Squeak: Une syntaxe minimaliste!

Dr. Stéphane Ducasse

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

Le mois dernier nous vous avons montré un très rapide tour d'horizon de Squeak. Dans cet article, nous voulons présenter la syntaxe de Squeak par le biais de petits scripts. Nous vous proposons encore une fois de tester toutes les expressions avec la version de Squeak qui est fournie sur le CD joint à ce numéro. Nous allons commencer par regarder comment on peut exécuter de manière interactive des expressions. Ensuite nous abordons rapidement la syntaxe et montrons comment écrire de petits scripts qui vont vous permettre d'accéder au Web, manipuler des images ou générer de la musique. Pour donner quelques repères, il faut savoir que Smalltalk a été influencé par les langages suivants: Lisp (fermeture lexicale, uniformité), LOGO (un langage fonctionnel dynamique) et Simula (programmation à objets).

Le mois prochain nous montrerons comment définir d'autres scripts et discuterons aussi des principes de conception qui anime Squeak. Le mois suivant nous aborderons la définition de classes en programmant des Morphs. Nous montrerons aussi comment Smalltalk offre au programmeur une relation très étroite avec les objets qu'il crée car il est basé sur des cycles de compilation incrémentale très courts qui permettent une constante interaction avec les objets créés.

1. Exécuter une expression en Squeak

Nous vous suggérons de travailler en Morph (**menu open...->Morph project**) puis **enter** sur le projet. L'environnement Morph est l'environnement standard dans la version 3.0 et il est beaucoup plus joli et convivial. Pour exécuter un script, ouvrer un workspace (**menu open...-> workspace**), tapez le script, sélectionnez-le (si vous cliquez jusque avant le premier caractère, le texte entier est sélectionné) et faites apparaître un menu puis choisissez le choix *"doIt"* comme le montre la Figure 1. Nous vous suggérons de tapez le script décrit dans la Figure 1 car le résultat est saisissant. Essayez d'ouvrir plein de fenêtres ou de morphs de couleurs. En fait, dans n'importe quelle fenêtre prévue à cet effet (browseur de code, débogueur,...) on peut compiler et exécuter du code de cette manière.

Si Squeak s'exécute lentement sur votre machine, essayez d'avoir la même profondeur (le nombre de bits représentant une couleur) en Squeak que dans votre système: (**menu appareance -> set display depth et choisissez un nombre**). Sur les cartes actuelles 32 évitera à Squeak de simuler d'autres modes graphiques.

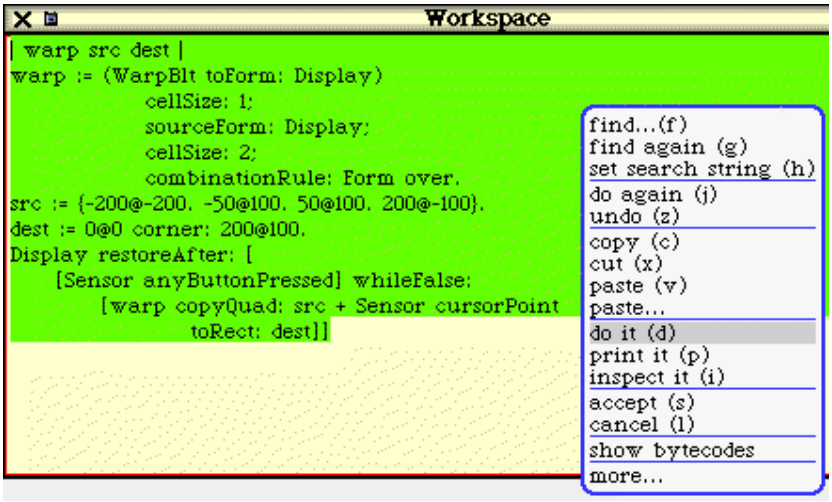

Figure 1 Comment compiler dynamiquement un script en Squeak.

2. Uniformité et simplicité

La syntaxe des Smalltalk est simple. Très simple si l'on parvient à se débarrasser des vieux réflexes créés par les autres langages que sont C, C++ ou Java. Ralph Johnson un des quatre auteurs des Design Patterns a l'habitude de dire que la syntaxe de Smalltalk tient sur une carte postale. Et c'est vrai (voir plus loin)!!!

Mots réservés en Smalltalk.

- `self` (équivalent de this) et `super` sont des pseudo-variables qui font références à l'objet exécutant une méthode qui les contient.
- `true`, `false` représentent les booléens. Il sont les uniques instances des classes `True` et `False`. Et oui tout est objet même les booléens.
- `nil` est l'équivalent de `NULL` ou `null`. `nil` est l'unique instance de la classe `UndefinedObject`

Eléments syntaxiques.

