Meta-Matters in Squeak

Andrew P. Black
What's Meta?

• Metaprogramming is the act of writing a program that writes or manipulates another program... or themselves

• Why not? After all programs are just data!
Simple Example

- In the visitor pattern last week:
Solution (continued)

2. Every class *Foo* in the hierarchy gets a method `accept: aVisitor` defined as follows:

```plaintext
Foo >> accept: aVisitor
   \ aVisitor visitFoo: self
```

Note how the selector of the message tells the visitor what kind of node it is visiting.
2. Every class *Foo* in the hierarchy gets a method *accept: aVisitor* defined as follows:

```
Foo >> accept: aVisitor
    aVisitor visitFoo: self
```

Note how the selector of the message tells the visitor what kind of node it is visiting.

We wrote these methods with a metaprogram:

```
Expression allSubclassesDo: [ :each | each compile: 'accept: aVisitor
    aVisitor visit', (each name), ': self' classified: 'visiting' ]
```
Alternative Solution

• Instead of writing a separate program to write our program, we could make the program write itself:
  
• Put the following single method at the root of the hierarchy:

  ```
  Expression → accept: aVisitor
  ↑ aVisitor perform: ('visit', (self class name), ':') asSymbol
  with: self
  ```

• This is a reflective program — one that writes *itself* dynamically
Another Example

- We know that we can write this:
  \[(1 \text{ to } 10) \text{ select: } [ :x \mid x \text{ even}]\]

- How about this?
  \[(1 \text{ to } 10) \text{ select even}\]

- Can we make this work? What about other unary messages \(\text{odd, isPrime, } \ldots\)?
Summary of Solution

• (1 to: 10) select must answer an object that "remembers" the collection and the fact that we plan to do a select operation

• This object is called a Trampoline

• How can we make the trampoline understand even, odd, isPrime, factorial …

• Reflection!
Another Example

• Remember ParserFun?
Another Example
Another Example
Another Example
Another Example
Naming Parsers

• The name given to the parser is often (but not always) the same as the selector of the method that builds it.
Naming Parsers

• The name given to the parser is often (but not always) the same as the selector of the method that builds it.

```plaintext
digit
+ self satisfies: [:x | x isDigit] name: 'digit'
```
Naming Parsers

• The name given to the parser is often (but not always) the same as the selector of the method that builds it.

```plaintext
digit
  + self satisfies: [:d | d isDigit] name: 'digit'

alphaNum
  + self letter +++ self digit name: 'alphaNum'
```
Naming Parsers

- The name given to the parser is often (but not always) the same as the selector of the method that builds it.

```plaintext
digit
  + self satisfies: [:d | d isDigit] name: 'digit'

alphaNum
  + self letter +++ self digit name: 'alphaNum'

identifier
  + self lower >>= [:x |
    self alphaNum many >>= [:xs |
    self optionalSpaces >>
      (self unit: (xs copyWithFirst: x) as: Symbol)])]
  name: 'identifier'
```
Naming Parsers

• The name given to the parser is often (but not always) the same as the selector of the method that builds it.

```plaintext
digit
  + self satisfies: [:d | d isDigit] name: 'digit'

alphaNum
  + self letter +++ self digit name: 'alphaNum'

identifier
  + self lower >>= [:x |
    self alphaNum many >>= [:xs |
    self optionalSpaces »
    (self unit: (xs copyWithFirst: x) as: Symbol))]
  name: 'identifier'

keyword
  + (self string: 'if') +++ (self string: 'then') +++ (self string: 'else')
  >>= [:r | self spaces » (self unit: r)] name: 'keyword'
```
Obtain the name reflexively
Obtain the name reflexively

satisfies: aPredicate

"When evaluated, I parse an item from the input, and check whether it
satisfies aPredicate. If it does, I will answer that item; if it does not, I
fail."

