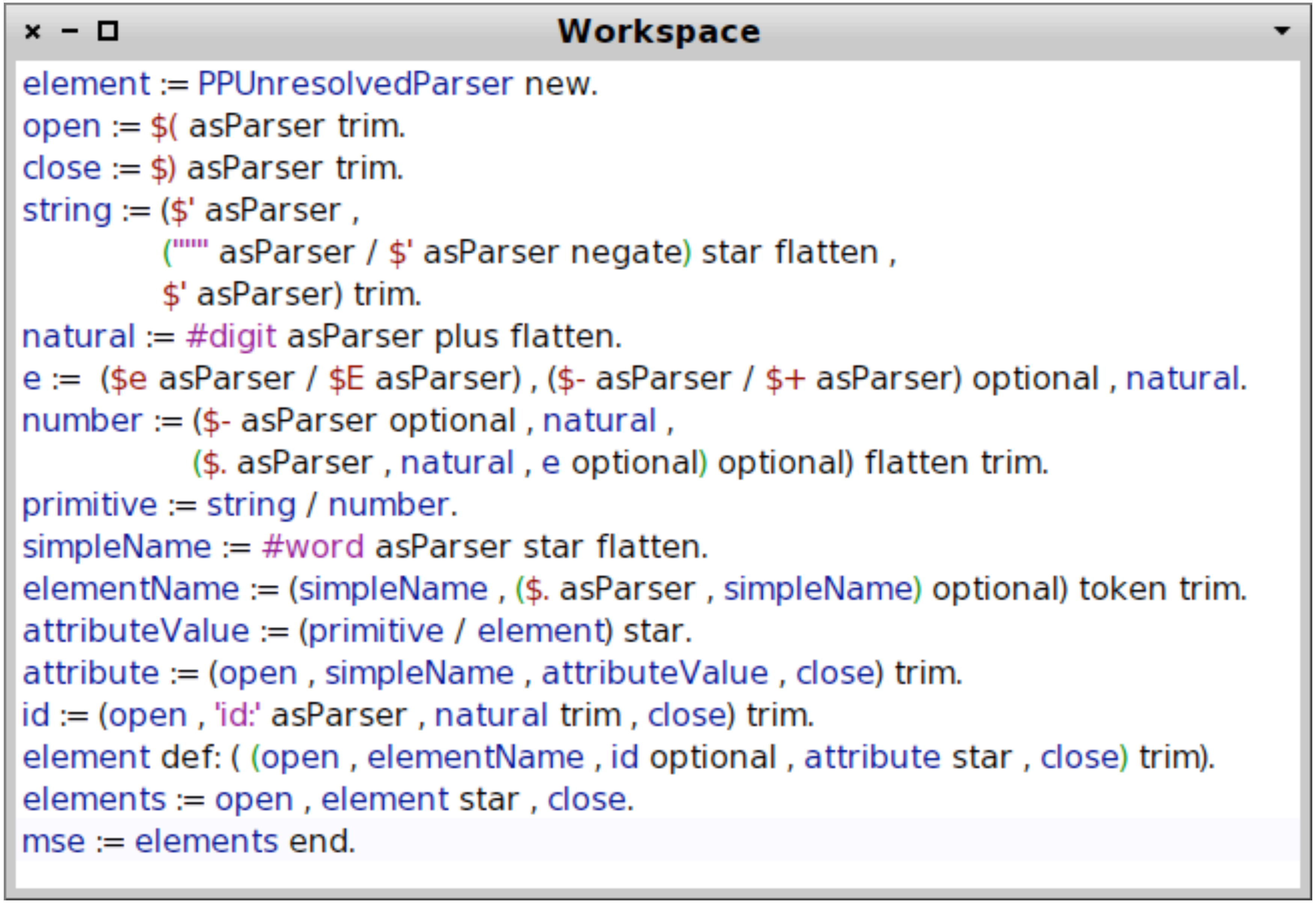
Workspace

```
stringText    := $' asParser negate star flatten.
string        := $' asParser ,
                 stringText ,
                 $' asParser
                 ==> [:token | token second ].
string parse: '''string''' 'string'
```

```
stringText     := PPUnresolvedParser new.
string         := $' asParser ,
                  stringText ,
                  $' asParser
                  ==> [:token | token second ].
stringText def: ($' asParser negate star flatten).
```



```
stringText     := PPUnresolvedParser new.
string         := $' asParser ,
                  stringText ,
                  $' asParser
                  ==> [:token | token second ].
stringText def: ($' asParser negate star flatten).
string parse: '''string''' 'string'
```



