Squeak: A minimalist syntax!

Dr. Stéphane Ducasse

Generously translated in english by Yanick Beaudet

ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/

Last month we have shown you a very quick overview of Squeak. In this article, we wish to introduce Squeak's syntax in the way_of small scripts. We suggest again to try out all expressions using the Squeak application on the CD provided with this issue. We will begin by looking at how we can execute expressions interactively. Then we will quickly introduce the syntax and show how to write small scripts that will allow you to access the web, manipulate images or generate music. To give some background, you should know that Smalltalk was influenced by the following languages: Lisp (lexical closure, uniformity), LOGO (a dynamic functional language) and Simula (object programming).

Next month we will show how to define other scripts and will also discuss the principles of design which define Squeak. The following month we will introduce class definitions while programming Morphs. We will also show how Smalltalk offers to the programmer a very close relationship with the objects which it creates because it is based on very short cycles of incremental compilation which allow a constant interaction with the objects created.

1. Executing a Squeak expression

We suggest that you work in Morph (menu open... ->Morph project) then enter on the project. The Morph environment is the stansard environment in version 3.0 and it is much prettier and user-friendly. To execute a script, open a workspace (**menu open...-> workspace**), type the script, select it (if you click just before the first character, the whole text is selected) and reveal a menu then choose the " doIt " option as shows it Figure 1. We suggest you type the script shown in Figure 1 because the result is striking. Try to open multiple windows or colour Morphs. In fact in any window where text can be typed (code browser, debugger...) one can compile and execute code in this manner.

In case Squeak runs slowly on your computer, try to select the same depth (the number of bit representing each colour) in Squeak and in your system: (**menu appareance -> set display depth and select a number**). On current video cards 32 will not force Squeak to emulate other graphic settings.

