Université Montpellier II
Maîtrise d'Informatique
Module Langage, Evaluation, Compilation — 3 heures

R. Ducournau – M. Lafourcade

juin 2004

Documents autorisés : notes de cours, polys, pas de livre.
2 parties indépendantes, notées respectivement sur 12 et 8.

1 Langage Forth

Forth est un langage très rudimentaire fondé sur la manipulation de piles et qui utilise une syntaxe postfixée (notation polonaise inverse). Toutes les opérations s'effectuent en supposant que les arguments se trouvent sur une pile de données.

Les instructions sont placées les unes derrières les autres et il est seulement nécessaire que les fonctions soient définies avant d'être utilisées. Forth ressemble ainsi à un Lisp inversé (les opérateurs sont situés après les arguments) et sans parenthèse. Par exemple :

- Empilement d'une constante sur la pile : **3**
- Addition de deux nombres (idem pour les autres opérations arithmétiques) : **3 4 +**
- Définition d'une fonction comme le carré (`:` et `;` sont des mots clés du langage) : **: carre dup * ;**
- Appel d'une fonction d'un argument : **2 carre**
- Appel d'une fonction de deux arguments : **3 4 distance**
- Composition de fonction : **1 2 + 2 carre distance** produit le même résultat que l'expression précédente.

Pour présenter la sémantique des opérations, on les commente en indiquant la manipulation de la pile de données qu'ils effectuent de la manière suivante : <**pile avant**> → <**pile après**>. Dans ce commentaire, on place le sommet de pile sur la droite. Par exemple, la fonction `+` prend 2 nombres, `x` et `y`, sur la pile et empile le résultat `z=x+y`. Ce que l'on note ainsi :

`+ / - x y → - z`.

Le `/` introduit un commentaire, et le signe `-` indique le fond de la pile invariant par l'opération. De même, l'exécution d'une constante s'écrit ainsi :

`<n> / - → - <n>`.

Il n'y a, dans ce langage, ni variable, ni paramètre et le seul type de donnée disponible est numérique. Les arguments sont directement récupérés sur la pile. Lorsqu'on a besoin de faire référence à des arguments, on utilise des opérateurs de manipulation de pile, dont voici les principaux :

```
dup   / - y → - y y        duplique le sommet de pile
swap  / - x y → - y x      échange les deux valeurs du sommet de pile
rot   / - x y z → - y z x  permutation des 3 premiers éléments
drop  / - x → -            supprime le sommet de pile
```

Par exemple, pour faire le calcul de la distance euclidienne ($z = sqrt(x \times x + y \times y)$) on écrit :

```
: distance
carre   / - x y → - x (y*y)
swap    / - x (y*y) → - (y*y) x
carre   / - (y*y) x → - (y*y) (x*x)
+       / - (y*y) (x*x) → - ((y*y)+(x*x))
sqrt    / - ((y*y)+(x*x)) → (sqrt((y*y)+(x*x)))
;
```

Opérateurs relationnels et booléens. On suppose en Forth, que le 1 représente vrai et que 0 représente faux. Les opérateurs relationnels sont traité de manière classique :

```
    =       / - x y → - 1 si x = y, 0 sinon
    <       / - x y → - 1 si x < y, 0 sinon
    <=      / - x y → - 1 si x <= y, 0 sinon
etc.
```

Structures de contrôle. Les structures de contrôles de Forth utilisent une syntaxe assez particulière du fait de la notation postfixéee. Le *if then* s'écrit ainsi :

```
<condition>+ if <instr-then>+ endif
```

Le *if then else* s'écrit ainsi :

```
<condition>+ if <instr-then>+ else <instr-else>+ endif
```

La boucle conditionnelle s'écrit ainsi :

```
begin <condition>+ while <instr>+ repeat
```

Question 1

Écrire en Lisp un évaluateur de Forth suivant les spécifications qui suivent :

- On considérera qu'un programme Forth est une liste Lisp formée d'atomes : on supposera que les opérateurs spéciaux comme `:` ou `;` sont des symboles Lisp normaux, ce qui n'est bien entendu pas le cas.
- L'évaluateur gère une pile explicite constituée par une liste d'atomes Lisp passée en paramètre des fonctions à définir. L'évaluateur traite donc l'expression dans un ordre naturel, de gauche à droite, en procédant aux transformations de la pile correspondant aux atomes rencontrés.
- L'évaluateur est constitué d'une fonction générale `eval-forth` qui évalue toute expression Forth et d'une fonction spécifique `eval-xxx` pour chaque opérateur prédéfini (`dup`, `if`, `+`, etc.), ainsi que d'une fonction spécifique `eval-forth-fun` pour appeler les fonctions définies.

