Shufflers-Combinators

C.Kibleur

dernière version:
January 28, 2009

Contents

0.1 Stack Shufflers 2
0.1.1 Désavantages 3
0.1.2 Les quotations 3
0.1.3 Les combinators 4
0.1.4 Les combinators qui expriment une intention 4
0.1.5 Cleave, Spread, Apply 5
0.1.6 Quelques détails sur les types données 7

Jusqu'à présent, nous avons utilisé Factor un peu comme une calculatrice RPN (Notation Polonaise Inversée). Je veux simplement être sûr que vous ne sous- estimiez pas Factor à cause des exemples que j'ai utilisé jusqu'à présent. Factor est un langage moderne, vraiment bon dans tous domaines: des applications Web, à la programmation de jeux, en passant par la manipulation complexe de texte, etc. J'utilise donc des exemples ultra-simplistes pour illustrer mes propos à propos du langage. Si vous vous suivez correctement, je peux vous assurer que ceux-ci deviendront à la longue beaucoup plus intéressants.

Mise en garde: ce second tutoriel va certainement vous sembler beaucoup plus abstrait que le précédent. Compensation? Il est aussi beaucoup plus intéressant.

0.1. Stack Shufflers

Dû au fait que Factor est un langage à pile, vous aurez de temps en temps besoin de réarranger l'ordre de ses éléments, ou encore de copier ces éléments pour s'assurer qu'ils sont dans le bon ordre pour une utilisation future. Il existe un groupe de mots pour ce faire, on les appelle les **stack shufflers**.

Ce sont des mots dédiés à changer l'odre (ou le nombre) d'éléments au dessus de la pile. On dit même que "les stack shufflers contrôlent le flux de données entre les mots qui exécutent des actions". En fait, nous avons déjà rencontré un stack-shuffler : `drop`.

Il existe ainsi trois variétés de stack-shufflers. Les voici avec un exemple de chaque:

1. Removing Shufflers (ils retirent des éléments de la pile)

   ```
   drop ( x - )
   ```

   enlève le dernier élément de la pile, puis le jette.

2. Duplicating Shufflers (ils dupliquent des éléments sur la pile)

   ```
   dup ( x - x x )
   ```

   duplique le dernier élément sur la pile

3. Permuting Shufflers (permutation d'éléments)

   ```
   swap ( x y - y x )
   ```

   échange les positions des premier et deuxième éléments sur la pile.

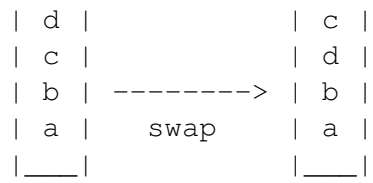
Visualisation de `drop`:

```
| c |            |   |
| b | --------> | b |
| a |   drop    | a |
|___|            |___|
```

Celle de `dup`:

```
|   |            | b |
| b | --------> | b |
| a |    dup    | a |
|___|            |___|
```

Une illustration de l'effet `swap`:



Voici maintenant deux `mots` de la bibliothèque standard de Factor qui utilisent ces shufflers.

Le premier, `sq` va calculer le carré d'un nombre qui lui est donné en entrée, le second `neg` calcule l'opposé d'un nombre donné:

```
: sq ( x -- y )
  dup * ;

: neg ( x -- -x )
  0 swap - ;
```

Maintenant, on peut tester ces deux mots dans le listener :

```
(scratchpad) 5 sq .
25
(scratchpad) 1 neg .
-1
```

0.1.1. Désavantages

Il existe encore beaucoup de Shuffle Words* qui provoquent un réarrangement ou qui dupliquent/suppriment un ou des éléments de la pile. Le problème est que du code rempli de shufflers peut vite devenir très confus, voire illisible. En général, on doit essayer de les utiliser au minimum afin de rendre notre code plus compréhensible.

Bien sûr, il existe des exceptions à cette règle, des fois ou les `shufflers` prennent tout leur sens, mais la meilleure alternative est l'utilisation des `combinators`. Mais avant d'aborder les combinators, on doit tout d'abord survoller un concept essentiel avec Factor...

0.1.2. Les quotations

Les `quotations` partent d'une idée simple : placer une portion de code entre crochets sur la pile, sans l'évaluer, de façon à pouvoir l'appeller plus tard.

Elles rapellent sont les fonctions anonymes (lambdas) de Factor. Anonymes car elles représentent fondamentalement un mot sans nom.

Vous-vous rapellez du "Hello World" du précédent article ?

```
(scratchpad) "Hello world!" print
Hello world!
```

*http://docs.factorcode.org/content/article-shuffle-words.html

Alors comment allez-vous donc faire pour afficher “hello” à l’écran, disons…cinq fois consécutives ?