- `:=` (et _ qui apparaît comme une flèche en Squeak) permet l'affectation du contenu d'une variable.
- `^` permet de rendre un résultat dans une méthode. Par défaut, toute méthode rend le receveur (`self`), mais les browseurs de code ne le montrent que si nécessaire.
- `.` est un séparateur d'expression (et non terminateur donc pas besoin de toujours le mettre à la fin).
- `;` (appelée cascade) permet d'envoyer plusieurs messages à un même objet.
- `| |` déclaration de variables locales dans un contexte. `| x | x := 3. x+ 1`
- `|` fin de déclaration d'arguments de fermeture lexicale.
- `[]` définit une fermeture lexicale qui est un objet de première classe comme en Lisp. `[:y| y + 2] value: 3` rend 5
- `:x` définit un argument de fermeture lexicale.
- `#` définit un symbole (des chaînes de caractères à identité unique). `#JesuisUnique`
- `''` chaîne de caractères on double pour avoir une quote: `'idiot' 'idiot n''est pas celui que l''on croit' 'Je ne suis pas unique'`
- `#()` définit un tableau défini lors de la compilation. `#(a b #(true))`

- `#[]` définit un tableau de byte `#[1 2 234]`
- `""` commentaire: `"voila un commentaire"`
- Par convention les variables locales commencent par une minuscule et les variables globales commencent par une majuscule. Les noms de classe sont commencent par une majuscule car elles sont des variables globales. `true` est donc un instance de la classe `True`.

Squeak est en retard au niveau des fermetures lexicales car elles sont simulées et non implémentées comme dans tous les autres Smalltalk. Ceci est du au fait que Squeak est basé sur une des premières implémentations de Smalltalk. Cette situation va vraisemblablement changée.

Sur une carte postale. Voici la syntaxe de Smalltalk compl te sur une carte postale comme le montre Ralph Johnson. Evidemment la méthode ne fait rien de bien utile ! Nous montrerons comment compiler une méthode dans le prochain 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. Envoi de messages par l'exemple

En Smalltalk, à l'exception des éléments syntaxiques dénombrés dans la section précédente (`:=^ . ; # #() [] | |`), TOUT est invocation de méthode, aussi nommée en jargon Smalltalk, *envoi de message*. En Smalltalk, on envoie un message à un objet, et il n'y a pas de notion d'opérateurs, ni de surcharge donc on peut définir + sur une classe de son choix (contrairement à Java) mais on n'a pas la possibilité de définir leur précédence comme en C++. Ceci doit être résolu explicitement comme le montre les exemples suivants. De plus, on ne peut surcharger une méthode.La simplicité a bien un prix!

L'intention qui a poussé la création de Smalltalk était de permettre à des enfants de lire le code, ce choix se voit dans l'appel de méthode. Smalltalk propose trois types d'invocation ou d'envoi de messages unaire, binaire ou à mots-cl ès basée sur la façon dont le nom de la méthode est formé (les exemples donnés sont pris du code des scripts présentés plus loin):

- unaire: Le nom de la méthode est simplement formé d'une suite de caractères (`factorial`, `open`, `class`). Essayez:

```
    2000 factorial
    Browser open
```

- binaire: le nom de la méthode est formé de un ou deux caractères choisis parmi les symboles `+ - / \ * ~ < > = @ & ? ,` comme par exemple: `>=`, `=` , `@`
  `100@100 “créé un point“`
- à base de mots-clés: le nom de la méthode est formé d’un ou plusieurs mots terminés par un : qui spécifie qu’un argument est attendu.
  Exemple: `r:g:b:` (trois arguments), `playFileNamed:` (un argument), `at:put:` (deux arguments),
  `Color r: 1 g: 0 b: 0`
  `MIDIFileReader playFileNamed: 'LetItBe.MID'`

Par exemple, pour créer une instance de la classe couleur on peut utiliser la méthode: `r:g:b:` comme suit `Color r: 1 g: 0 b: 0` ce qui serait équivalent à une méthode Java ou C++ `r:g:b:( 1, 0, 0)`. Les : font partie du nom de la méthode.

Ces différents envois de messages ont des précédences différentes qui permet leur composition de manière élégante. Lorsque les précédences sont les memes les expressions sont exécutées de la gauche vers la droite.

Unaire > Binaire > Mots-Clès

Ainsi les invocations unaires sont toujours évaluées en premier puis les binaires et ensuite les mots-clés. Bien sur les parenthèses permettent de changer cet ordre.

Illustration: Tapez et compilez les expressions suivantes (si vous voulez voir le résultat utilisez “*printIt*” au lieu de “*doIt*”).

```
Browser open                    “message unaire”
1000 factorial                  “message unaire“
2 + 3 * 5                       “message binaires composés de gauche à droite”
(2 + 3) * 5
1/3 + 1             “on ajoute une fraction à un entier on obtient une fraction”

| array |
array := #(1 2 3).
array at: 1 put: 4.             “exemple de message à mots-clès“
array

MIDIFileReader playFileNamed: 'LetItBe.MID'“Ouvre le MIDI player sur le fichier
spécifié“

Display restoreAfter: [WarpBlt test4]
“message à mots-clès, essayez test1, test12, test3, test4 et test 5”
```

Comme vous le voyez la syntaxe et en particulier les messages à mots-clès comme dans l’exemple `array at: 1 put: 4` permettent d’écrire du code qui une structure s’approchant des langues naturelles. Ceci était un des objectifs initiaux afin que les enfants puissent programmer.

