IFT2035 – Travail pratique #2 – 2019.11.22

PROGRAMMATION FONCTIONNELLE

Marc Feeley

Le TP2 a pour but de vous faire pratiquer la programmation fonctionnelle et en particulier les concepts suivants : les fonctions récursives, la forme itérative, les continuations, et le traitement de liste. Vous devez réaliser en Scheme en style fonctionnel un interprète pour le langage du TP1 (avec quelques changements) et comparer l'approche fonctionnelle avec l'approche impérative.

Pour ce travail vous devez utiliser uniquement le sous-ensemble fonctionnel de Scheme (en particulier, vous ne devez pas utiliser les formes-spéciales "`set!`" et "`begin`" dans votre programme, ni les fonctions prédéfinies contenant un point d'exclamation, comme "`set-car!`" et "`vector-set!`"). Vous pouvez cependant utiliser les fonctions "`vector-set`" et "`list-set`" qui sont similaires à "`vector-set!`" et "`list-set!`" mais qui sont purement fonctionnelles (elles ne modifient pas le vecteur ou la liste mais retournent un nouveau vecteur ou liste). Pour utiliser ces fonctions, ainsi que d'autres fonctions prédéfinies sur les listes ("`fold`", "`fold-right`", "`iota`", "`make-list`", "`take`", "`drop`", "`last`", etc), vous aurez besoin de la version v4.9.3 de Gambit disponible sur gambitscheme.org et github.com/gambit/gambit.

Le but du travail étant de vous faire pratiquer les concepts de programmation fonctionnelle, utilisez les concepts que nous avons vus en classe lorsque c'est approprié (fonctions d'ordre supérieur, forme itérative, continuations, etc). L'élégance de votre codage est un facteur important dans l'évaluation de ce travail.

Vous avez à réaliser un programme en Scheme ainsi que rédiger un rapport contenant une analyse du programme et une comparaison du style fonctionnel et le style impératif.

---

1 Programmation

Le travail demandé consiste à réaliser en Scheme en style fonctionnel un interprète pour le langage défini pour le TP1 mais avec quelques changements. Votre interprète fera la lecture du code source du programme à exécuter, puis analysera ce code pour construire sa représentation sous forme d'un arbre de syntaxe abstraite (ASA), puis fera l'exécution du programme directement à partir de l'ASA.

Nous vous fournissons un squelette du programme (`petit-interp.scm` sur Studium) qui fait la lecture du code source, une analyse syntaxique partielle et une exécution de l'ASA partielle. Seul l'énoncé "`print`" avec une constante en paramètre est implanté, par exemple le programme "`print(42);`" exécutera correctement.

Vous ne devez pas changer la lecture du code source qui est dans une zone bien délimitée à la fin du squelette que nous vous fournissons. Vous pouvez modifier le reste du code comme vous le voulez, voire même le remplacer totalement par votre propre code, à condition que votre programme respecte la spécification dans cet énoncé (entre autre il faut que l'exécution donne la bonne sortie et que vous limitiez votre codage au sous-ensemble fonctionnel de Scheme). Dans le squelette le traitement se fait en utilisant le "Continuation Passing Style" (CPS) pour définir des fonctions qui "retournent" plus d'une valeur, mais si vous préférez un style où les fonctions retournent une structure de donnée contenant les résultats de la fonction vous pouvez récrire les fonctions pertinentes du squelette. La solution du professeur est à peu près 200 lignes de code plus longue que le squelette.

Votre interprète sera exécuté par Gambit avec la commande suivante :

```
% gsi petit-interp.scm < prog.c
```

Cela fera en sorte que le contenu du fichier `prog.c` soit lu par le programme `petit-interp.scm`, puis analysé, puis exécuté, ce qui affichera à la sortie les valeurs imprimées avec "`print`" ou bien un message d'erreur si `prog.c` est syntaxiquement invalide ou cause une erreur à l'exécution (comme une division par zéro).

2 Langage

Votre interprète doit accepter le langage spécifié par la grammaire suivante.

```
<program> ::= <stat>

<stat> ::= "if" <paren_expr> <stat>
         | "if" <paren_expr> <stat> "else" <stat>
         | "while" <paren_expr> <stat>
         | "do" <stat> "while" <paren_expr> ";"
         | ";"
         | "{" { <stat> } "}"
         | <expr> ";"
         | "print" <paren_expr> ";"

<expr> ::= <test>
         | <id> "=" <expr>

<test> ::= <sum>
         | <sum> "<"  <sum>
         | <sum> "<=" <sum>
         | <sum> ">"  <sum>
         | <sum> ">=" <sum>
         | <sum> "==" <sum>
         | <sum> "!=" <sum>

<sum> ::= <mult>
        | <sum> "+" <mult>
        | <sum> "-" <mult>

<mult> ::= <term>
         | <mult> "*" <term>
         | <mult> "/" <term>
         | <mult> "%" <term>

<term> ::= <id>            ;; les identificateurs ont une longueur quelconque
         | <int>           ;; les constantes entieres ont une longueur quelconque
         | <paren_expr>

<paren_expr> ::= "(" <expr> ")"
```