Une solution serait d’utiliser les shufflers, en particulier `dup` pourrait être utilisé ainsi:

```
(scratchpad) "hello" dup dup dup dup print print print print print
hello
hello
hello
hello
hello
```

Evidemment ce n’est pas très esthétique, et puis qu’en serait-il si nous devions afficher ce même message 100 fois ? 1000 fois ?

Exemple:

```
(scratchpad) 5 [ "hello" print ] times
hello
hello
hello
hello
hello
```

Pour créer une `quotation`, placez juste votre portion de code entre crochets `[ ... ]` et il sera alors placé sur la pile. Afin de faire quelque chose d’utile avec elle, on aura besoin d’un `combinator`.

0.1.3. Les combinators

Les combinators sont des mots qui prennent une/plusieurs quotation(s) en argument. Dans l’exemple précédent, `times` est un combinator qui prend en argument un entier (n) et une quotation, et qui éxécute cette quotation n fois.

Factor utilise ces quotations de manière intensive, que ce soit dans les conditions, les boucles, le parcourt de séquences, les espaces de noms, les fermetures, etc. Mais on s’égart un peu, et avant de voir tout cela de plus près, je voudrais tout d’abord vous montrer comment minimiser le recours aux shufflers en les remplaçant par un combinator judicieusement choisi.

0.1.4. Les combinators qui expriment une intention

Jusqu’à présent, nos portions de code ont été faciles à suivre, mais lorsque l’on commence à travailler sur des problèmes plus réalistes, les stack shufflers vont commencer à peupler votre code et le rendre illisible. Si nous pouvions représenter nos intentions plus proprement à l’aide d’un combinator, notre code deviendra plus lisible et vous pourrez alors vous occuper davantage de votre problème que d’organiser votre pile.

Pour illustrer ceci, je voudrais indroduire deux combinators simples :

1. `dip ( x quot -- x )`

   Appelle une quotation tout en cachant temporairement le dernier élément de la pile.

2. `keep ( x quot - x )`

   Appelle une quotation avec un élément sur la pile, et restaure cet élement une fois que la quotation termine son appel.

Bien qu'ils aient la même signature, ces deux combinators n'en ont pas moins des effets différents:

```
(scratchpad) 1 2 4 [ + ] dip

--- Data stack:
3
4
(scratchpad) clear
(scratchpad) 1 2 4 [ + ] keep

--- Data stack:
1
6
4
```

Même l'utilisation de ces deux combinators réduisent considérablement le nombre de shufflers dont votre code à besoin. Si vous êtes un peu curieux sur leur fonctionnement, ils profitent d'une pile auxiliaire (dite "pile de retenue") pour stocker temporairement des éléments tout en éxécutant leur quotation. Il existe quelques autres stacks combinators qui utilisent cette technique, n'oubliez pas de les étudier!

0.1.5. Cleave, Spread, Apply

Les combinators `cleave`, `spread`, et `apply` seront vos meilleures armes lorsque vous essayerez de réduire le nombre de shufflers tout en exprimant votre intention. La clé est qu'ils doivent exprimer une intention... si vous vous trouvez en train de coder et que ces combinators ne s'emboitent pas correctement, alors essayer autre chose.

Cleave Combinators Ils sont utilisés lorsque vous désirez appliquer plusieurs quotations au même ensemble d'éléments au dessus de la pile.

Par exemple, supposons que nous voulions calculer la moyenne d'une liste de nombres (array en Factor). L'algorithme est simple, vous ajoutez tous les nombres, puis vous divisez le total par le nombre de données (la longeur de la liste).

```
(scratchpad) { 1 2 3 } dup sum swap length / .
2
```

Nous pouvons éliminer le besoin de tous ces shufflers et meiux exprimer notre intention en utilisant un combinator pour obtenir le même effet:

- `bi ( x p q - )`

  Applique la quotation p sur x, puis applique la quotation q sur x

```
(scratchpad) { 1 2 3 } [ sum ] [ length ] bi / .
2
```

Les différents cleave combinators changent soit le nombre de quotations qui sont appliquées à nos éléments (`bi`, `tri`), soit le nombre d'éléments utilisés pour vos quotations (`bi` et `2bi`).

Spread Combinators Ils sont utilisés lorsque vous voulez appliquer une quotation différente sur différents éléments au dessus de la pile. Les spread combinators sont très proches de `dip`, mais sont un poil plus flexibles tout en exprimant aussi une intention.

Prenons un exemple, nous diposons de deux coordonnées de points sous la forme { x y }, et nous voulons extraire la l'abscisse (x) du premier point et l'ordonnée (y) du second pour former un nouveau point avec ces valeurs:

```
(scratchpad) { 1 2 } { 3 4 } swap first swap second 2array .
{ 1 4 }
```

On peut éliminer l'utilisation de ces stack shufflers et mieux exprimer notre intention en utilisant un spread combinator pour obtenir le même effet:

- `bi* ( x y p q – )`

  applique la quotation p sur x, puis la quotation q sur y

```
(scratchpad) { 1 2 } { 3 4 } [ first ] [ second ] bi* 2array .
{ 1 4 }
```

Lorsque vous désirez faire la même chose avec plus de deux quotation/éléments, l'utilisation des spread combinators élimine le besoin de `dip` ou de shufflers imbriqués, et votre code devient ainsi plus lisible.