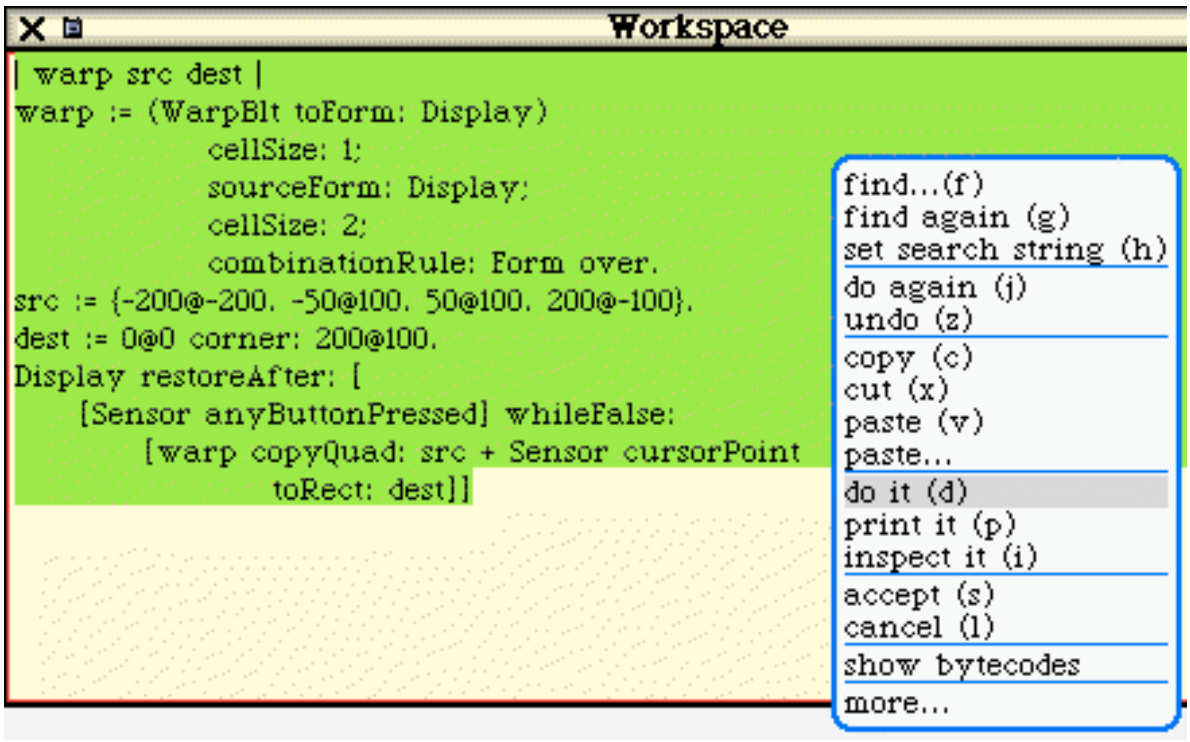

Figure 1 How to dynamically compile a script in Squeak

2. Uniformity and simplicity

The syntax in Smalltalk is simple. Very simple if one manages to get rid of old reflexes created by other languages such as C, C++ or Java. Ralph Johnson, one of the four authors of the "Design Patterns" book frenquently says that the syntax of Smalltalk holds on a postcard. And it is true (see further)!!!

Smalltalk reserved words

- `self` (équivalent of this) and `super` are pseudo-variables which refer to the object executing a method containing them.
- `true`, `false` represent boolean values. They are the unique instances of the classes True and False respectively. And yes everything is an object even the boolean values.
- `nil` is the equivalent of `NULL` or `null`. `nil` is the unique instance of the class `UndefinedObject`

Syntactic elements

- **`:=`** (and _ which appears as an arrow in Squeak) allow the assignment of the contents of a variable.
- `^` returns a result from a method. By default, every method returns the receiver (self), but the code browsers will show it only if necessary.
- **`.`** is an expression separator (and not a terminator thus not required at the end).
- **`;`** (called cascade) makes it possible to send several messages to the same object
- `|` `|` declaration of local variables in a context. `| x | x := 3. x+ 1`
- `|` end of argument declaration in a lexical closure.

- `[]` defines a lexical closure which is an first class object as in Lisp . `[:y| y + 2] value:` 3 returns 5
- `:x` defines an argument of a lexical closure.
- `#` defines a symbol (uniquely identified character strings). `#IAmUnique`
- `''` character string. The quote is doubled to include the quote in the string: `'idiot' 'idiot n''est pas celui que l''on croit' 'Je ne suis pas unique'`
- `#()` defines an array at compile-time. `#(a b #(true))`
- `#[]` defines a byte array `#[1 2 234]`
- `""` comment: `"this is a comment"`
- By convention the local variables start with a lowercase and the global variables start with an uppercase letter. The names of class start with an uppercase letter because they are global variables. Thus, true is instance of the class True.

Squeak is late with regards to the lexical closure because they are simulated and not implemented as in all other smalltalk. This is due to the fact that Squeak is based on one of the first implementations of Smalltalk. This situation will probably change.

On a postcard.

Here the complete syntax of Smalltalk on a postcard as shown by Ralph Johnson. Obviously the method does not do anything useful! We will show how to compile a method in the next article.

```
exampleWithNumber: x
 "This is a small method that illustrates every part of Smalltalk
 method syntax except primitives, which aren't very standard. It
 has unary, binary, and key word messages, declares arguments and
 temporaries (but not block temporaries), accesses a global
 variable (but not class and instance variable), uses literals
 (array,character, symbol, string, integer, float), uses the
 pseudo variables true false, nil, self, and super, and has
 sequence, assignment, return and cascade. It has both zero
 argument and one argument blocks. It doesn't do anything useful,
 though"

  |y|
  true & false not & (nil isNil) ifFalse: [self halt].
  y := self size + super size.
  #($a #a 'a' 1 1.0)
   do: [:each | Transcript
              show: (each class name);
              show: (each printString);
              show: ' '].
  ^ x < y
```

3. Messages sending by example

In Smalltalk, except for the syntactic elements listed in the preceding section (: = ^. # # () [ ] | |), EVERYTHING is a method invocation, also referred in Smalltalk slang, “message send”. In Smalltalk, one sends a message to an object, and there is no concept of operator, nor of overload thus one can define + on a class of his/her choice (contrary to Java) but one does not have the possibility of defining their precedence as in C++. This must be solved explicitly as shows the following example. Moreover, one cannot overload a method. Simplicity has a price!

