LICENCE INFORMATIQUE L3 2007-08 — TP 13

Programmation Scheme Avancée et Interprétation

Exercice 13.1 Téléchargez le fichier `mfcalc-lexer.rkt`. Il contient l'analyseur lexical pour *mfcalc*, la calculette interactive. Une évaluation de ce fichier produira automatiquement le fichier de test `test-lex.dat` qui contient une suite d'unités lexicales [lexèmes]. Essayez d'abord la fonction `(test2-lexer)` pour lire tous les lexèmes. Programmez ensuite une fonction `(exo13.1)` pour lire ces unités une à une et retourner la liste de tous les nombres contenus dans ce fichier, dans l'ordre d'apparition :

```
> (exo13.1)
TEST : les nombres du fichier "test-lex.dat"
(2 100.5 1 100.0 0.1)
```

Exercice 13.2 Téléchargez le fichier `mfcalc4.rkt`. Il contient un analyseur syntaxique sans conflits pour la calculette interactive. Testez-le, en n'oubliant pas de terminer vos demandes de calcul par un point-virgule ! Pour quitter la calculette, cliquer dans le bouton <eof> situé en bas à droite du toplevel…

a) Introduisez des variables `pi = 3.14`, `L = 2` et `g = 9.81`. Que vaut $2\pi\sqrt{\frac{L}{g}}$?

b) Rajoutez à la calculette une variable spéciale @ signifiant *@nswer* : la dernière réponse. Exemple :

```
? 2^3+4;
==> 12
? @-3;          <-- @ représente la dernière réponse, ici 12
==> 9
```

Exercice 13.3 Un Scheme [presque] sans parenthèses… On vous demande d'écrire un analyseur lexical [*lexer*] et un analyseur syntaxique [*parser*] pour un mini-langage MISSP [MIcro Scheme Sans Parenthèses] ayant une sémantique analogue à celle de Scheme mais une syntaxe infixée. Le *parser* fera office de traducteur en produisant du code SCHEME équivalent au code infixé. Voici quelques exemples de traductions, nous vous laissons construire la grammaire [préférez les récursivités gauches !]. Vous pouvez **compléter le fichier** `missp.rkt` fourni sur le Web !

`(- (expt x 3) (* 2.5 x))`	`x^3-2.5*x`
`(if (= x y) (+ x 1) (- y 2))`	`if x=y then x+1 else y+2 endif`
`(let ((x 1) (y (+ x 1))) (+ x y -1))`	`let x:=1 and y:=x+1 in { x+y-1 }`
`(letrec ((fac (lambda (n) ...))) (fac 5))`	`letrec fac:=proc(n){...} in { fac(5) }`
`(lambda (x1 ... xN) e1 ... eN)`	`proc(x1,...,xN){ e1 ; ... ; eN }`
`(begin e1 ... eN)`	`{ e1 ; ... ; eN }` ← *un « bloc »*
`(set! x (+ (fac 5) 2))`	`x := fac(5)+2`
`(let ((g253 n)) (set! n (+ n 1)) g253)`	`n++` ← *à la C/Java…*
`(begin (set! n (- n 1)) n)`	`--n` ← *à la C/Java…*

Le point-virgule est donc plus un *séparateur* d'instructions [comme en Pascal ou Maple] qu'un *terminateur* d'instruction [comme en C ou Java]. Un programme n'est autre qu'un bloc d'instructions précédé d'un nom de programme, comme le montre l'exemple ci-dessous.

Voici en effet la traduction d'une suite d'instructions avec des récursivités [dont du CPS] vers du Scheme [non optimisé, avec des `begin` inutiles, etc]. Le compilateur [i.e. le *parser*] traduit le fichier `test.msp` fourni sur le Web :

```
foo ::= let n:=10 and cont:=id
          in { letrec fac := proc(n,f) { if n=0 then f(1) else fac(n-1,proc(r){ f(r*n) }) endif }
                 in {  writeln(fac(n,cont)) } }
```

lexer + parser dans le même fichier !

```
(define (foo)
  (let ((n 10) (cont id))
    (begin
      (letrec ((fac
                (lambda (n f)
                  (begin
                    (if (= n 0)
                      (f 1)
                      (fac (- n 1) (lambda (r) (begin (f (* r n))))))))))
        (begin (writeln (fac n cont)))))))
```

Le cours suivant se chargera d'exécuter le code Scheme produit via la programmation d'un interprète, mais dans l'immédiat on peut le faire exécuter directement [modulo l'incorporation de `writeln` et autres librairies initiales] par Racket… Evidemment, si vous préférez enlever dès l'analyse [ou ensuite !] les `begin` *inutiles* et faire toute autre optimisation, personne en vous en blâmera ☺…

A vous de jouer !…

Exécution :

```
> (define (writeln x)
    (printf "~a\n" x))
> (foo)
3628800
```