Année 2004 – 2005 Session de septembre 2005
LST Informatique
Épreuve de Programmation Fonctionnelle et Symbolique (INF207)
Date : le 2 septembre 2005 Durée : 1h30
Tous documents légaux autorisés.
Votre voisin n'est pas un document
Vous devez répondre directement sur le sujet
Épreuve de M Strandh

Répondre directement sur le sujet dans les cadres prévus à cet effet. Aucune réponse ou partie de réponse en dehors des cadres ne sera prise en compte.

Écrire votre numéro d'anonymat sur chaque feuille du sujet.

Rappel : Soyez bref et clair ! Personne n'est impressionné par la quantité !

Bon courage !

Numéro d'anonymat :

Exercice 1 (12 p)

On souhaite écrire une fonction nommée `iota` inspirée de la fonction ι du langage APL. Cette fonction doit prendre un seul paramètre qui doit être un entier non négatif, disons n. La fonction `iota` doit renvoyer une liste des n entiers de 0 à $n-1$. Pour $n = 0$ elle renvoie la liste vide (ou `NIL`).

Exemples :

```
CL-USER> (iota 10)
(0 1 2 3 4 5 6 7 8 9)
CL-USER> (iota 0)
NIL
CL-USER> (iota 5)
(0 1 2 3 4)
```

a. Écrire une version récursive de la fonction `iota` selon cette spécification. Utiliser la macro Common Lisp `labels` (voir page 8)(5p).

Réponse :

b. Écrire une version itérative de la fonction `iota` selon la même spécification. Utiliser pour cela la macro Common Lisp `dotimes` (2p).

Réponse :

On souhaite étendre la fonction `iota` de manière compatible (le code existant doit continuer à marcher) en admettant un deuxième paramètre qui indique le pas (également non négatif) d'itération comme ceci :

```
CL-USER> (iota 10)
(0 1 2 3 4 5 6 7 8 9)
CL-USER> (iota 10 2)
(0 2 4 6 8 10 12 14 16 18)
CL-USER> (iota 5 1)
(0 1 2 3 4)
CL-USER> (iota 4 10)
(0 10 20 30)
```

c. Écrire une version de la fonction `iota` étendue de cette façon. Pour cela, supposer que l'ancienne fonction ait été renommée `simple-iota`, puis écrire la version étendue en utilisant `simple-iota` et la fonction Common Lisp `mapcar` (2p).

(réponse sur la page suivante)

Numéro d'anonymat :

Réponse :

Une autre extension utile de la fonction `iota` est de permettre un troisième paramètre qui indique par quel élément la liste renvoyée commence, comme ceci :

```
CL-USER> (iota 10 6 5)
(5 11 17 23 29 35 41 47 53 59)
```

d. Écrire une version entièrement nouvelle de la fonction `iota`, également compatible avec les deux versions précédentes. Pour cela, utiliser la macro `loop` (3p).

Réponse :

Exercice 2 (8 p)

Dans une application de type système d'information géographique, on souhaite pouvoir déterminer si un point arbitraire se trouve dans une zone donnée. Une telle zone peut être *élémentaire* ou *composée*. Il y a trois types de zones élémentaires : *polygonales*, *circulaires* et des zones qui sont définies par une *fonction caractéristique* qui, étant donné un point renvoie `t` si le point est dans la zone et `nil` sinon.

Une zone polygonale est définie par une suite de points qui constituent les coins du polygone. Une zone circulaire est définie par un centre et un rayon.

Il y a deux types de zones composées : des zones définies par l'*union* d'autres zones (élémentaires ou composées) et des zones définies par l'*intersection* d'autres zones (élémentaires ou composées).

a. Exprimer cette situation grâce à des définitions de classes Common Lisp et écrire la définition de la fonction `point-in-zone-p` (4p).

(réponse sur la page suivante)

Numéro d'anonymat :

Réponse :

(suite sur la page suivante)

Numéro d'anonymat :

b. Normalement, le calcul de l'union ou de l'intersection de deux zones quelconques donne une zone composée. L'exception est que l'intersection entre deux zones polygonales doit donner une zone polygonale. De plus, on souhaite permettre des optimisations futures sous la forme de types de zones supplémentaires. Écrire les définitions des deux fonctions `zone-union` et `zone-intersection`. Vous pouvez omettre le calcul de l'intersection entre deux polygones (4p).

Réponse :

Special Operator FLET, LABELS, MACROLET

Syntax :

labels ((function-name lambda-list [[local-declaration* — local-documentation]] local-form*)*) declaration* form*

=¿ result*

Arguments and Values :

function-name—a function name.

name—a symbol.

lambda-list—a lambda list

local-declaration—a declare expression ; not evaluated.

declaration—a declare expression ; not evaluated.

local-documentation—a string ; not evaluated.

local-forms, forms—an implicit progn.

results—the values of the forms.

Description :

labels defines local functions, and execute forms using the local definitions. Forms are executed in order of occurrence.

The body forms (but not the lambda list) of each function created by labels and each macro created by macrolet are enclosed in an implicit block whose name is the function block name of the function-name or name, as appropriate.

The scope of the declarations between the list of local function/macro definitions and the body forms in flet and labels does not include the bodies of the locally defined functions, except that for labels, any inline, notinline, or ftype declarations that refer to the locally defined functions do apply to the local function bodies. That is, their scope is the same as the function name that they affect. The scope of these declarations does not include the bodies of the macro expander functions defined by macrolet.

flet

flet defines locally named functions and executes a series of forms with these definition bindings. Any number of such local functions can be defined.

The scope of the name binding encompasses only the body. Within the body of flet, function-names matching those defined by flet refer to the locally defined functions rather than to the global function definitions of the same name. Also, within the scope of flet, global setf expander definitions of the function-name defined by flet do not apply. Note that this applies to (defsetf f ...), not (defmethod (setf f) ...).

The names of functions defined by flet are in the lexical environment; they retain their local definitions only within the body of flet. The function definition bindings are visible only in the body of flet, not the definitions themselves. Within the function definitions, local function names that match those being defined refer to functions or macros defined outside the flet. flet can locally shadow a global function name, and the new definition can refer to the global definition.

Any local-documentation is attached to the corresponding local function (if one is actually created) as a documentation string.

labels

labels is equivalent to flet except that the scope of the defined function names for labels encompasses the function definitions themselves as well as the body.

```
(defun recursive-times (k n)
  (labels ((temp (n)
             (if (zerop n) 0 (+ k (temp (1- n))))))
    (temp n))) =>  RECURSIVE-TIMES
(recursive-times 2 3) =>  6

(defun integer-power (n k)
  (declare (integer n))
  (declare (type (integer 0 *) k))
  (labels ((expt0 (x k a)
             (declare (integer x a) (type (integer 0 *) k))
             (cond ((zerop k) a)
                   ((evenp k) (expt1 (* x x) (floor k 2) a))
                   (t (expt0 (* x x) (floor k 2) (* x a)))))
           (expt1 (x k a)
             (declare (integer x a) (type (integer 0 *) k))
             (cond ((evenp k) (expt1 (* x x) (floor k 2) a))
                   (t (expt0 (* x x) (floor k 2) (* x a))))))
   (expt0 n k 1))) =>  INTEGER-POWER
```

Affected By : None.

Exceptional Situations : None.