

Petit *Parser*

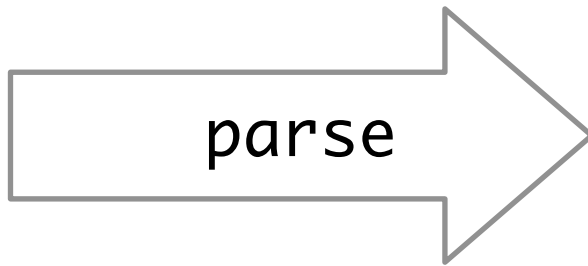
www.tudorgirba.com

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

Petit *Parser*

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

input



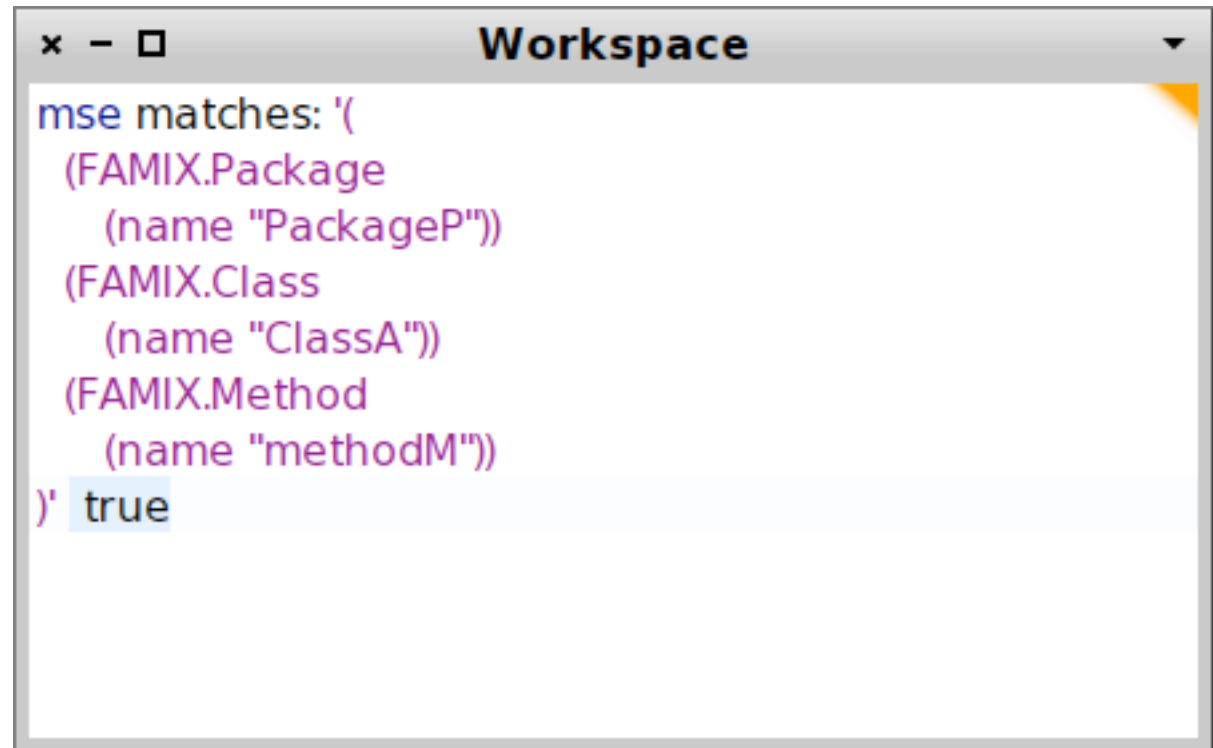
output

The logo features the text "Petit Parser" centered within a large, hollow arrow pointing to the right. The arrow is outlined in a dark gray color. The word "Petit" is rendered in a thin, blue, sans-serif font. The word "Parser" is rendered in a bold, dark blue, sans-serif font with a slight shadow effect behind it.

Petit **Parser**

```
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



The screenshot shows a window titled "Workspace" with a light gray title bar. The main content area is white and displays a match result in a structured, indented format. The text is as follows:

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

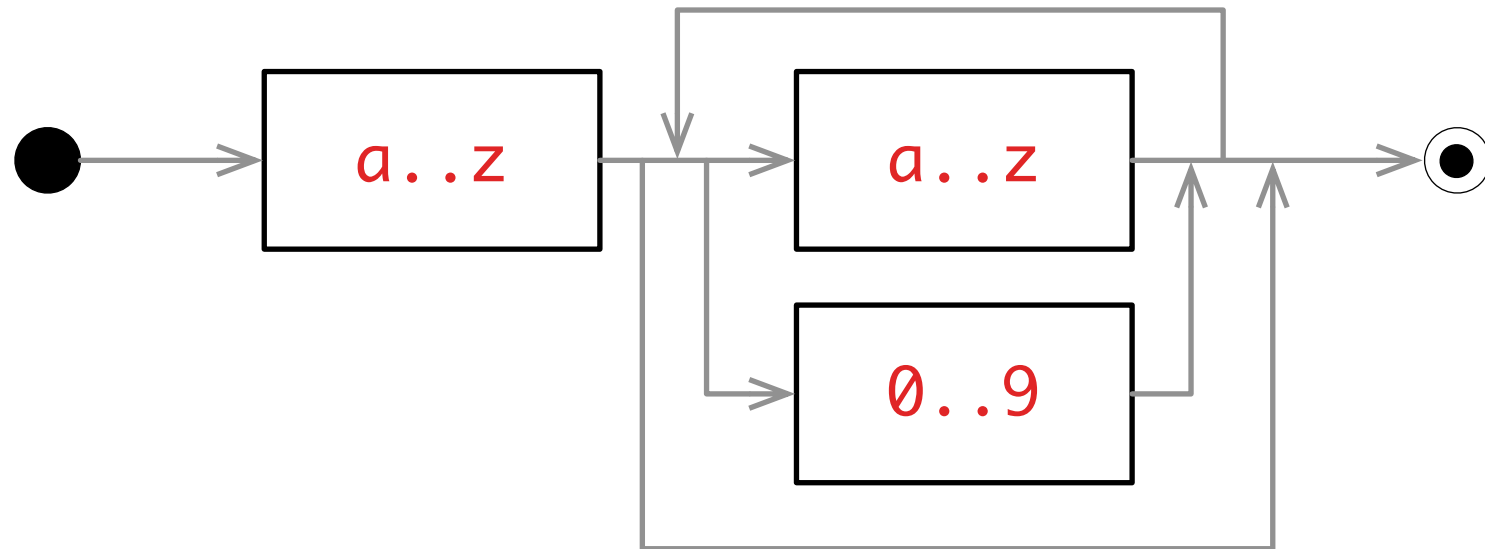
The text is color-coded: "mse matches:" is blue, the opening brace '{' is blue, and the closing brace '}' is blue. The word "true" is blue. The opening and closing parentheses of the match are blue. The opening and closing parentheses of the package, class, and method are purple. The names "PackageP", "ClassA", and "methodM" are purple. The word "true" is blue.

x - □

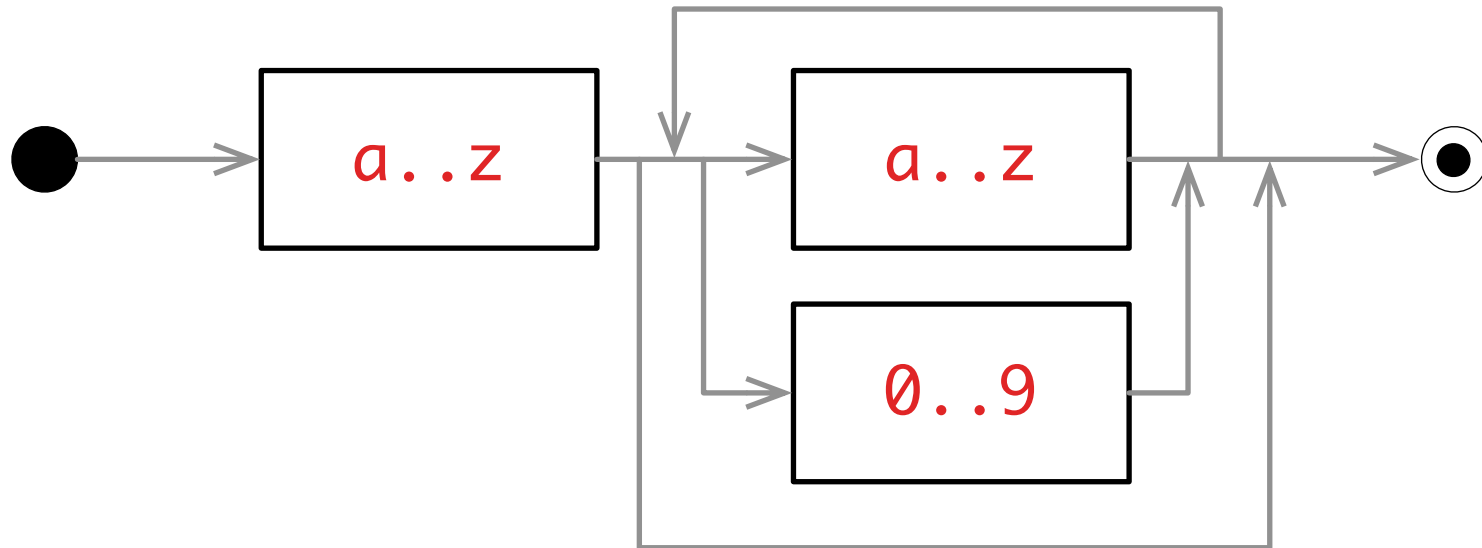
Workspace