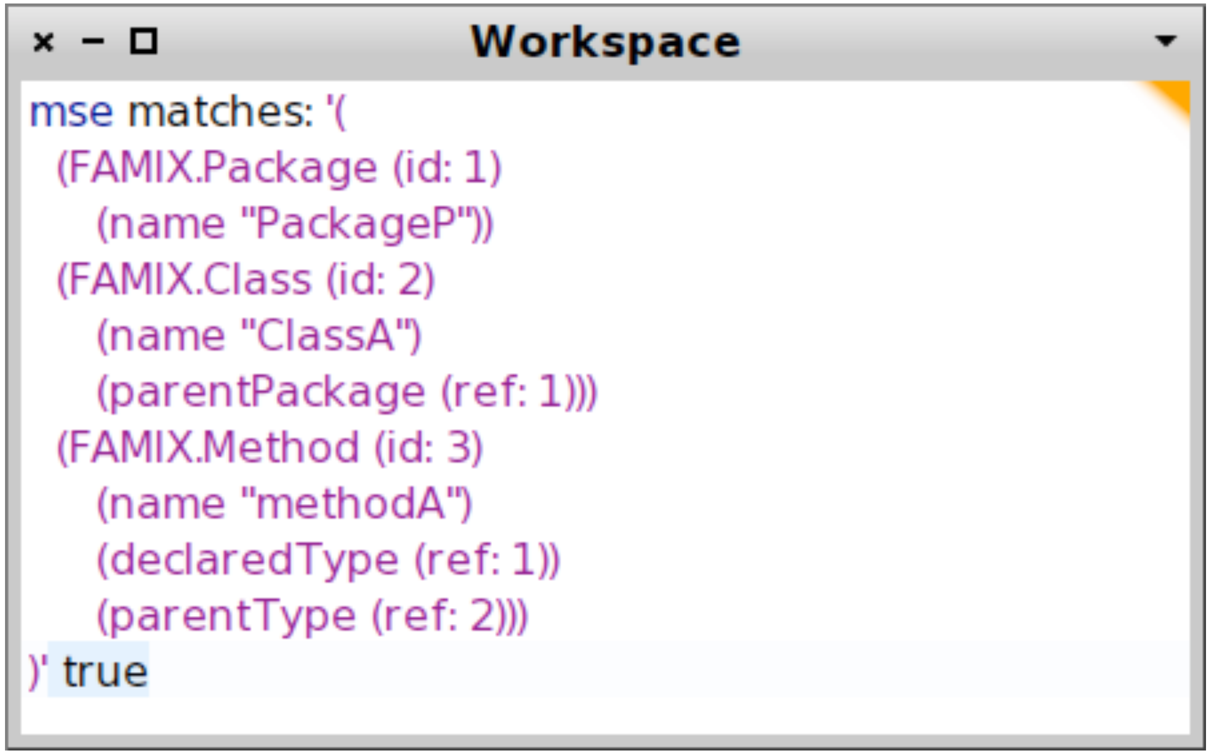
```
element := PPUnresolvedParser new.
open := $( asParser trim.
close := $) asParser trim.
string := ($' asParser ,
            ('''' asParser / $' asParser negate) star flatten ,
            $' asParser) trim.
natural := #digit asParser plus flatten.
e :=  ($e asParser / $E asParser) , ($- asParser / $+ asParser) optional , natural.
number := ($- asParser optional , natural ,
              ($. asParser , natural , e optional) optional) flatten trim.
primitive := string / number.
simpleName := #word asParser star flatten.
elementName := (simpleName , ($. asParser , simpleName) optional) token trim.
attributeValue := (primitive / element) star.
attribute := (open , simpleName , attributeValue , close) trim.
id := (open , 'id:' asParser , natural trim , close) trim.
element def: ( (open , elementName , id optional , attribute star , close) trim).
elements := open , element star , close.
mse := elements end.
```

exercise

```
Root := Document ?
Document := OPEN ElementNode * CLOSE
ElementNode := OPEN ELEMENTNAME Serial ? AttributeNode * CLOSE
Serial := OPEN ID INTEGER CLOSE
AttributeNode := OPEN SIMPLENAME ValueNode * CLOSE
ValueNode := Primitive | Reference | ElementNode
Primitive := STRING | NUMBER | Boolean | Unlimited
Boolean := TRUE | FALSE
Unlimited := NIL
Reference := IntegerReference | NameReference
IntegerReference := OPEN REF INTEGER CLOSE
NameReference := OPEN REF ELEMENTNAME CLOSE
OPEN := "("
CLOSE := ")"
ID := "id:"
REF := "ref:"
TRUE := "true"
FALSE := "false"
ELEMENTNAME := letter ( letter | digit ) * ( "." letter ( letter | digit ) ) *
SIMPLENAME := letter ( letter | digit ) *
INTEGER := digit +
NUMBER := "-" ? digit + ( "." digit + ) ? ( ( "e" | "E" ) ( "-" | "+" ) ? digit + ) ?
STRING := ( "'" [^'] * "'" ) +
digit := [0-9]
letter := [a-zA-Z_]
comment := """ [^"] * """
```