1. spécifier l'implémentation de la pile ;
2. préciser les paramètres de la fonction `eval-forth`, sa valeur de retour, son schéma, au choix, d'itération ou de récursion (avec sa condition d'arrêt) : le mettre en œuvre sur la fonction `eval-forth` réduite à l'évaluation des constantes ;
3. définir le traitement des opérateurs de pile (`eval-dup`, `eval-rot`, etc.), en l'intégrant dans le squelette de la fonction `eval-forth` ;
4. faire de même pour les opérateurs arithmétiques (`eval-+`, `eval-=`, etc.) ;
5. définir le traitement des fonctions : définition (`: ;`) et application (`eval-forth-fun`), toujours en l'intégrant dans `eval-forth` ;
6. faire de même pour les structures de contrôle. NB Le schéma de la fonction `eval-forth` utilisé jusqu'ici peut se révéler mal adapté aux structures de contrôle : vous pouvez en proposer un autre en indiquant juste comment il s'obtient à partir du schéma précédent.

Question 2

On se pose le problème de la génération de code Forth à partir de Lisp. Pour cela on introduit une nouvelle opération de manipulation de pile.

```
stack   / - yk ... y1 <k> → - yk ... y1 yk
duplique le  <k>ième élément de pile au sommet
```

1. avec cette nouvelle opération, écrire la définition Forth correspondant à la fonction LISP (dont les opérations numériques ont été binarisées) :

```
(defun foo (x y)
       (+ (* 3 (* x x))
          (+ (* (- 5) (* x y)) (* 2 (* y y)))))
```

   Avant de quitter, la fonction ne doit pas oublier de nettoyer la pile, y compris de ses paramètres.

2. écrire en LISP la fonction `compile-forth` qui compile une fonction LISP quelconque à la restriction près qu'elle ne comporte aucune variable locale (ni `lambda`, `let` ou `labels`).

   Deux remarques :
   - un compteur pourra servir à mémoriser la position des paramètres par rapport au sommet de pile courant,
   - une fonction annexe récursive sera utile : elle pourra prendre en paramètre ce compteur et aura pour invariant de générer du code qui laisse toujours la pile dans l'état où elle était.

2 Appariement d'arbres

Soit 2 arbres binaires nommés respectivement le *motif* et la *donnée*. Ces arbres sont implémentés par des listes LISP imbriquées : les feuilles sont des atomes et les autres nœuds sont des cellules, dont les `car` et `cdr` pointent sur les sous-arbres.

Les feuilles du *motif* sont composées exclusivement de symboles, y compris le symbole `nil` (la liste vide). La *donnée* est quelconque.

Question 3

Un motif et une donnée s'apparient si les deux arbres ont la même structure de cellules et si les `nil` du motif correspondent à des `nil` de la donnée. Un appariement est alors constitué par une liste d'association qui fait correspondre à chaque symbole le sous-arbre correspondant dans la donnée.

Ecrire la fonction `apparier` qui prend en argument un motif et une donnée et retourne l'appariement s'il est possible, le mot-clé `:false` sinon.

Exemple : `(apparier '(x (y z) . w) '(3 (4 5) 6 7))` doit retourner '((x . 3) (y . 4) (z . 5) (w 6 7)) à l'ordre des symboles près.

Question 4

Une généralisation de l'appariement consiste à 1) introduire des constantes (on considérera uniquement des entiers) dans le motif, 2) tenir compte des occurrences multiples d'un même symbole.

Pour que le motif et la donnée s'apparient, il faut alors qu'à chaque constante du motif corresponde la même constante dans la donnée, et que les occurrences multiples d'un même symbole s'apparient au même sous-arbre dans la donnée. Ecrire la fonction *apparier++* qui réalise cet appariement.

Question 5

Une dernière généralisation consiste à autoriser l'appariement d'un symbole avec une sous-liste de la donnée : ces symboles seront préfixés par '*'.

Exemple : l'appariement de `(x (y *z w))` et de `(1 (2 3 4 5 6))` donnera `((x . 1)(y . 2)(z 3 4 5)(w . 6))`.

Ecrire la fonction `apparier*` qui réalise cet appariement. La solution est-elle toujours unique ?