```
element := PPU unresolvedParser 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) *



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



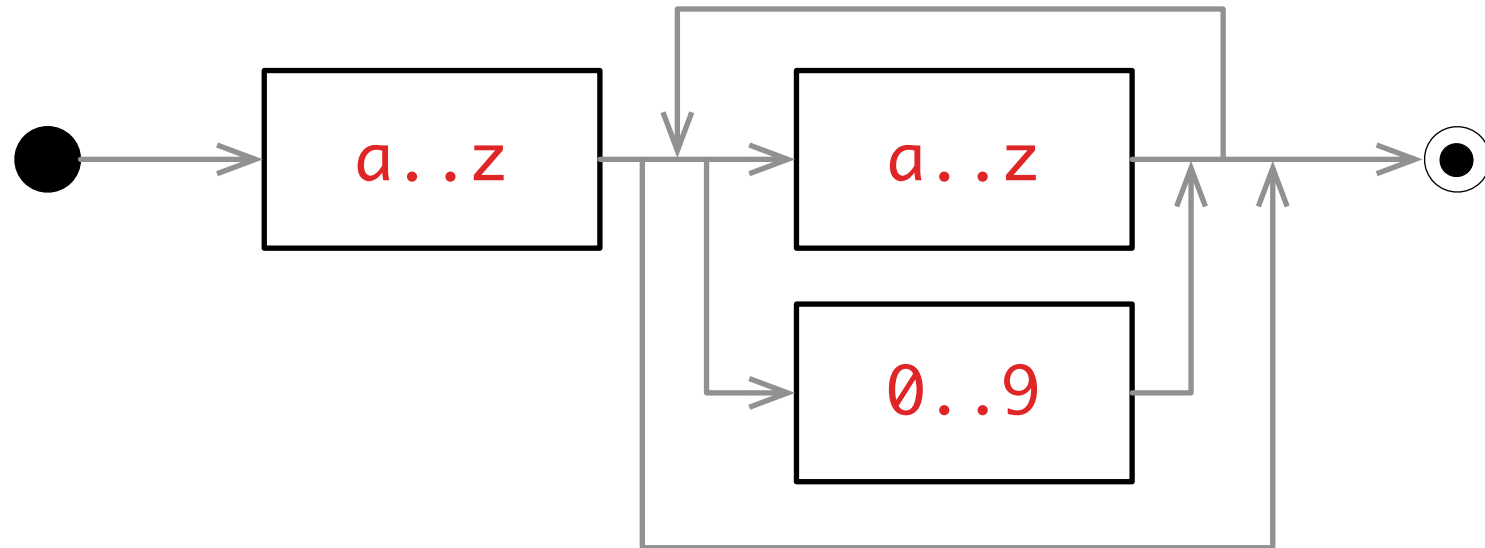
exercise

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

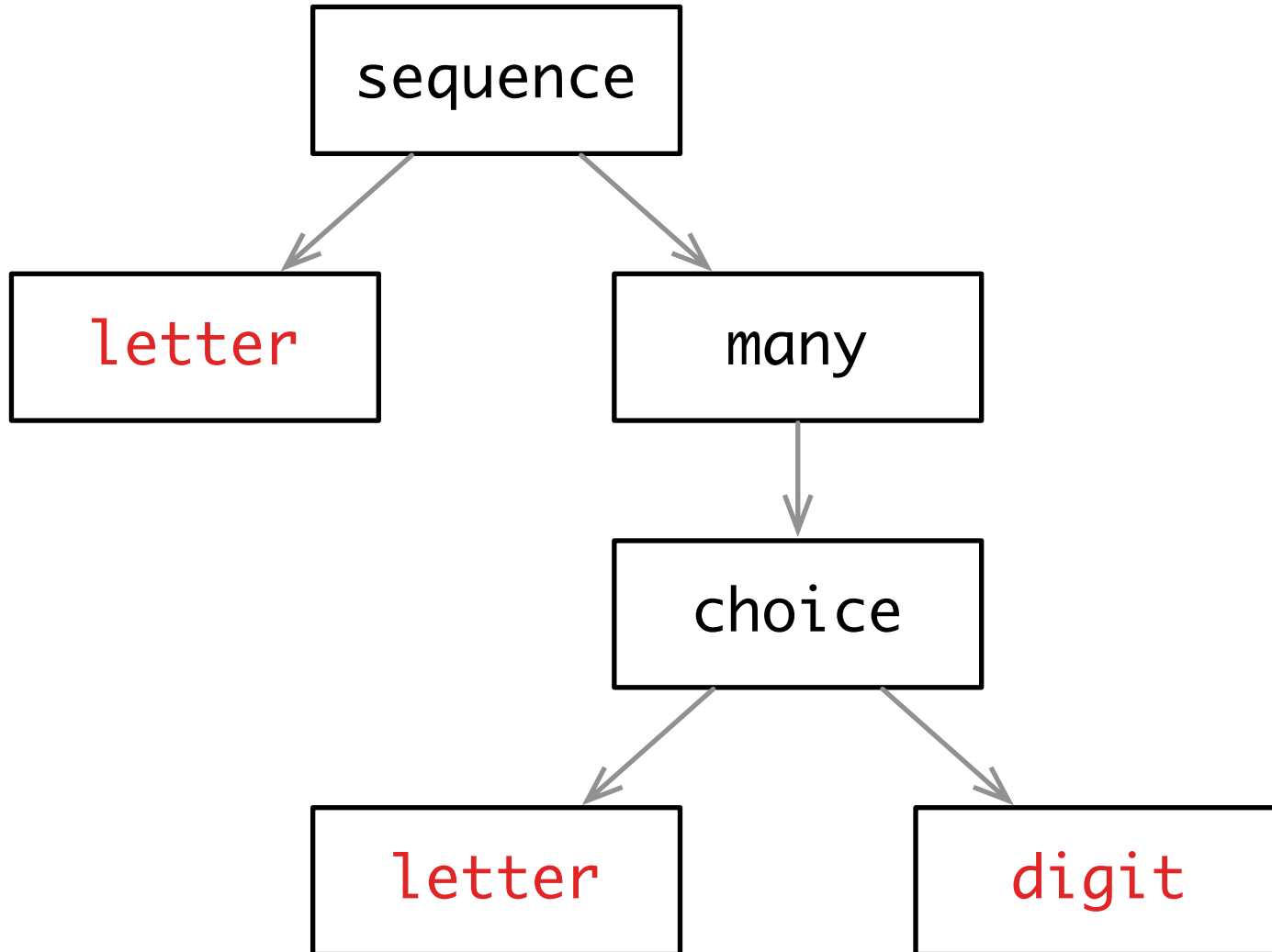
```
x - □ Workspace
identifier := #letter asParser ,
            ( #letter asParser /
              #digit asParser ) star.
identifier parse: 'valid2' #($v #($a $l $i $d $2))
```