Donc, par rapport au langage du TP1, les énoncés `break` et `continue` ont été retirés du langage, les opérateurs `/` et `%` ne sont pas contraints à diviser par 10 et les identificateurs et constantes entières ont une longueur quelconque.

Tout comme pour le TP1 les opérateurs "`/`" et "`%`" correspondent à la division entière et le reste après division, c'est-à-dire l'expression "`14/4`" s'évalue à 3 et "`14%4`" s'évalue à 2 (ce sont les mêmes calculs que "`(quotient 14 4)`" et "`(remainder 14 4)`" en Scheme). Les calculs se font donc exclusivement avec des nombres entiers. D'autre part, il n'y a pas de limite sur la taille des entiers calculés (à part la mémoire disponible sur l'ordinateur). Cela ne pose pas vraiment un problème d'implantation car le langage Scheme ne place pas de limite sur la taille des entiers. Il faut cependant que l'interprète détecte les cas de division par zéro et termine l'exécution du programme après avoir affiché un message explicatif.

Voici un exemple d'utilisation de l'interprète. Si le fichier `prog.c` contient

```
{
  nb = 2;
  i = 1;
  while (i < 9) {
    nb = nb*nb;
    i = i+1;
    print(nb);
  }
}
```

alors l'exécution de

```
% gsi petit-interp.scm < prog.c
```

doit donner la sortie

```
4
16
256
65536
4294967296
18446744073709551616
340282366920938463463374607431768211456
115792089237316195423570985008687907853269984665640564039457584007913129639936
```

Votre analyseur syntaxique (la fonction `parse` du squelette) doit construire un ASA du programme source. En Scheme c'est assez naturel d'utiliser des S-expressions pour les ASA. Une S-expression c'est une liste qui contient comme premier élément un symbole qui indique la nature de ce noeud de l'ASA, et les éléments restants sont les attributs et/ou enfants de ce noeud. Par exemple, la S-expression `(PRINT (INT 42))` est l'ASA du programme source "`print(42);`". Un exemple plus complexe est le programme `prog.c` ci-dessus dont l'ASA est :

```
(SEQ (EXPR (ASSIGN "nb" (INT 2)))
     (SEQ (EXPR (ASSIGN "i" (INT 1)))
          (SEQ (WHILE (LT (VAR "i") (INT 9))
                      (SEQ (EXPR (ASSIGN "nb" (MUL (VAR "nb") (VAR "nb"))))
                           (SEQ (EXPR (ASSIGN "i" (ADD (VAR "i") (INT 1))))
                                (SEQ (PRINT (VAR "nb"))
                                     (EMPTY)))))
               (EMPTY))))
```

L'analyseur syntaxique est incomplet, par exemple il ne traite pas les expressions contenant un opérateur, comme l'addition et la multiplication. Pour faire une analyse de ces expressions il faut tout d'abord étendre la fonction `next-sym` pour que ces opérateurs soient reconnus (similairement au traitement de `(`, `)` et `;`). De plus il faut étendre les fonctions `<sum>`, `<mult>`, etc pour reconnaître ces catégories syntaxiques. Une bonne compréhension du fonctionnement de la fonction `<expr_stat>` sera utile pour vous inspirer pour compléter l'écriture de ces fonctions.

Le coeur de votre interprète (la fonction `execute` du squelette) fait l'exécution du programme à partir de son ASA. Cette fonction va donc parcourir récursivement la S-expression pour exécuter les énoncés et expressions qui s'y trouvent. L'interprète doit gêrer une structure d'environnement (par exemple une liste d'association) qui indique la valeur de chaque variable du programme. L'interprète doit aussi accumuler (par exemple dans une chaîne de caractères) toutes les sorties effectuées par les "`print`" exécutés pour qu'à la fin de l'exécution toutes les sorties accumulées soient affichées (par la fonction `main` du squelette).

Il est conseillé de faire un développement incrémental des diverses fonctionnalités en commençant par les plus simples. Par exemple, ajouter un cas à la fonction `exec-expr` pour faire l'évaluation des additions, puis ajouter un cas pour les multiplications, et ainsi de suite. Le traitement des variables et des énoncés `if`, `while`, etc (par la fonction `exec-stat`) est plus complexe, donc c'est probablement mieux de s'y attaquer en dernier.