Les différents spread combinators changent le nombre de quotations appliquées au nombre d'éléments correspondants sur la pile (`bi*` vs. `tri*`).

Apply Combinators Ils sont utilisés lorsque vous désirez appliquer une seule quotation à plusieurs éléments au dessus de la pile.

Supposons que nous ayons deux chaînes, chacune contenant un nom, et que vous vouliez savoir s'ils sont ou non identiques. Afin d'éviter de comparer la case de chaque caractère, nous décidons de convertir ces deux chaines en majuscules avant de les comparer:

```
(scratchpad) "john" "John" swap >upper swap >upper = .
t
```

Nous pouvons éliminer l'utilisation massive de stack shufflers et mieux exprimer notre intention en utilisant un apply combinator qui produira le même effet:

- `bi@ ( x y quot – )`

  applique la quotation à x, puis à y

```
(scratchpad) "john" "John" [ >upper ] bi@ = .
t
```

Les différents apply combinators changent le nombre d'éléments soumis à celle-ci (`bi@` vs. `tri@`).

Les combinators `cleave`, `spread`, et `apply` sont tous très proches les uns des autres, et si vous avez du mal à les retenir, essayez au moins de retenir leurs conventions de nommage :

- Si il n'y a pas de suffixe, c'est un `cleave`
- Si le suffixe est `*`, c'est un `spread`
- Si le suffixe est `@`, c'est un `apply`

Une fois que vous maîtrisez ces combinators, vous devriez être en mesure d'exprimer quasiment n'importe quelle combinaison de shufflers. A noter qu'il existe aussi des formes génériques pour tous les combinators qui peuvent prendre des arguments additionnels sur la pile. Si jamais vous remarquez que vous utilisez un peu trop les formes génériques, il y a de grandes chances pour qu'il faille revoir votre approche dans sa forme: il faut factoriser!

0.1.6. Quelques détails sur les types données

Voici quelques détails concernant les divers types de données de Factor.

Les séquences Dans les exemples utilisés plus haut avec `cleave` et `spread`, nous avons utilisé des séquences sans même en parler réèlement. Une séquence est une collection finie et ordonnée d'éléments. Tout type de données qui implémente la classe sequence mixin (çad qui connaît sa longeur et qui vous laisse prendre/changer un élément à un index spécifique) gagne l'abilité à utiliser les puissants opérateurs de séquences. Essayer de lire la documentation pour vous faire une idée de la manipulation des sequences et de leurs éléments.

Factor possède de nombreux types de sequence auxquels vous êtes peut-être déjà familier, comme les arrays (sequences de longueurs fixes, de contenu variable) et les vectors (de longueur et contenu variables), mais il y a encore d'autres types de données que vous ne soupconniez peut-être même pas d'être des sequences, comme les chaines (strings). Avec Factor, les chaînes sont tout simplement un array de code Unicode 5.0.

En utilisant à la fois les opérateurs sur les séquences et les combinators, vous pouvez créer toutes sortes de puissantes abstractions dont nous parlerons la prochaine fois. Voici un ou deux exemples pour vous mettre en appétit:

```
(scratchpad) { 1 2 3 4 5 } [ even? ] filter .
{ 2 4 }
(scratchpad) { 1 2 3 } [ . ] each
1
2
3
(scratchpad) "Hello" [ alpha? ] all? .
```

```
t
(scratchpad) "Hello!!!" [ alpha? ] all? .
f
```

Pensez qu'avec Factor, les sequences partent de l'indice zéro.

Les nombres Le dernier sujet que nous allons aborder concerne les nombres. Jusqu'ici, nous avons utilisé uniquement les entiers, mais Factor supporte aussi les rationnels, les flottants, et les complexes.

```
(scratchpad) 100 330 / .
10/33
(scratchpad) 5/4 1/2 + .
1+3/4
(scratchpad) 5/4 0.5 + .
1.75
(scratchpad) 2 5 rect> .
C{ 2 5 }
```

Les entiers possèdent une propriété vraiment très particulière... ceux qui sont positifs sont des séquences! Un entier positif (`n`) est une séquence de longeur n, dont les éléments sont les entiers positifs qui lui sont inférieur (càd de `0` à `n-1`). Cette propriété peut-être utile pour former des boucles ou pour contrôler le flux des instructions.

Ainsi, les deux assertions suivantes sont équivalentes:

```
{ 0 1 2 3 4 5 6 7 8 9 } [ . ] each
```

et

```
10 [ . ] each
```