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

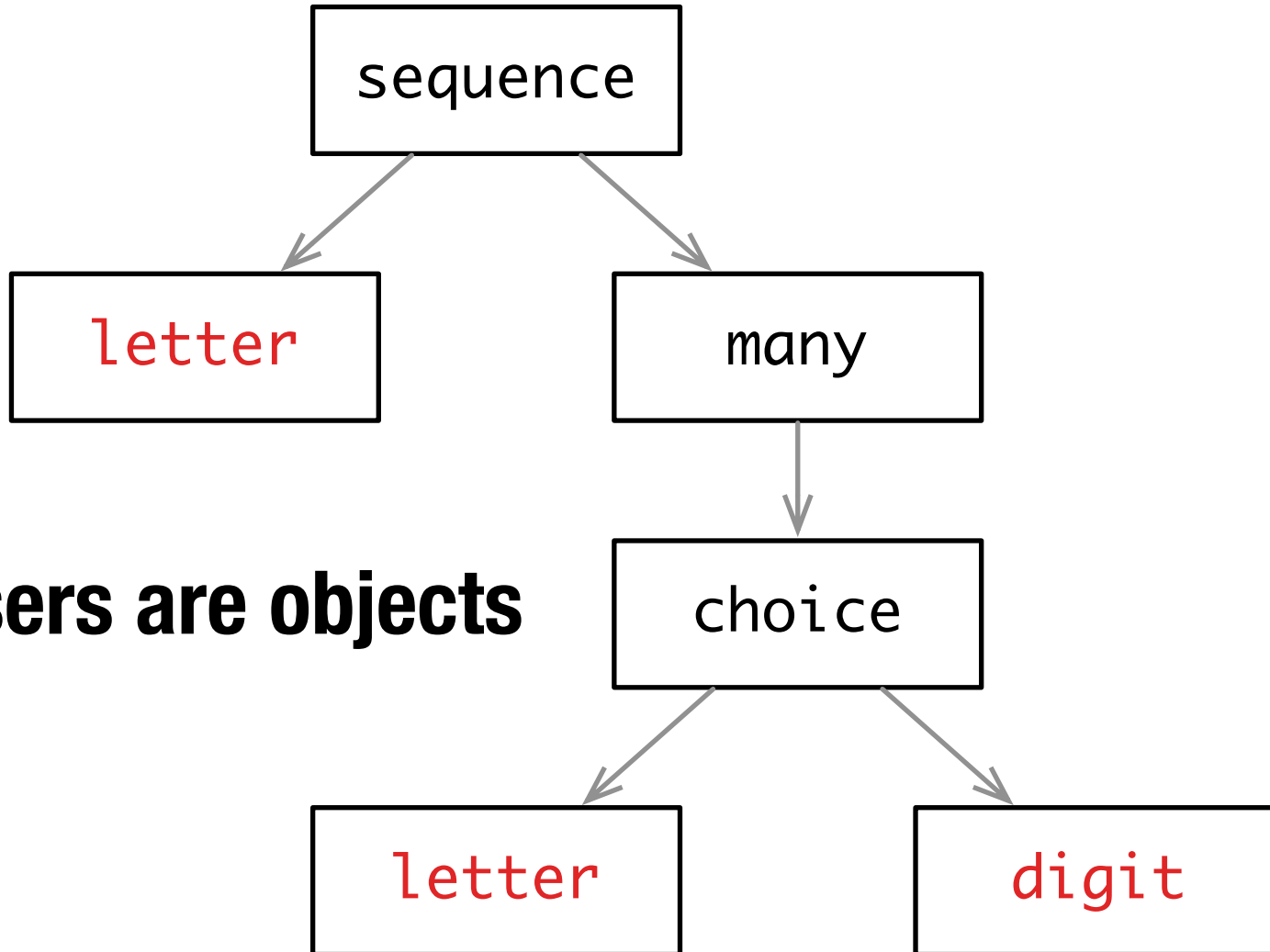
identifrier := #letter asParser ,
(#letter asParser /
#digit asParser) star.