exercise

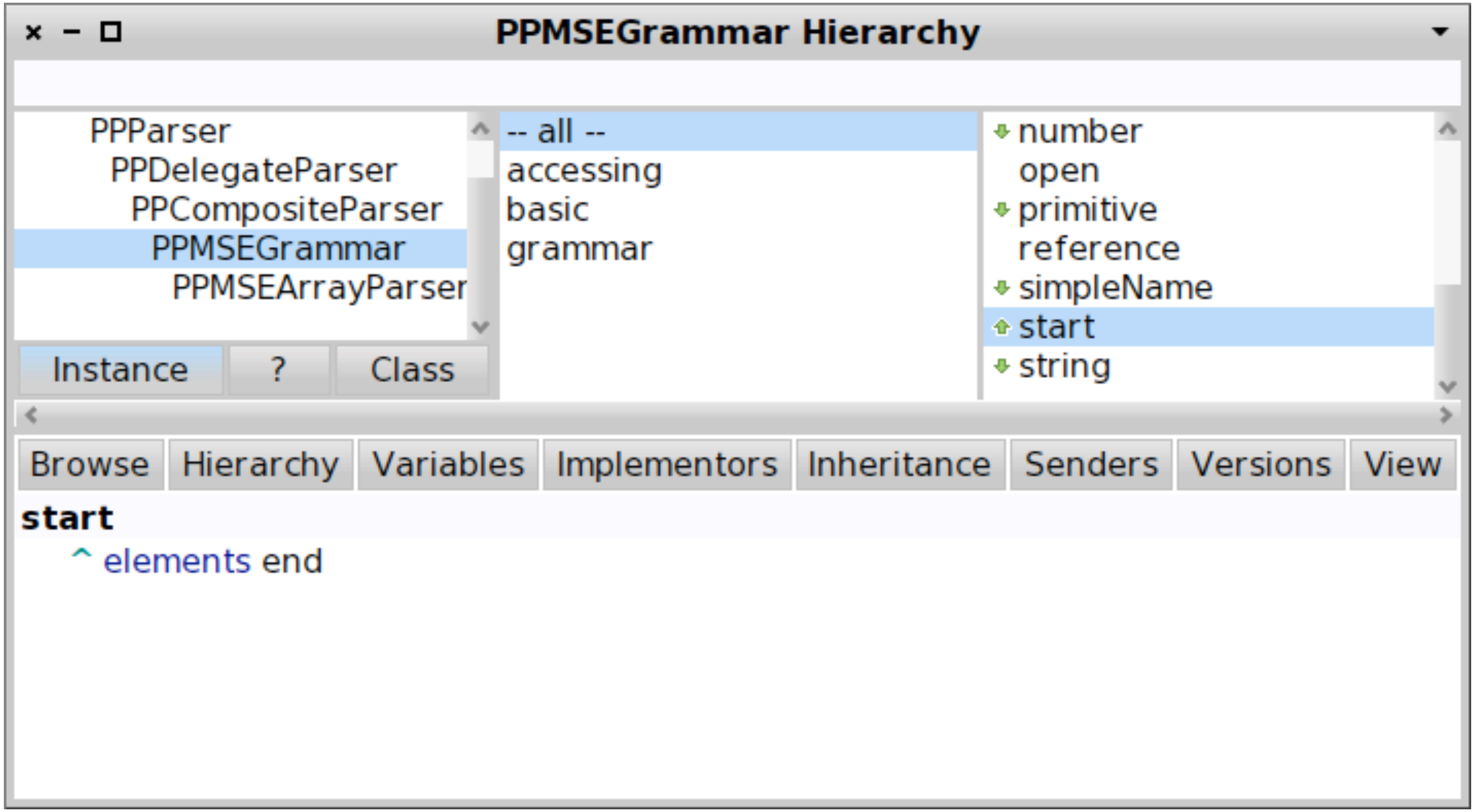
```
(
  (FAMIX.Package (id: 1)
     (name 'PackageP'))
  (FAMIX.Class (id: 2)
     (name 'ClassA')
     (parentPackage (ref: 1)))
  (FAMIX.Method (id: 3)
     (name 'methodA')
     (declaredType (ref: 1))
     (parentType (ref: 2)))
)
```



```
mse matches: '(
  (FAMIX.Package (id: 1)
     (name "PackageP"))
  (FAMIX.Class (id: 2)
     (name "ClassA")
     (parentPackage (ref: 1)))
  (FAMIX.Method (id: 3)
     (name "methodA")
     (declaredType (ref: 1))
     (parentType (ref: 2)))
)' true
```