The original intention for Smalltalk was to make it possible to children to read the code, this choice is seen in method calls. Smalltalk has three types of method invocation or message sends: unary, binary or by keyword, based on the name of the method (the examples given are taken from scripts presented further):

- unary: The name of the method is simply made of a succession of characters (factorial, open, class).
  ```
  Try: 2000 factorial
  Browser open
  ```
- binary: the method name is composed of one or two characters chosen from the following symbols `+ - / \ * ~ < > = @ & ? ,` for instance: `> =` , `= ` , `@`
  ```
  100@100 "creates a point"
  ```
- keyword: the method name is formed by one or many words ending by a : which specifies that an argument is expected.
  Example: `r:g:b:` (three arguments), `playFileNamed: (one argument), at:put: (two arguments),`
  ```
  Color r: 1 g: 0 b: 0
  MIDIFileReader playFileNamed: 'LetItBe.MID'
  ```

For instance, to create an instance of the class Color one can use the following method: r:g:b: Color R: 1 G: 0 b: 0 that would be equivalent to a Java or C++ method r:g:b:(1, 0, 0). The : are part of the name of the method.

These various message sends have different precedences which allows their composition in an elegant way. When precedence is the same the expressions are carried out left to right.

Unary > Binary > Keyword

Thus the unary invocations are always evaluated first then the binary ones and then the keywords. Of course parenthesis allow to change this order.

Illustration:

Type and compile the following expressions (if you want to see the result use " printIt " instead of " doIt ").

```
Browser open                                    "unary message"
1000 factorial                                        " unary message "
2 + 3 * 5                              "Binary messages executed from
left to right"
(2 + 3) * 5
1/3 + 1                    "adding an integer to a fraction results in a
fraction"

| array |
array := #(1 2 3).
array at: 1 put: 4.
  "example of keyword message"
array

MIDIFileReader playFileNamed: 'LetItBe.MID'
"Opens the MIDI player using the specified file"

Display restoreAfter: [WarpBlt test4]
"Keyword message, try test1, test12, test3, test4 and test 5"
```

As you can see the syntax and in particular the keyword messages as in the example `array at: 1 put: 4` make it possible to write code with a structure approaching that of the natural languages. This was one of the initial objectives so that the children can program.

Message Composition

We will now present some examples of message composition.

- Composition of uanry and binary messages..... Yes we obtain 1000 but try this in your favorite language to see if you obtain the same result as quickly. Note that it is an excellent example of automatic coercion and exact handling of a number. Try to display the result of 1000 factorial, it takes more time to display it than to calculate it.

```
1000 factorial / 999 factorial
```

- To try if you are connected, this script shows the composition of a keyword message and a unary message. Without the brackets, the display message would have been sent to the string and not to the received image.

```
(HTTPSocket httpShowGif: 'www.altavista.digital.com/av/pix/default/av-
adv.gif') display
```

- the sound named Clarinet is created then passed as an argument of the message `lowMajorScaleOn`:

```
(FMSound lowMajorScaleOn: FMSound clarinet) play
```

- An instance of speaker is created then it is asked to pronounce a string that you can of course mofify!

```
Speaker manWithEditor say: 'Hello readers of Programmez, I hope you will
have fun with Squeak'
```

- Composition of unary messages. An instance of the digitizer is created then visualized. If you microphone is plugged in try a sample!

```
RecordingControlsMorph new openInWorld
```

4. Graphics Handling

Images.