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



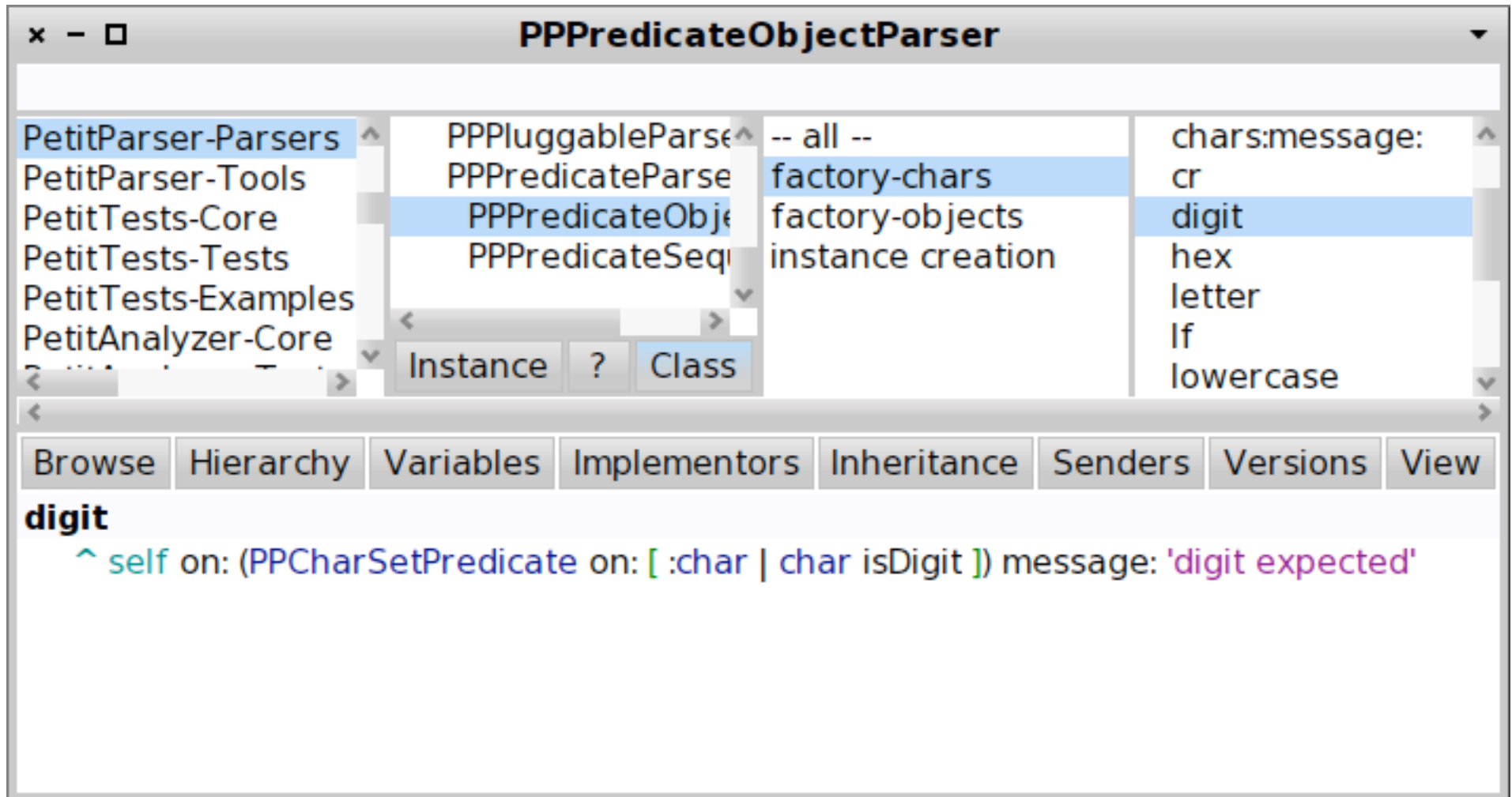
```
identifier := #letter asParser ,  
            ( #letter asParser /  
              #digit asParser ) star.
```



terminals

<code>\$c</code>	<code>asParser</code>	parse character "c"
<code>'string'</code>	<code>asParser</code>	parse string "string"
<code>#any</code>	<code>asParser</code>	parse any character
<code>#digit</code>	<code>asParser</code>	parse one digit
<code>#letter</code>	<code>asParser</code>	parse one letter

terminals are defined in PPPredicateObjectParser



The screenshot shows an IDE window titled "PPPredicateObjectParser". The class hierarchy is displayed in a tree view on the left, with "PPPredicateObjectParser" selected. The right pane shows the implementation of the "digit" class, which is a subclass of "PPCharSetPredicate". The implementation is as follows:

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

exercise: browse PPPredicateObjectParser

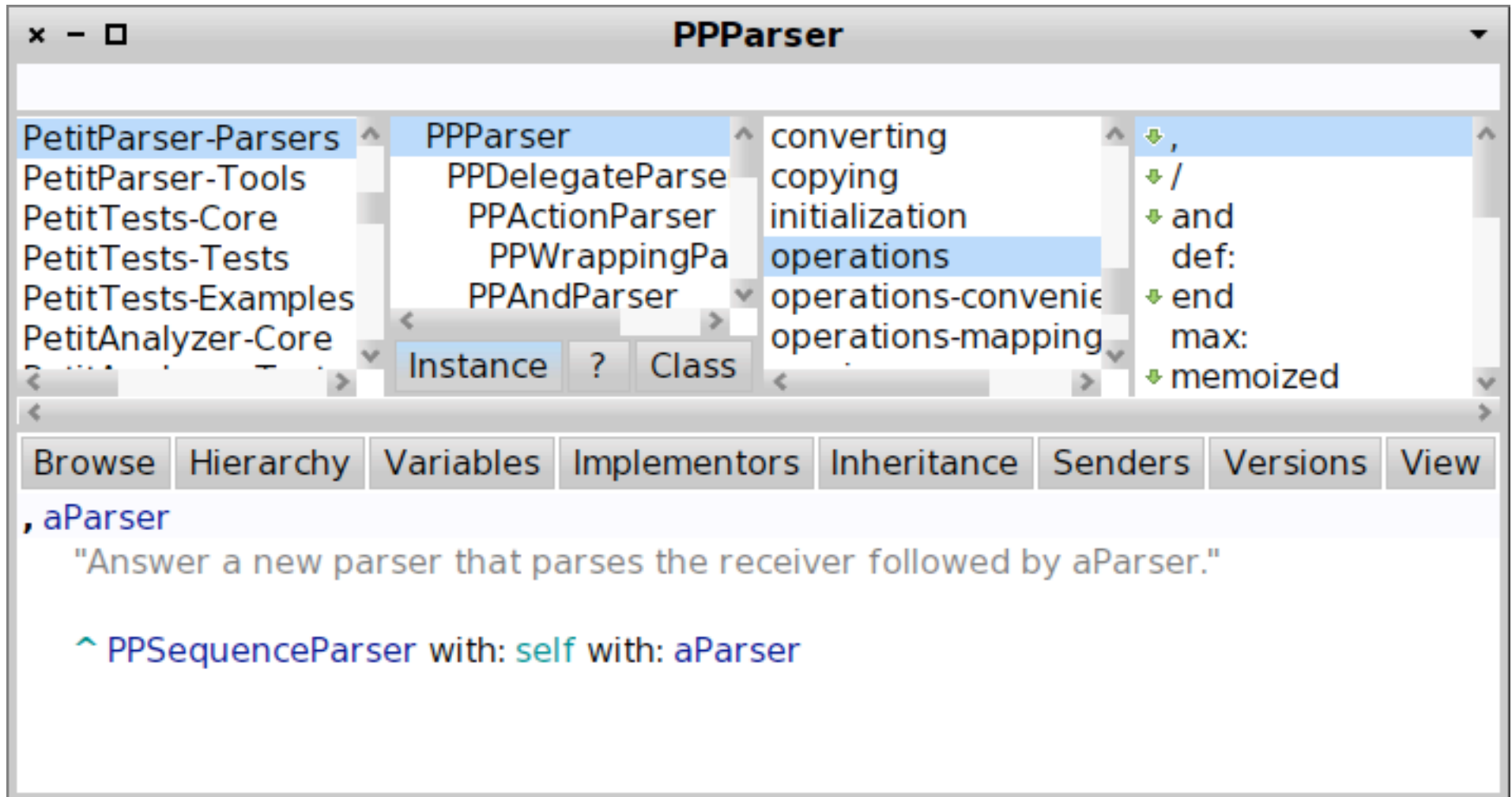
combinators

p_1 , p_2	parse p_1 followed by p_2 (sequence)
p_1 / p_2	parse p_1 , otherwise parse p_2 (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