scripting = nice for prototyping
but messy

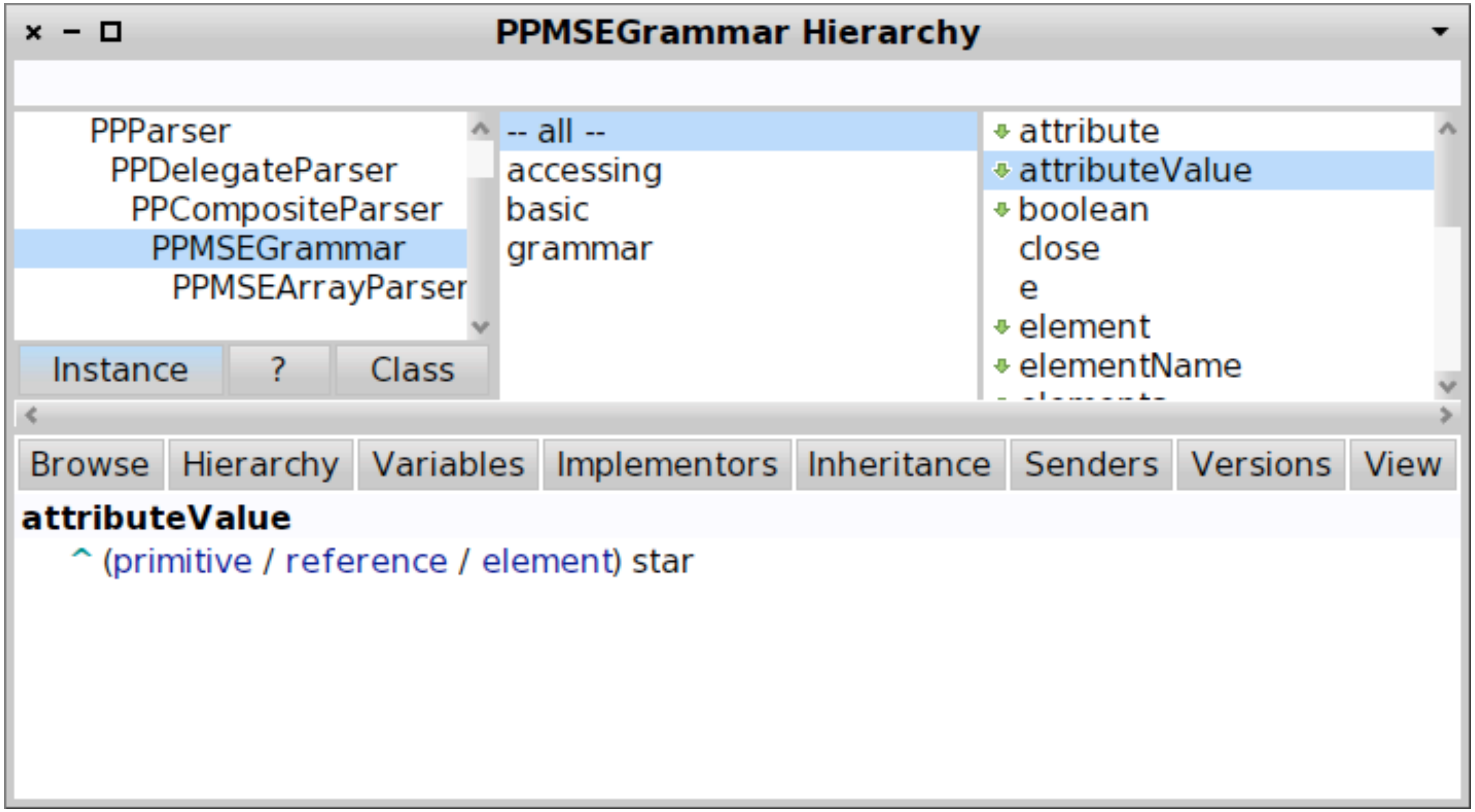
subclass PPCompositeParser



```
start
	^ elements end
```