Squeak allows the handling of bitmaps (classes BitBlt and WarpBlt) as well as usual image formats like GIF, JPEG, png... The simplest script is to allow the capture of a part of the screen and save it in a file.

```
| im |
im := Form fromUser.
GIFReadWriter putForm: im onFileNamed: 'test.gif'
```

We declare a variable, ask the user to capture a part of the screen and then save the part captured as a GIF file.

Image handling.

All image formats are in fact internally transformed into Form. a Form allows multiple transformatino possibilities flip, rotation, zoom... Here is an example which makes a rotation and zooms on the portion of screen where the mouse is as shown in Figure 2.

```
| angle f |
f := Form fromDisplay: (0@0 extent: 300@300).
angle := 0.
[Sensor anyButtonPressed] whileFalse:
  [((Form fromDisplay:
      (Rectangle  center: Sensor cursorPoint
                extent: 130@66))
         rotateBy: angle
         magnify: 2
```

```
        smoothing: 1) display.
  angle := angle + 5].
f display
```

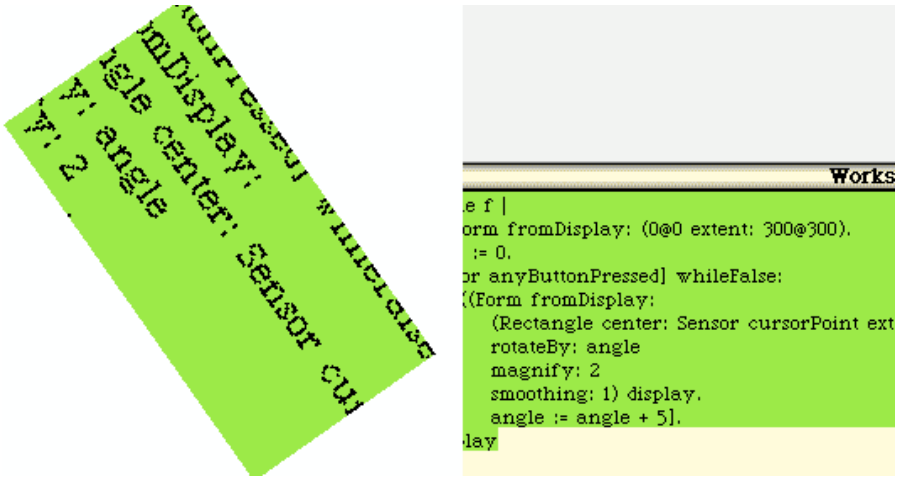

Figure 2 How to dynamically compile a script in Squeak

Let us explain a little...

- First of all, after having declared two variables we capture and save a part of the screen by specifying a rectangular area. (0@0 extent: 300@300) returns a rectangle and @ is a message invoked with a number which returns a point.
- we then initialize a variable. Then, as long as no button is pressed, we create a new Form from a screen area based on the position of the mouse. Sensor cursorPoint returns the position of the mouse.. `Rectangle center: Sensor cursorPoin extent: 130@66` returns a rectangle centered on the mouse position with a size of 130 by 66 pixels.
- Then we asks this Form to rotate on itself while enlarging it. Note that the name of the method is `rotateBy: magnify:smoothing`: and thus it has 3 arguments. We ask this Form to be displayed, then we increment the angle. We continue until a key is pressed.
- When a key is pressed, we redisplay the captured screen.

5. Let's conclude

Strangely, Smalltalk's syntax does not introduce loops, conditions, or classes or methods definitions... We will see how a minimalist syntax can perfectly take into account these functionalities in an elegant way in the next article.

References

- www.mucow.com/squeak-qref.html (The quick guide to Squeak syntax)
- www.squeak.org/ (The official website)
- minnow.cc.gatech.edu/ (The community's wiki which abounds with information)
- www.iam.unibe.ch/~ducasse/ (My Smalltalk course)