The screenshot shows the Ruby IDE interface for the `PPParser` class. The class hierarchy is visible on the left, with `PPParser` selected. The `operations` category is expanded, showing methods such as `and`, `def:`, `end`, `max:`, and `memoized`. The `Browse` tab is active, displaying the documentation for `aParser`:

```
, 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

The screenshot shows a class browser window titled "PPParser". The left sidebar lists several packages, with "PetitParser-Parsers" selected. The main area shows the class hierarchy for "PPParser", including subclasses like "PPDelegateParse", "PPActionParser", "PPWrappingPa", and "PPAndParser". The "Instance" and "Class" buttons are visible. The right pane shows a list of methods, with "operations" selected. Below the browser, a row of tabs includes "Browse", "Hierarchy", "Variables", "Implementors", "Inheritance", "Senders", "Versions", and "View". The "Browse" tab is active, displaying the documentation for the "aParser" method: "Answer a new parser that parses the receiver, if the receiver fails try with aParser (ordered-choice)." and a callout for "PPChoiceParser with: self with: aParser".

PPParser

PetitParser-Parsers
PetitParser-Tools
PetitTests-Core
PetitTests-Tests
PetitTests-Examples
PetitAnalyzer-Core

PPParser
PPDelegateParse
PPActionParser
PPWrappingPa
PPAndParser

converting
copying
initialization
operations
operations-convenie
operations-mapping

,
/
and
def:
end
max:
memoized

Instance ? Class

Browse Hierarchy Variables Implementors Inheritance Senders Versions View

/ aParser

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

^ PPChoiceParser with: self with: aParser

exercise

```
x - □ Workspace
string parse: "string" #('$' #($s $t $r $i $n $g) $')
```

```
x - □ Workspace
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.

PPParser

PetitParser-Parsers
PetitParser-Tools
PetitTests-Core
PetitTests-Tests
PetitTests-Examples
PetitAnalyzer-Core
PetitAnalyzer-Tests

PPParser
PPDelegateParse
PPActionParser
PPWrappingPa
PPAndParser

operations
operations-convenie
operations-mapping
parsing
printing
testing

==>
>=>
answer:
flatten
foldLeft:
foldRight:
map:

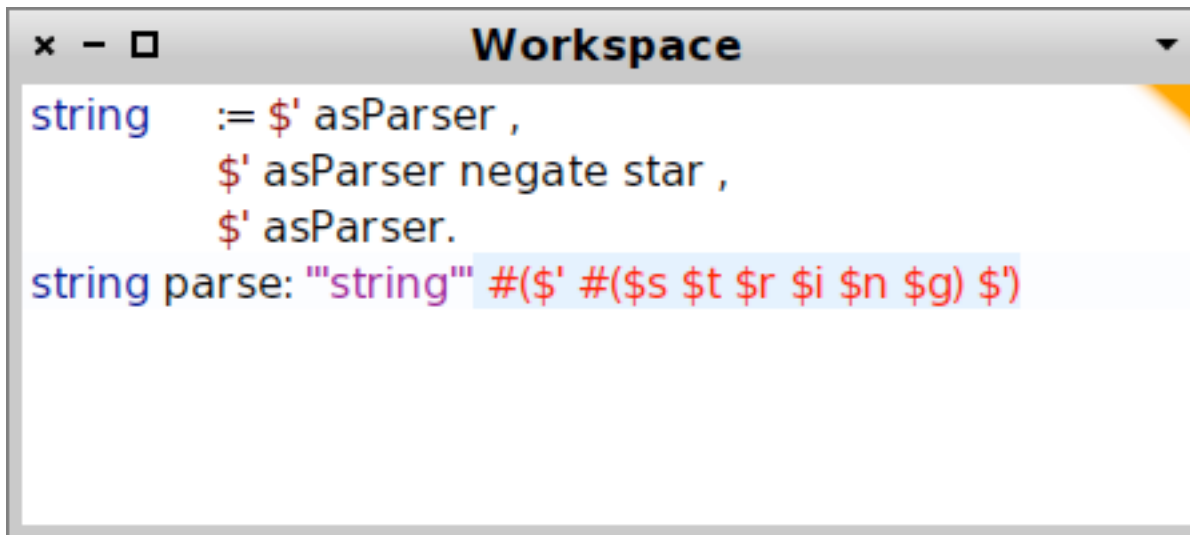
Instance ? Class

Browse Hierarchy Variables Implementors Inheritance Senders Versions View

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

^ PPActionParser on: self block: aBlock

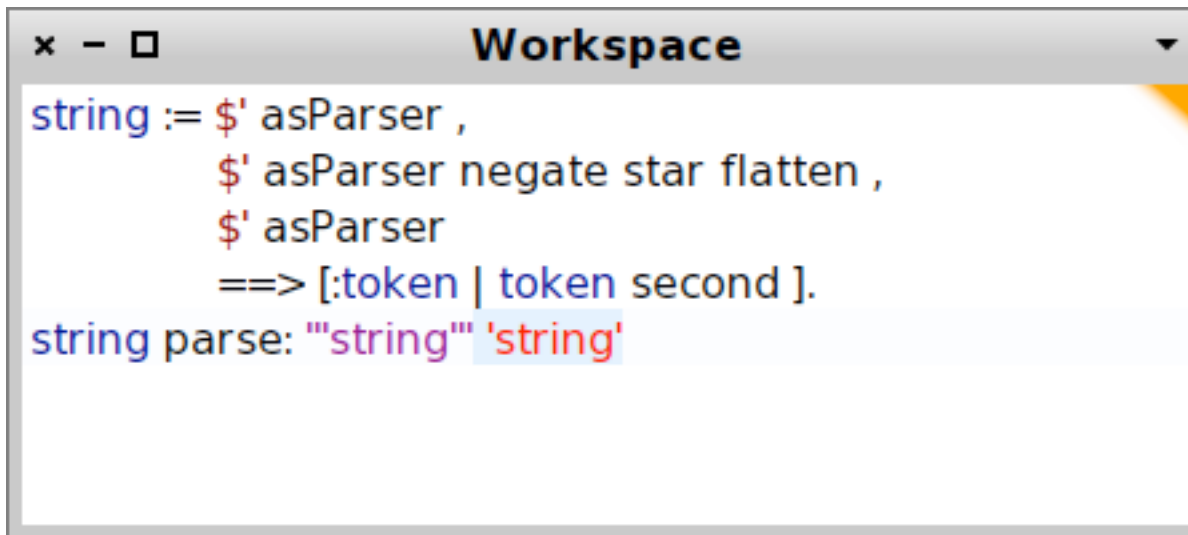
```
string      := '$' asParser ,  
            '$' asParser negate star ,  
            '$' asParser.
```



The screenshot shows a window titled "Workspace" with a close button (x), a maximize button (-), and a refresh button (□). The window contains the following code:

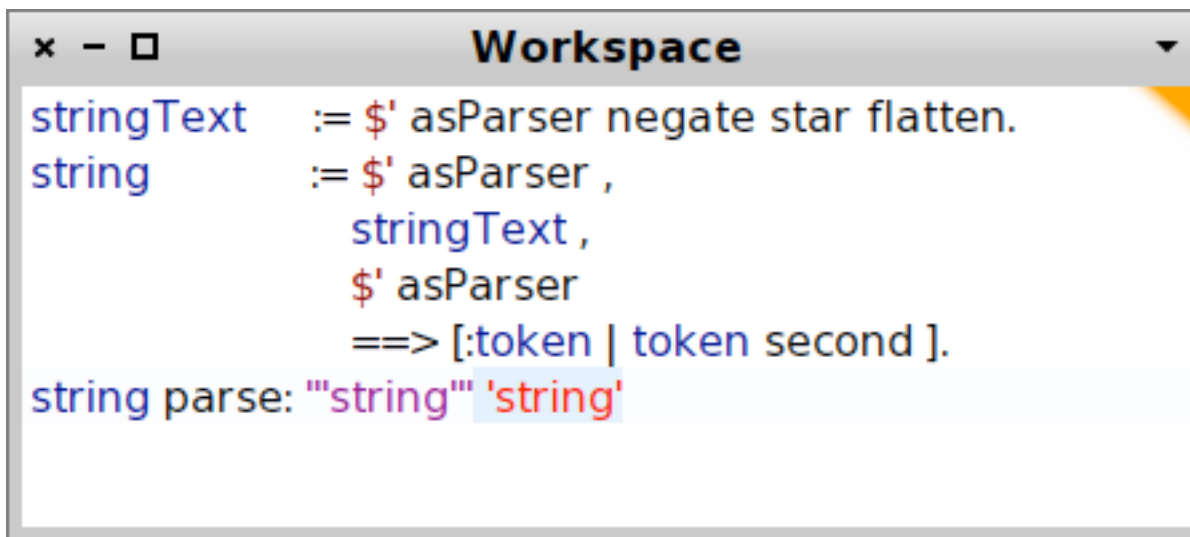
```
string      := '$' asParser ,  
            '$' asParser negate star ,  
            '$' asParser.  
string parse: "string" #('$' #('$s $t $r $i $n $g) $')
```

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



```
x - □ Workspace  
string := '$' asParser ,  
         '$' asParser negate star flatten ,  
         '$' asParser  
         ==> [ :token | token second ].  
string parse: "string" 'string'
```

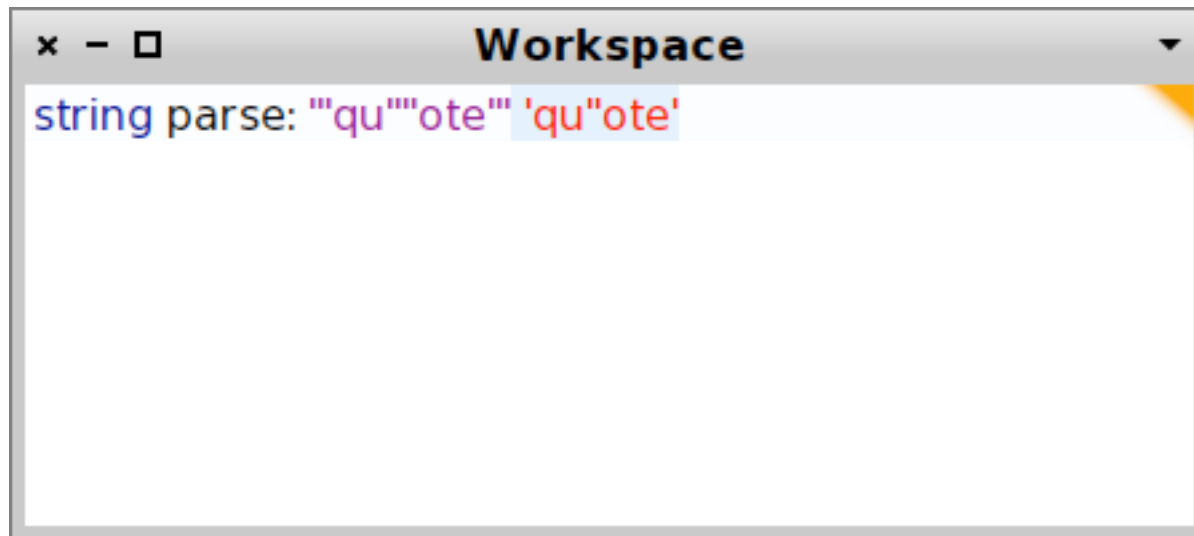
```
stringText := '$' asParser negate star flatten.  
string     := '$' asParser ,  
            stringText ,  
            '$' asParser  
            ==> [ :token | token second ].
```



The screenshot shows a workspace window titled "Workspace" with a close button (x) and a maximize button (□). The code from the previous block is displayed in the workspace, with the same color coding. Below the code, the result of a parse operation is shown: "string parse: 'string' 'string'". The second "string" is highlighted in blue.

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

exercise



A screenshot of a workspace window titled "Workspace". The window contains a single line of code: `string parse: "qu""ote" 'qu"ote'`. The code is color-coded: "string parse:" is blue, "qu""ote" is purple, and 'qu"ote' is red. The text is highlighted with a light blue background.


```

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 := "" [^"] * ""

```

exercise

```

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

```

The screenshot shows a window titled "Workspace" with a white background and a grey title bar. The text inside the window is as follows:

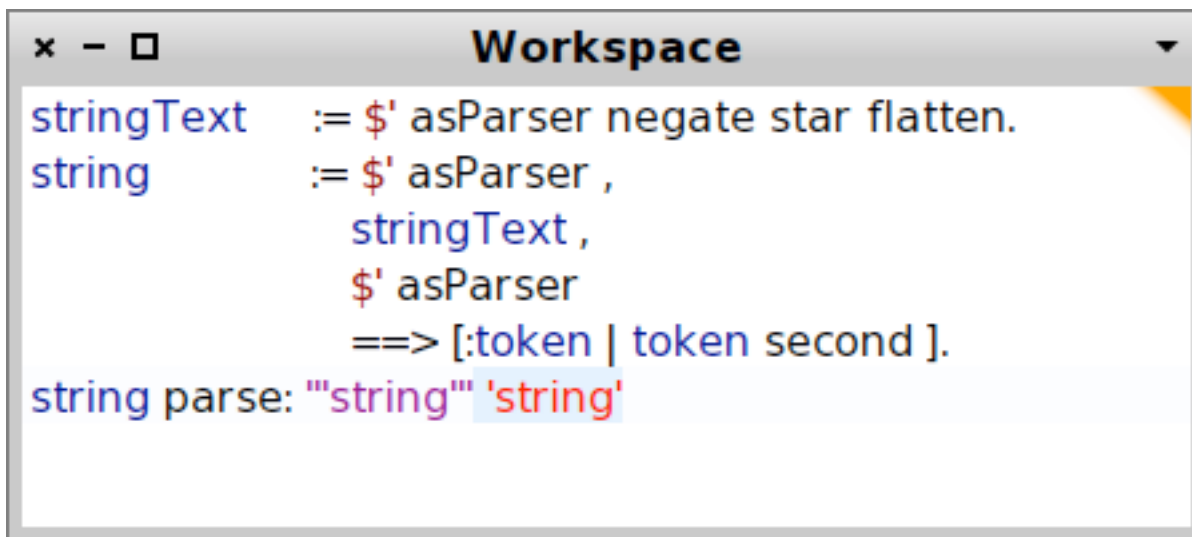
```

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