+ self satisfies: aPredicate name: thisContext sender selector
Obtain the name reflexively

```plaintext
satisfies: aPredicate
"When evaluated, I parse an item from the input, and check whether it
satisfies aPredicate. If it does, I will answer that item; if it does not, I
fail."
> self satisfies: aPredicate name: thisContext sender selector

+++ anotherParser
"Deterministic choice.
Answers a parser that is the OR of myself and anotherParser."
self assert: [anotherParser isParserFor: parserStream].
> [] start |
start := parserStream position.
sel self parse
ifFail: [parserStream position: start.
anotherParser parse]]
asParserNamed: thisContext sender selector
on: parserStream
```
Obtain the name reflexively

```plaintext
satisfies: aPredicate
  "When evaluated, I parse an item from the input, and check whether it
  satisfies aPredicate. If it does, I will answer that item; if it does not, I
  fail."
  + self satisfies: aPredicate name: thisContext sender selector

+++ anotherParser
  "Deterministic choice.
  Answers a parser that is the OR of myself and anotherParser."
  self assert: [anotherParser isParserFor: parserStream].
  + [ | start |
    start := parserStream position.
    self parse
      ifFail: [parserStream position: start.
      anotherParser parse]]
  asParserNamed: thisContext sender selector
  on: parserStream

>> aParser
  "Sequencing.
  Answers a parser that, when run, will run the sequential composition
  of myself and aParser."
  + self
    === [:irrelevant | aParser] name: thisContext sender selector
```
Obtain the name reflexively

```ruby
satisfies: aPredicate
  "When evaluated, I parse an item from the input, and check whether it
  is accepted by aPredicate."

# aOneArgumentBlock

> aOneArgumentBlock
  "sequencing.
  Answers a parser that, when run, will first run me, and then evaluate
  aOneArgumentBlock, with the result of running me as its argument, and then finally run
  the answer obtained from evaluating the block, which is assumed to be a parser."

+ [ start ]
start := parserStream position.
self parse
  ifNotFailDo: [:v | (aOneArgumentBlock value: v) parse
    ifFailDo: [:f | parserStream position: start, f]]]
asParserNamed: thisContext sender selector
on: parserStream

ifFail: [parserStream position: start,
  anotherParser parse]]
asParserNamed: thisContext sender selector
on: parserStream

# aParser

> aParser
  "sequencing.
  Answers a parser that, when run, will run the sequential composition
  of myself and aParser."
+ self

> [ :irrelevant | aParser ] name: thisContext sender selector
Obtain the name reflexively

```plaintext
satisfies: aPredicate
    "When evaluated, I parse an item from the input and check whether it
    answers a parser that, when run, will first run me, and then evaluate
    aOneArgumentBlock, with the result of running me as its argument, and then finally run
    the answer obtained from evaluating the block, which is assumed to be a parser."

+ [1 start]
  start := parserStream position.
  self parse
    ifNotFailDo: [:v | (aOneArgumentBlock value: v) parse
      ifFailDo: [:f | parserStream position: start. f]]
    asParserNamed: thisContext sender selector
  on: parserStream
    ifFail: [parserStream position: start.
      anotherParser parse]
    asParserNamed: thisContext sender selector
  on: parserStream
```
Obtain the name reflexively

- There are only a few combinators that build parsers
  - Each combinator can extract the name from the context
Structural Equality

• We say how to build a recursive equality operation in Haskell that reaches down into the structure of a data type

• Can we do the same in Squeak?
  • How is equality defined in Object?
Try a new Equality Operation

```plaintext
=== anObject

"Answer whether the receiver and the argument have the same values and the same structure."

| nInstanceVars nIndexedVars |
| nInstanceVars := self class instSize. |
| nIndexedVars := self basicSize. |

anObject class instSize = nInstanceVars ifFalse: [^false].
anObject basicSize = nIndexedVars ifFalse: [^false].

1 to: nInstanceVars do:
  [ :i | (self instVarAt: i) === (anObject instVarAt: i) ifFalse: [^false]].
1 to: nIndexedVars do:
  [ :i | (self basicAt: i) === (anObject basicAt: i) ifFalse: [^false]].