start = default start parser

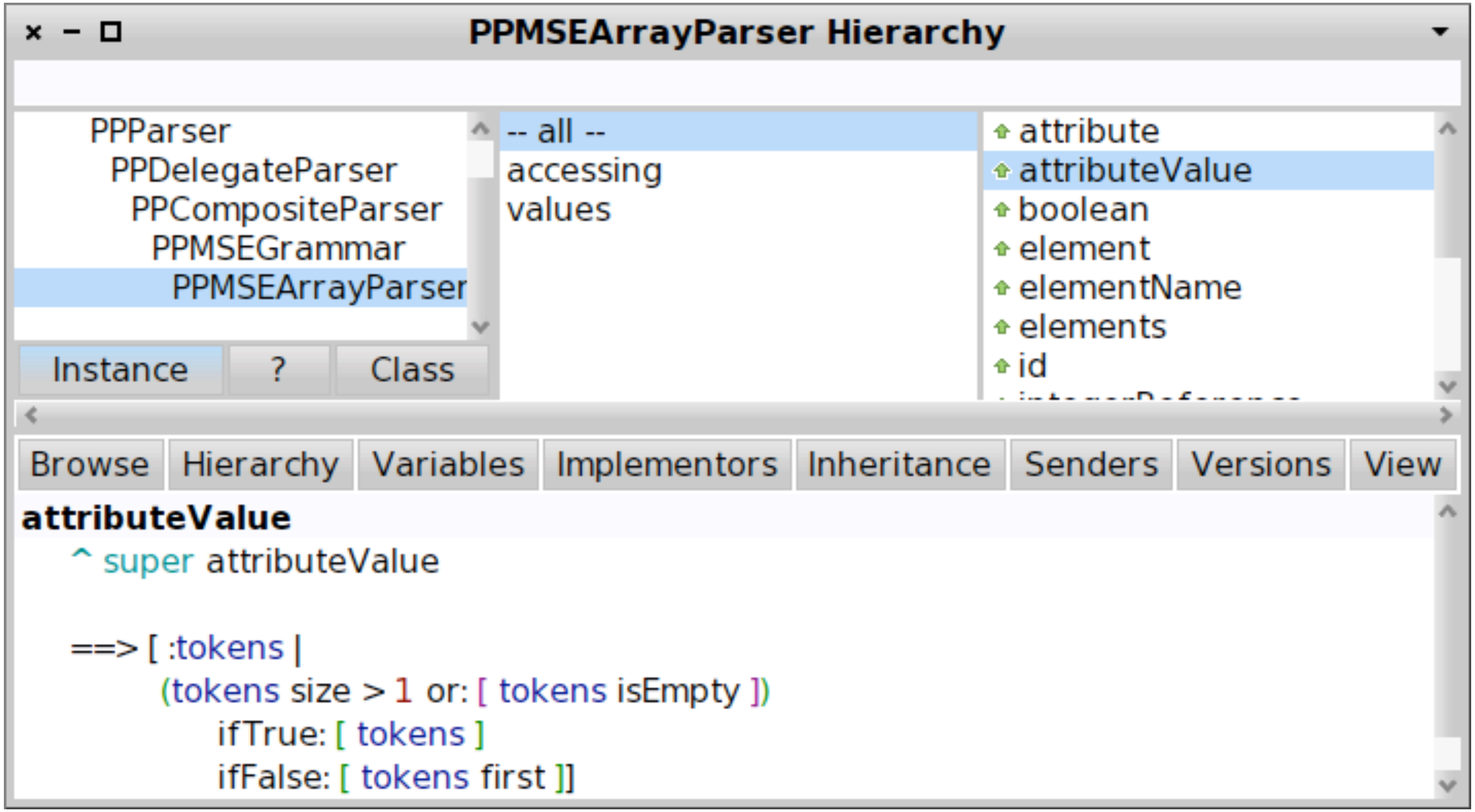
externally, parsers map on methods



```
attributeValue
	^ (primitive / reference / element) star
```

internally, parsers map on instance variables

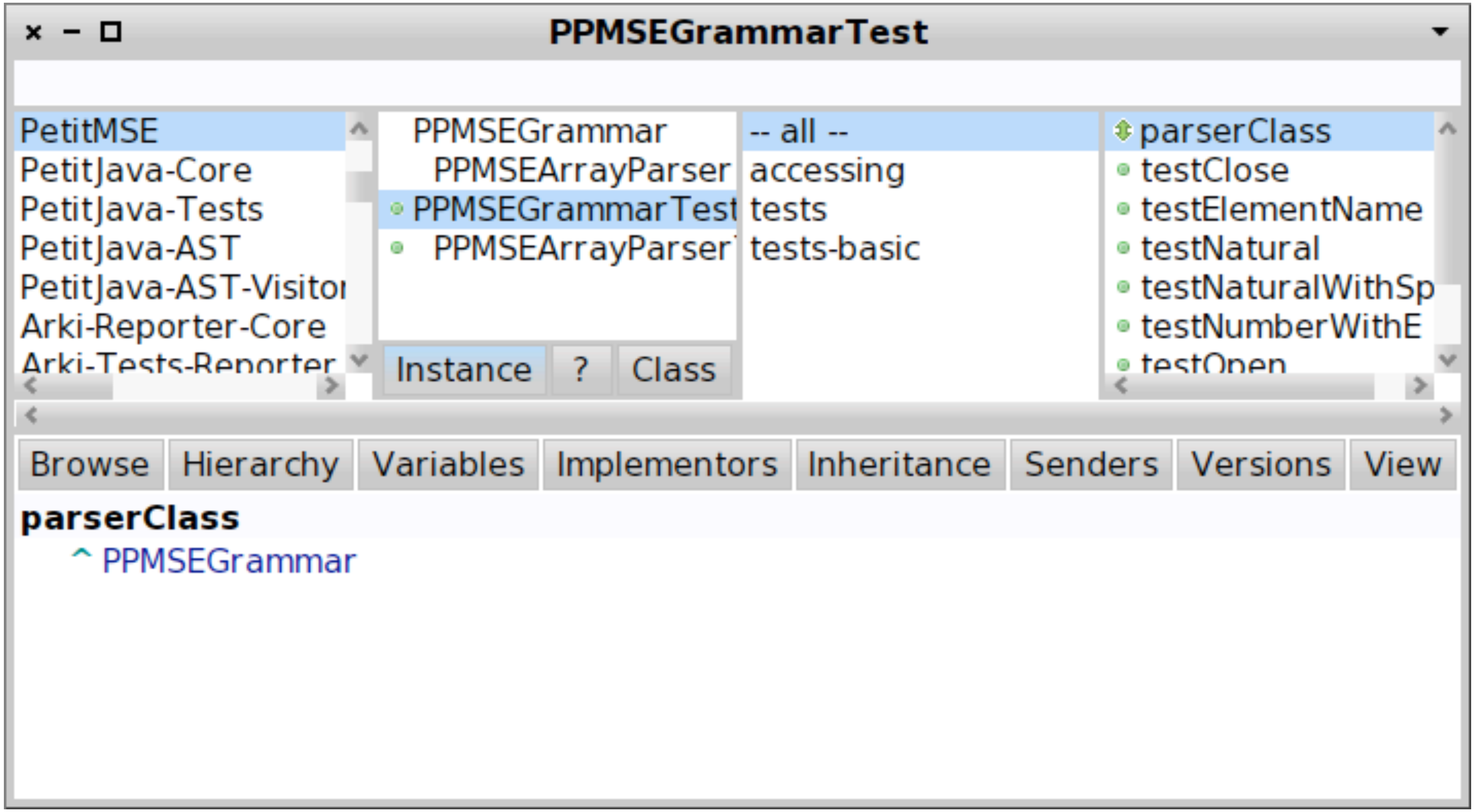
to specify actions, subclass the base grammar



```
attributeValue
    ^ super attributeValue

    ==> [ :tokens |
        (tokens size > 1 or: [ tokens isEmpty ])
            ifTrue: [ tokens ]
            ifFalse: [ tokens first ]]
```