```

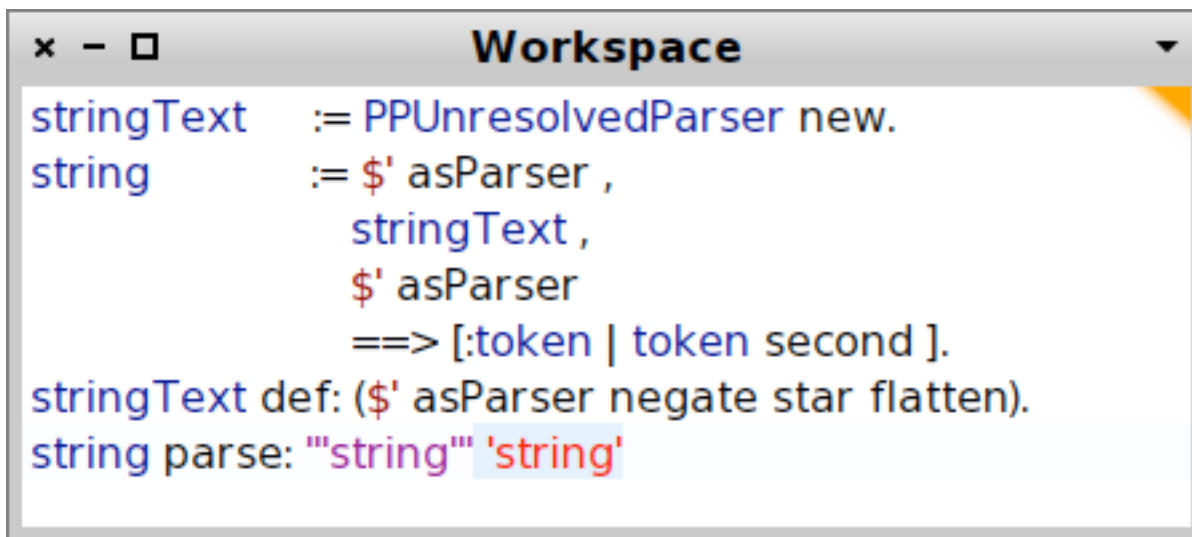
The text is color-coded: "mse matches:" is blue, the opening parenthesis "(" is blue, and the closing parenthesis ")" is blue. The word "true" is highlighted in light blue. The rest of the text is in purple.

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



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

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



The screenshot shows a workspace window titled "Workspace" with a standard window control bar (close, minimize, maximize). The code from the previous block is displayed in the workspace, with the same color-coding: blue for identifiers, red for literals, and black for operators and punctuation. The last line of code, `string parse: "string" 'string'`, is highlighted with a light blue background.

```
x - □ Workspace  
stringText := PUnresolvedParser new.  
string := '$' asParser ,  
        stringText ,  
        '$' asParser  
        ==> [:token | token second ].  
stringText def: ('$' asParser negate star flatten).  
string parse: "string" 'string'
```

x - □

Workspace

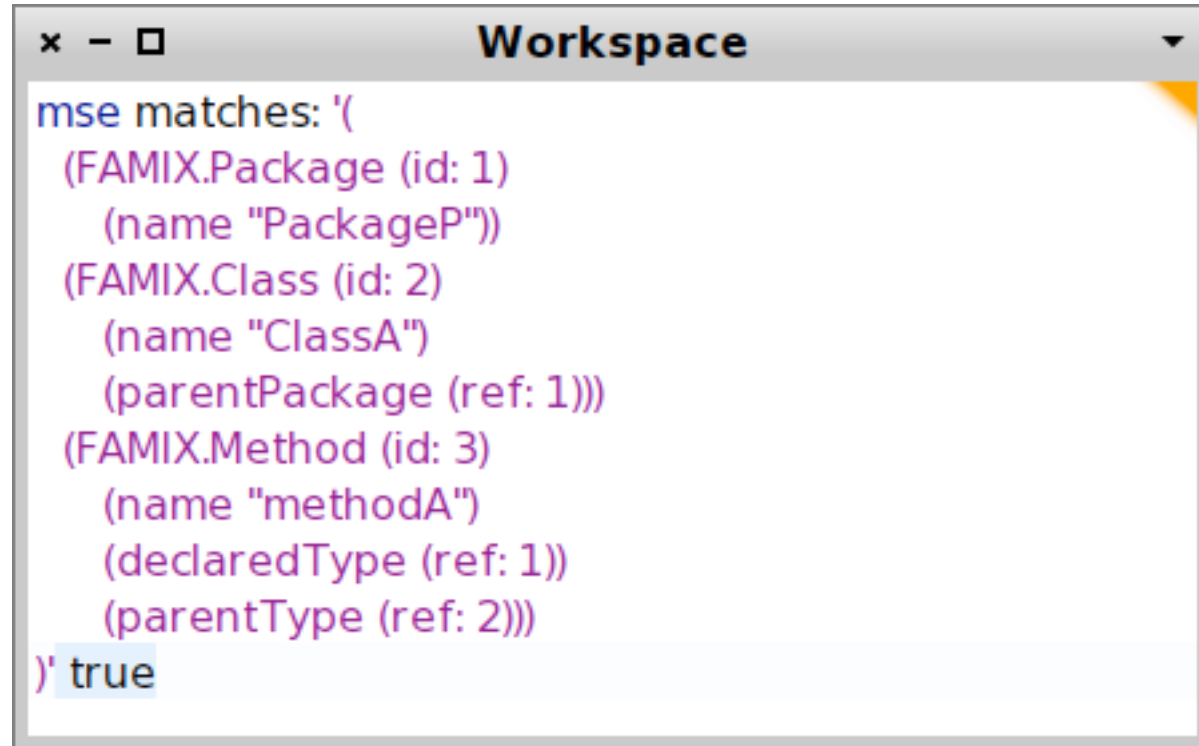
```
element := PPU unresolvedParser new.  
open := $( asParser trim.  
close := $) asParser trim.  
string := ($' asParser ,  
           ("" asParser / $(' asParser negate) star flatten ,  
           $(' asParser) trim.  
natural := #digit asParser plus flatten.  
e := ($e asParser / $E asParser) , ($- 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)))  
)
```