Votre interprète doit être robuste et ne pas terminer abruptement avec une erreur autre que l'affichage de messages d'erreur de syntaxe ou de division par zéro du programme interprété.

Votre code Scheme doit être entièrement contenu dans un fichier dont le nom est "`petit-interp.scm`". Vous devez utiliser le fichier "`petit-interp.scm`" de la page Studium du cours comme point de départ et seulement modifier la première section. Pour faciliter le déboguage, l'exécution de l'interprète peut se faire en lançant `gsi` avec la commande "`gsi -:dar petit-interp.scm`", ou vous pouvez faire "`chmod +x petit-interp.scm`" suivi de "`./petit-interp.scm`". Cela vous donnera une REPL de déboguage lorsqu'il y a une erreur d'exécution dans votre code. La commande "`,?`" vous donnera de l'aide sur les commandes de déboguage. N'oubliez pas aussi que l'utilisation de "`trace`" et d'appels à "`pp`" peut aider à comprendre l'exécution de votre code. Vous avez avantage à utiliser `emacs` pour éditer vos fichiers et faire le déboguage. Les instructions relatives à l'utilisation de Gambit dans `emacs` sont données ici : http://www.iro.umontreal.ca/~gambit/doc/gambit.html#Emacs-interface . Le package "geiser" pour emacs peut également vous être utile : https://gitlab.com/jaor/geiser .

3 Rapport

Vous devez rédiger un rapport en format PDF qui :

1. Explique brièvement le fonctionnement général du programme (maximum de 1 page au total).
2. Explique comment les problèmes de programmation suivants ont été résolus (en 2 à 3 pages au total) :
   (a) comment se fait l'analyse syntaxique du programme interprété
   (b) comment se fait l'interprétation du programme interprété (autant les énoncés que les expressions)
   (c) comment se fait l'interprétation des affectations aux variables et la gestion de l'environnement
   (d) comment se fait l'interprétation des énoncés "`print`"
   (e) comment se fait le traitement des erreurs

3. Compare votre expérience de développement avec le TP1 (en 1 à 2 pages au total). Combien de lignes de code ont vos programmes C et Scheme? Sans tenir compte de votre niveau de connaissance des langages C et Scheme, quels sont les traitements qui ont été plus faciles et plus difficiles à exprimer en Scheme? Indépendamment des particularités syntaxiques de Scheme, pour quelles parties du programme l'utilisation du style de programmation fonctionnel a-t'il été bénéfique et pour quelles parties détrimental? Pour quelles parties avez vous utilisé des récursions en forme itérative? Pour quelles parties avez vous utilisé des continuations?

4 Évaluation

- Ce travail compte pour 15 points dans la note finale du cours. Indiquez vos noms clairement au début du programme. **Vous devez faire le travail par groupes de 2 personnes. Vous devez confirmer la composition de votre équipe (noms des coéquipiers) au démonstrateur. Si vous ne trouvez pas de partenaire d'ici quelques jours, parlez-en au démonstrateur.**
- Le programme sera évalué sur 8 points et le rapport sur 7 points. Assurez-vous de prévoir toutes les situations d'erreur et de bien tester votre programme.
- Vous devez remettre un fichier "`.tar`" ou "`.zip`" qui contient uniquement deux fichiers : votre rapport (qui doit se nommer `rapport.pdf`) et le programme (qui doit se nommer `petit-interp.scm`). La remise doit se faire au plus tard à 23h55 vendredi le 13 décembre sur le site Studium du cours. En supposant que vos deux fichiers sont dans le répertoire `tp2`, vous pouvez créer le fichier "`.tar`" avec la commande "`tar cf tp2.tar tp2`".
- L'élégance et la lisibilité du code, l'exactitude et la performance, la lisibilité du rapport, et l'utilisation d'un français sans fautes sont des critères d'évaluation.

5 Annexe

Le programme suivant peut vous être utile pour tester votre interprète :

```
{
  n = 1000;
  one = 1;
  while (n>0) { one = one*10; n = n-1; }
  a = one;
  x = one*one/2;
  r = x; while ((n = (r+x/r)/2) < r) r = n;
  t = one/4;
  p = 1;
  while (a != r) {
    x = a*r;
    y = (a+r)/2;
    z = y-a;
    a = y;
    r = x; while ((n = (r+x/r)/2) < r) r = n;
    t = t - p*z*z/one;
    p = p*2;
  }
  x = a+r;
  print(x*x/(4*t));
}
```