subclass tests from PPCompositeParserTest


PPMSEGrammarTest

PetitMSE
PetitJava-Core
PetitJava-Tests
PetitJava-AST
PetitJava-AST-Visitor
Arki-Reporter-Core
Arki-Tests-Reporter

PPMSEGrammar
PPMSEArrayParser
PPMSEGrammarTest
PPMSEArrayParser

Instance | ? | Class

-- all --
accessing
tests
tests-basic

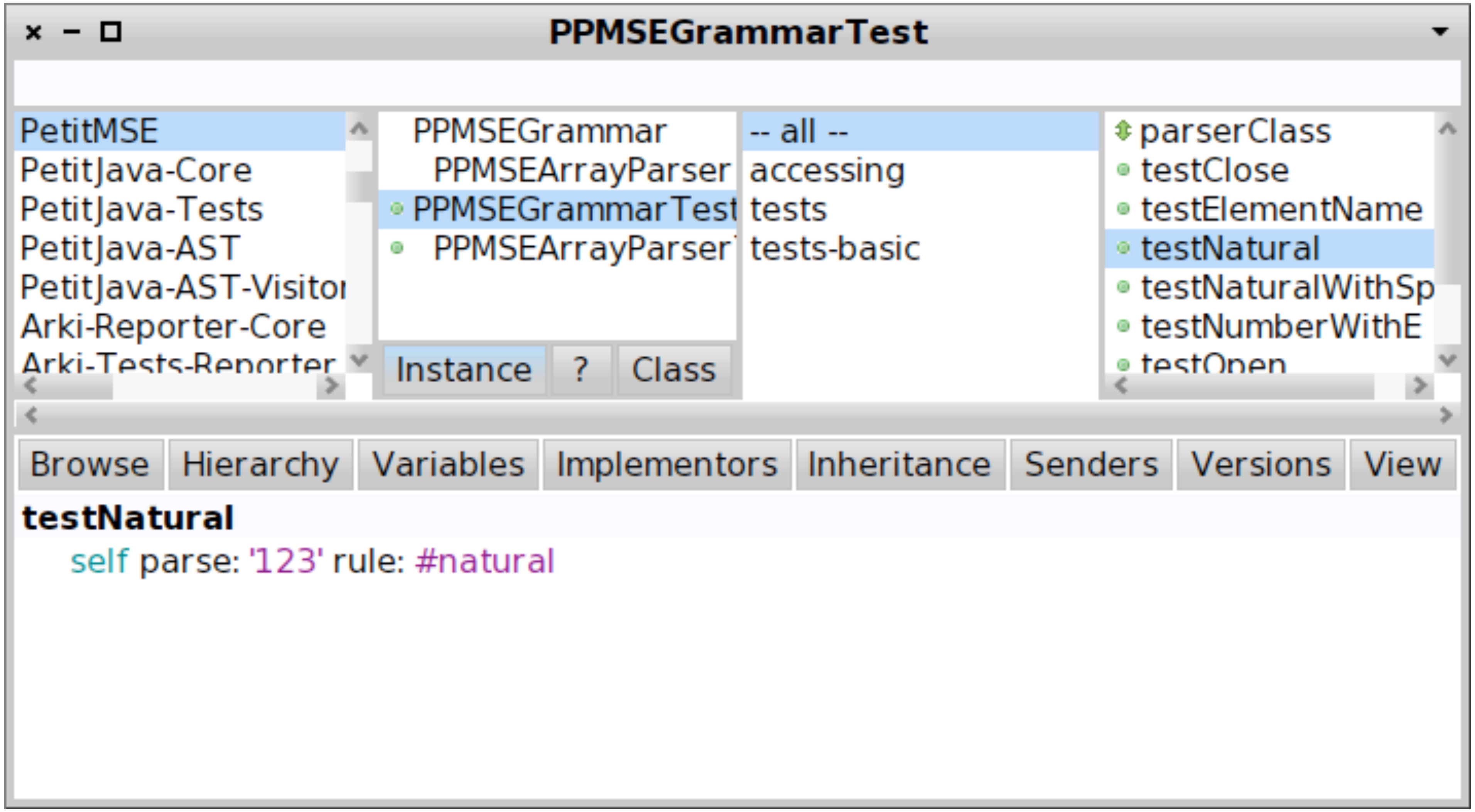
parserClass
testClose
testElementName
testNatural
testNaturalWithSp
testNumberWithE
testOpen