The screenshot shows a window titled "Workspace" with a standard window control bar (close, maximize, and a dropdown arrow). The main content area displays a match result in a purple monospace font. The text is as follows:

```
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

PPMSEGrammar Hierarchy

- PPParser
- PPDelegateParser
- PPCompositeParser
- PPMSEGrammar**
- PPMSEArrayParser

Instance ? Class

-- all --
accessing
basic
grammar

number
open
primitive
reference
simpleName
start
string

Browse Hierarchy Variables Implementors Inheritance Senders Versions View

start
^ elements end

start = default start parser

externally, parsers map on methods

The screenshot shows a window titled "PPMSEGrammar Hierarchy". On the left, a tree view lists several classes: PPParser, PPDelegateParser, PPCompositeParser, PPMSEGrammar (highlighted), and PPMSEArrayParser. Below the tree are buttons for "Instance", "?", and "Class". The main area is divided into three panes. The top-left pane shows a list of methods: "-- all --", "accessing", "basic", and "grammar". The top-right pane shows a list of methods: "attribute", "attributeValue" (highlighted), "boolean", "close", "e", "element", and "elementName". The bottom pane shows the details for the selected "attributeValue" method, including its signature: `^(primitive / reference / element) star`. At the bottom of the window, there are tabs for "Browse", "Hierarchy", "Variables", "Implementors", "Inheritance", "Senders", "Versions", and "View".

internally, parsers map on instance variables

to specify actions, subclass the base grammar

The screenshot shows a window titled "PPMSEArrayParser Hierarchy". The left pane lists the following classes: PPParser, PPDelegateParser, PPCompositeParser, PPMSEGrammar, and PPMSEArrayParser (which is selected). Below the list are buttons for "Instance", "?", and "Class". The middle pane shows the methods of the selected class: "-- all --", "accessing", and "values". The right pane shows the methods of the superclass: "attribute", "attributeValue" (selected), "boolean", "element", "elementName", "elements", and "id". Below the panes are tabs for "Browse", "Hierarchy", "Variables", "Implementors", "Inheritance", "Senders", "Versions", and "View". The bottom pane displays the implementation of the `attributeValue` method:

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

subclass tests from PPMSEGrammarTest

The screenshot shows an IDE window titled "PPMSEGrammarTest". The left pane displays a project tree with "PPMSEGrammarTest" selected. The middle pane shows the class hierarchy: "PPMSEGrammar" (superclass) and "PPMSEGrammarTest" (subclass). The right pane shows the "parserClass" attribute with a list of test methods: "testClose", "testElementName", "testNatural", "testNaturalWithSp", "testNumberWithE", and "testOpen". Below the panes are tabs for "Browse", "Hierarchy", "Variables", "Implementors", "Inheritance", "Senders", "Versions", and "View". The "Hierarchy" tab is active, showing the class hierarchy for "parserClass" as "PPMSEGrammar".

specify the parserClass

use #parse:rule: to check the grammar

The screenshot shows an IDE window titled "PPMSEGrammarTest". The window is divided into several panes. On the left, a project browser lists various packages, with "PPMSEGrammarTest" selected. The middle pane shows a class hierarchy for "PPMSEGrammarTest", listing "PPMSEGrammar", "PPMSEArrayParser", and "PPMSEGrammarTest" (selected). The right pane shows a list of methods, with "testNatural" selected. Below the panes, there are tabs for "Browse", "Hierarchy", "Variables", "Implementors", "Inheritance", "Senders", "Versions", and "View". The "testNatural" method is expanded, showing the code snippet: `self parse: '123' rule: #natural`.

Package	Class	Method
PetitMSE	PPMSEGrammar	-- all --
PetitJava-Core	PPMSEArrayParser	accessing
PetitJava-Tests	PPMSEGrammarTest	tests
PetitJava-AST	PPMSEArrayParser	tests-basic

testNatural
self parse: '123' rule: #natural

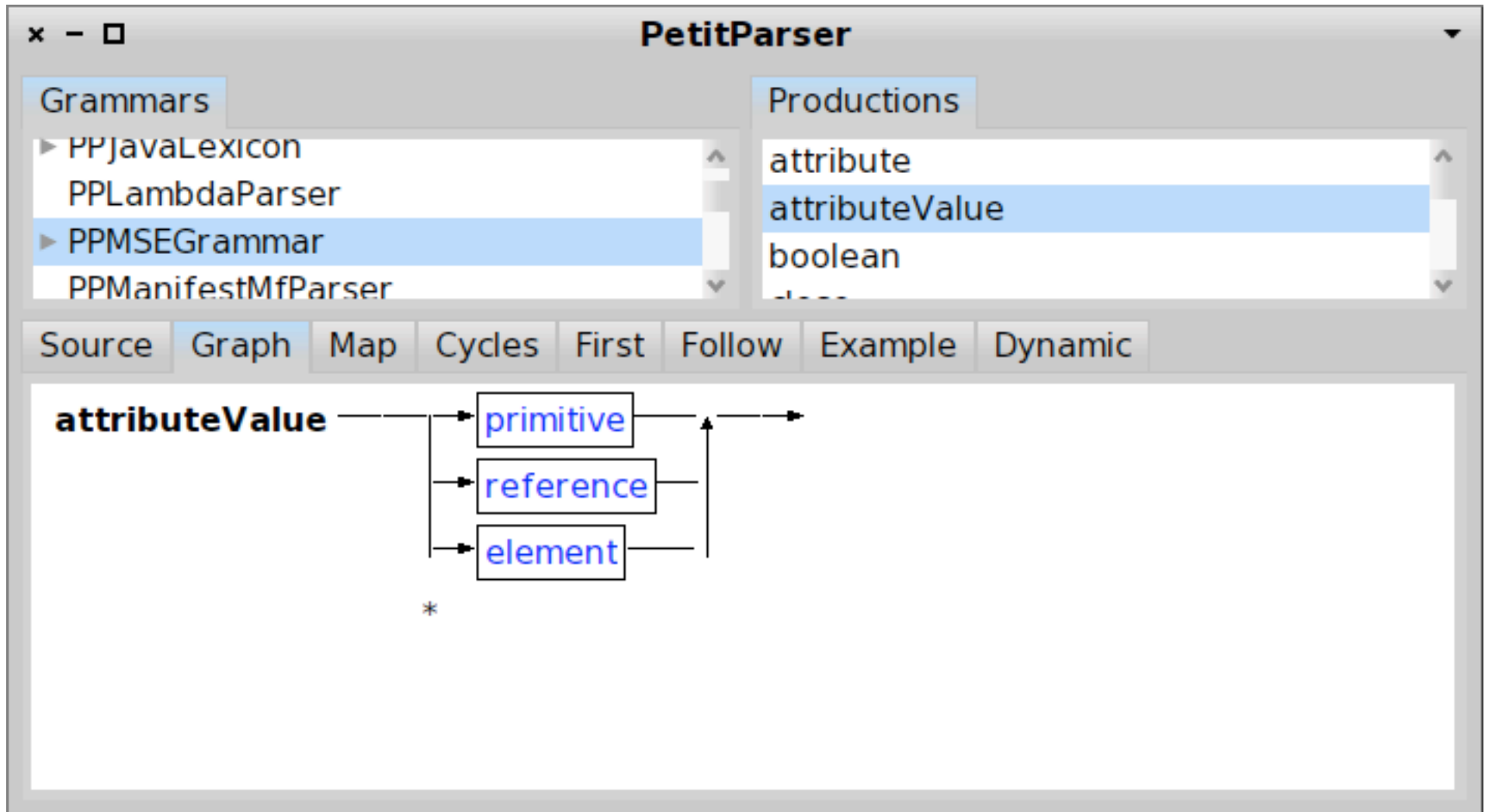
subclass to check the parser result

The screenshot shows an IDE window titled "PPMSEArrayParserTest". The interface is divided into several panes:

- Left Pane:** A project browser showing a tree of packages including PetitMSE, PetitJava-Core, PetitJava-Tests, PetitJava-AST, PetitJava-AST-Visitor, Arki-Reporter-Core, and Arki-Tests-Reporter.
- Middle Pane:** A class hierarchy for PPMSEArrayParserTest. It shows PPMSEGrammar and PPMSEArrayParser as superclasses. Under PPMSEArrayParser, there are two subclasses: PPMSEGrammarTest and PPMSEArrayParserTest. The latter is selected, and its superclass is PPMSEArrayParser.
- Right Pane:** A list of methods for the selected class. The methods are: parserClass, testElementName, testEmpty, testNatural (selected), testNaturalWithSp, testNumberWithE, and testOneElement.
- Bottom Pane:** A toolbar with buttons for "Browse", "Hierarchy", "Variables", "Implementors", "Inheritance", "Senders", "Versions", and "View". Below the toolbar, the selected method "testNatural" is displayed with its implementation:

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

PetitParser comes with a dedicated user interface



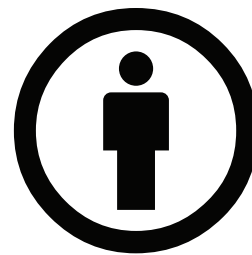
PPBrowser open.

Petit *Parser*

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

Tudor Gîrba

www.tudorgirba.com



creativecommons.org/licenses/by/3.0/