Composition de messages.

Nous vous présentons maintenant des exemples de compositions de messages.

- Composition de messages unaires et binaires.....Et oui on obtient 1000 mais essayez dans votre langage favori si vous obtenez le même résultat aussi rapidement. Notez que c’est un excellent exemple de coercion

automatique et manipulation exacte de nombre. Essayez d'afficher le résultat de 1000 factorial, cela prend plus de temps pour l'afficher que pour le calculer.

```
1000 factorial / 999 factorial
```

- A essayer si vous êtes connecté, ce script montre la composition d'une message à mots-clès et d'un message unaire. Sans les parenthèses, le message `display` aurait été envoyé à la chaîne et non à l'image reçue.

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

- Le son nommé `clarinet` est créé puis passé comme argument du message `lowMajorScaleOn`:

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

- Une instance de `Speaker` est créée puis on lui demande de prononcer une chaîne de caractères que vous pouvez bien sur modifier!

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

- Composition de messages unaires. Une instance de digitaliseur est créée puis visualisée. Si votre micro est branché essayez un sample!

```
RecordingControlsMorph new openInWorld
```

4. Manipulations graphiques

Images. Squeak permet la manipulation de bitmaps (classes `BitBlt` et `WarpBlt`) ainsi que des format d'images usuels comme GIF, JPEG, PNG...Le plus simple script est de permettre de capturer une partie de l'écran et de la sauvée dans un fichier.

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

On déclare une variable, demande à l'utilisateur de capturer une partie de l'écran et on sauve la partie capturée comme un fichier gif.

Manipulation d'images. Tous les formats d'images sont en fait transformés en interne en Form. Une Form permet de multiples possibilités de manipulation flip, rotation, zoom... Voici un exemple qui fait une rotation et un zoom de la portion d'écran ou se trouve la souris comme-montré dans la 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
```

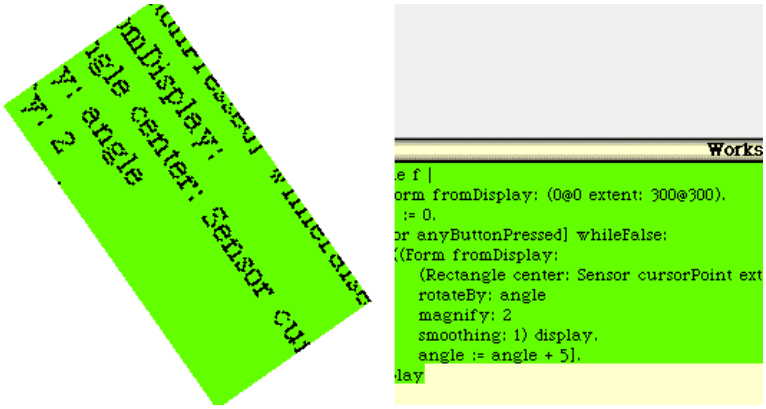

Figure 2 Comment compiler dynamiquement un script en Squeak.

Expliquons un peu...

- Tout d'abord après avoir déclaré deux variables, nous capturons et sauvons une partie de l'écran en spécifiant une zone rectangulaire. `(0@0 extent: 300@300)` rend un rectangle et `@` est un message envoyé à un nombre qui rend un point.
- Puis nous initialisons une variable. Ensuite, tant qu'aucun bouton n'est pressé, nous créons une nouvelle Form depuis une partie de l'écran basée sur la position de la souris. `Sensor cursorPoint` rend la position de la souris. `Rectangle center: Sensor cursorPoin extent: 130@66` rend un rectangle centré sur la position de la souris de 130 sur 66 pixels.
- Ensuite nous demandons à cette Form de tourner sur elle-même tout en la grossissant. Notez bien que le nom de la méthode est `rotateBy:magnify:smoothing`: et donc elle a 3 arguments. Nous demandons à cette form de s'afficher, puis nous incrémentons la valeur de l'angle. Nous faisons cela jusqu'a ce qu'une touche soit pressée.
- Lorsque l'on presse sur une touche, nous réaffichons l'écran que nous avions capturé.

5. Concluons

Etrangement la syntaxe de Smalltalk n'introduit pas de constructeurs de boucles, de conditions, de définition de classes ou méthodes....Nous verrons comment une syntaxe minimaliste peut parfaitement prendre en compte ces fonctionalités de manière élégante.

Références

- www.mucow.com/squeak-qref.html (Le guide rapide de la syntaxe Squeak)
- www.squeak.org/ (le site officiel)
- minnow.cc.gatech.edu/ (le wiki de la communauté qui regorge d'informations)
- www.iam.unibe.ch/~ducasse/ (Mon cours sur Smalltalk)