Browse | Hierarchy | Variables | Implementors | Inheritance | Senders | Versions | View

```
parserClass
    ^ PPMSEGrammar
```

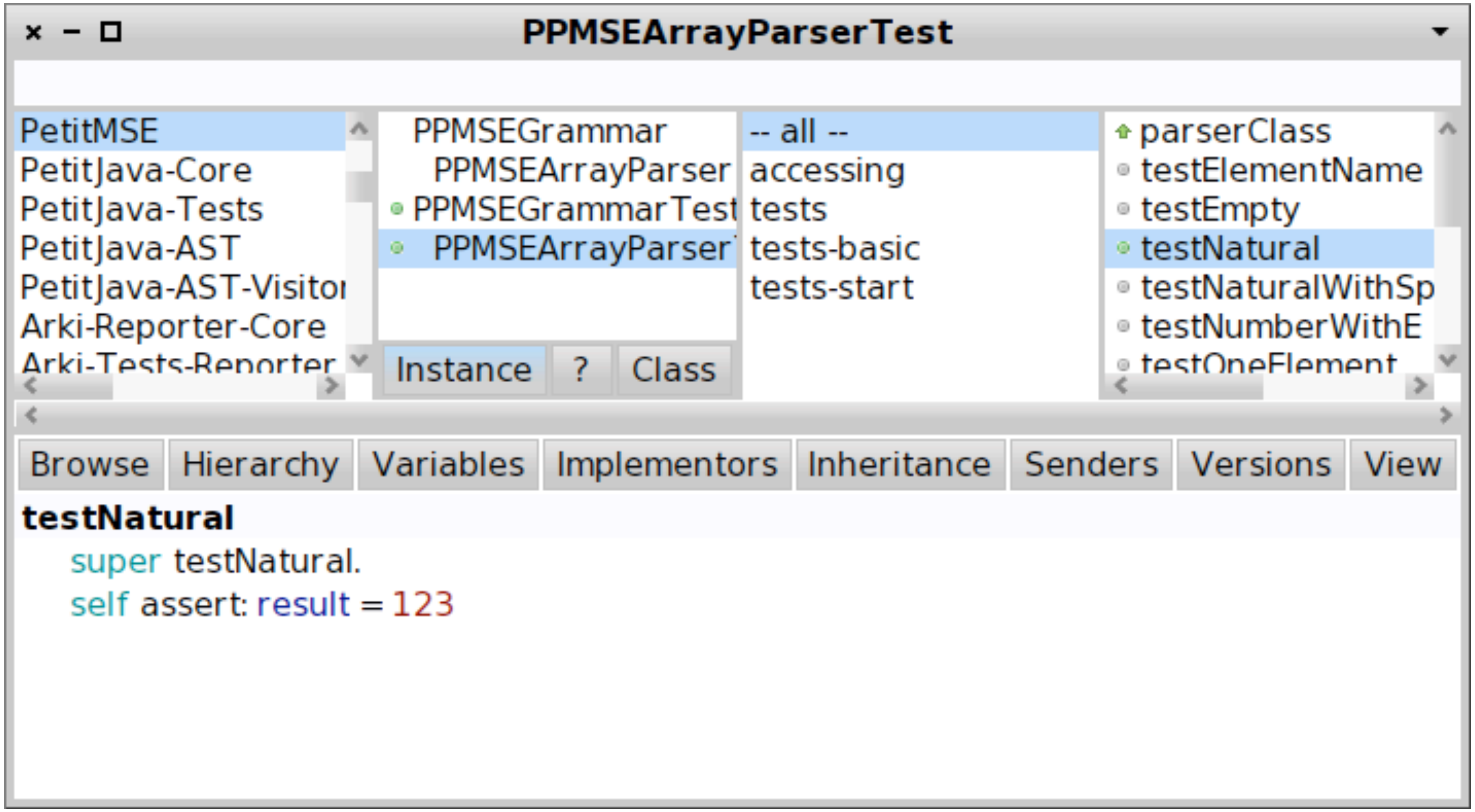
specify the parserClass

use #parse:rule: to check the grammar



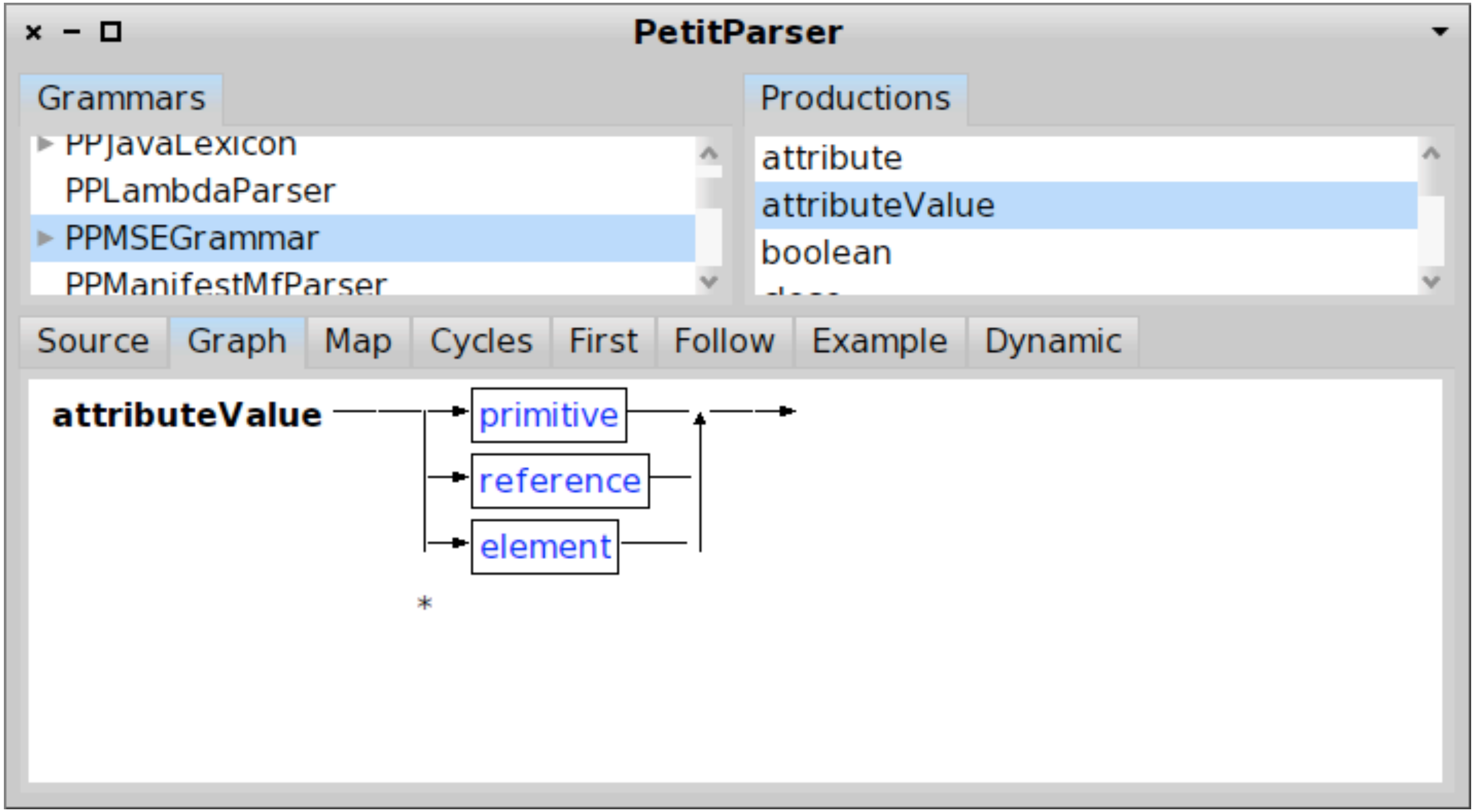
```
testNatural
    self parse: '123' rule: #natural
```

subclass to check the parser result



```
testNatural
    super testNatural.
    self assert: result = 123
```

PetitParser comes with a dedicated user interface



PPBrowser open.

PetitParser

built by Lukas Renggli
deeply integrated with Smalltalk
part of the Moose Suite

Tudor Gîrba

www.tudorgirba.com