^true
```
How does === work out?
What about zipAllWith:?

• We would like to be able to write

\[ \{ \lower{a\text{ to:}}\lower{z\text{ .}} \lower{A\text{ to:}}\lower{Z\text{ }} \text{zipAllWith:} \]
\[ \lower{\text{[ :lo :up }}| \text{ String \text{ with: lo \text{ with: up} ]} \]

for \( n \) collections and any \( n \) argument block

• Can we do it?
Smalltalk Browsers

• There are *lots* of different browsers in the Smalltalk environment
  • system browser, hierarchy browser, protocol browser, inheritance browser, … inspector, explorer, change set browser, file system browser

• Each one “knows” about the structure that it is browsing
  • *e.g.*, the system browser has hardwired into its code the facts that Categories contain Classes and Classes contain Protocols andProtocols contain methods
The OmniBrowser

• The OmniBrowser is a browser for everything and nothing in particular
  • it doesn’t “know” about any system structure
  • instead, it is driven by metadata that describes the thing that it is browsing

• The metadata takes the form of a graph of objects — the metagraph

• The domain that the browser navigates is also a graph of objects — the subject graph
A File System Browser

• We will build an instance of the OmniBrowser that examines the file system

• The file system is *not* a graph of objects

• That’s OK: we build **OBNodes** to represent the entities that we are browsing

• We define two subclasses of **OBNode**: **OBDirectoryNode** and **OBFileNode**

• What do these OBNodes have to do?
  • that is defined by the metagraph
File System: Graph & Metagraph

File
Directory
#files
N metanode
is a child of
#directories
N object node
/
/temp pic1.jpg
pic2.jpg
pic3.jpg

Figure 2: A graph describing a file system and its corresponding metagraph.
File System: Graph & Metagrapgh

```
N object node

/ directory

/temp   pic1.jpg
pic2.jpg
pic3.jpg
```

In a metagraph each type of node of an object graph is represented by a metanode. As a whole the metagraph describes the structure of the object graph navigated by the browser.

Edges between different types of nodes with in the browser area defined using another type of graph a metagraph. New metagraphs should be returned from classes of the class OBMetagraph. Thus we construct the file browser metagraph in the method #fileBrowser:

```
File System: Graph & Metagraph

The abstract relationships between different types of nodes within the browser are defined using another type of graph, a metagraph. In a metagraph, each type of node of an object graph is represented by a metanode. As a whole, the metagraph describes the structure of the object graph navigated by the browser.

In our file browser, there are two metanodes for the two types of nodes in the object graph: directory and file. The metagraph is a hierarchy where a metanode is a child of another metanode.

The class methods of the class OBMetagraph return the browser metagraph in the method #fileBrowser:

```
object node

is a child of

directory

directory

N metanode

is a child of

N object node

/

temp pic1.jpg

pic2.jpg ...
```

The file system graph and metagraph are shown in the figure below:

![File System Graph & Metagraph Diagram](image.png)
File System: Graph & Metagraph

File
Directory
#files
N metanode
is a child of
#directories
N object node
/
/temp pic1.jpg
pic2.jpg
pic3.jpg

Figure 2: A graph describing a file system and its corresponding metagraph.

Edges between different types of nodes within the browser are defined using another type of graph, a metagraph. In a metagraph, each type of node of an object graph is represented by a metanode. As a whole, the metagraph describes the structure of the object graph navigated by the browser.

Kind of node:
- **FileNode**
  - `newPath` method
  - `path` instance variable
  - `setPath:` class method
  - `name` method

Kind of directory:
- **FileNode**
  - `newPath` method
  - `path` instance variable
  - `setPath:` class method
  - `name` method

The abstract relationships between different types of nodes within the browser are defined using another type of graph, a metagraph. In a metagraph, each type of node of an object graph is represented by a metanode. As a whole, the metagraph describes the structure of the object graph navigated by the browser.
Metagraph as data
Metagraph as data