Université Bordeaux 1 LST Mention Informatique Semestre 5 2008-2009

Programmation Fonctionnelle et Symbolique : feuille 4

Zones géométriques

Dans leur article `http://haskell.org/papers/NSWC/jfp.ps` de 1994, Paul Hudak et Mark P. Jones rapportent une expérience rare de comparaison entre différents langages de programmation. Le logiciel implémenté manipule des zones géométriques. Nous allons étudier une version simplifiée (mais pas tant que ça) de ce logiciel.

1. Étudier le programme `zones-initial.lisp`. Lire les commentaires. S'assurer d'avoir compris son fonctionnement.
2. Lire la page de la `HyperSpec` concernant la fonction `abs`. Expliquer pourquoi et comment marche la fonction `make-disk`.
3. Écrire un jeu de tests qui vérifie le bon fonctionnement des fonctions du logiciel. Ici, un jeu de tests est une suite d'expressions de type (`assert <expression-qui-doit-renvoyer-vrai>`). On considère donc que le test passe si `NIL` est retourné (c'est la valeur retournée par `assert`) et que le test ne passe pas si une condition (exception) est signalée.
4. S'inspirer de la constante `+nowhere+` pour rajouter une constante `+everywhere+` dont la valeur doit être une zone contenant tous les points du plan.

Dans ce qui suit pour chaque nouvelle fonction on ajoutera dans le jeu de tests, un test qui permet de vérifier la correction de l'implémentation.

5. S'inspirer de la fonction `zone-intersection` pour rajouter une fonction `zone-union` dont le but est de retourner l'union de deux zones passées en arguments.
6. S'inspirer des fonctions `zone-intersection` et `zone-union` pour écrire une fonction `zone-difference` dont le but est de retourner la différence entre deux zones passées en arguments.
7. Écrire une fonction `make-rectangle (width heigth)` qui construit une zone rectangulaire de largeur `width` et de hauteur `heigth` dont le sommet inférieur gauche est en `(0,0)`.
8. Rajouter une assertion à la fonction `make-disk` pour vérifier que l'argument est un nombre réel.
9. S'inspirer de la fonction `move-zone` pour rajouter une fonction `scale-zone`. Cette nouvelle fonction aura deux paramètres, le premier est la zone à modifier, et le deuxième est un nombre complexe $\lambda_1 + i\lambda_2$. La fonction applique à la zone la transformation $(x, y) \mapsto (\lambda_1 x, \lambda_2 y)$.
10. Sans "casser" le code existant, modifier la fonction `make-disk` pour qu'elle accepte un paramètre facultatif indiquant le centre du cercle (le point `(0,0)` par défaut).
11. S'inspirer des fonctions `move-zone` et `scale-zone` pour rajouter une fonction `rotate-zone`, dont le but est la rotation d'une zone autour de l'origine. Cette nouvelle fonction aura deux paramètres, le premier est la zone à modifier, et le deuxième est un nombre réel qui est une mesure (en radians) de l'angle de rotation.
12. Modifier la fonction `scale-zone` pour qu'elle accepte un paramètre facultatif (dont la valeur par défaut est `(0,0)`) indiquant l'origine de la mise en échelle. Une façon

d'implémenter cette fonction est de déplacer la zone par l'opposé de la valeur du nouveau paramètre, puis d'effectuer la mise en échelle par rapport à `(0,0)` et finalement de déplacer le résultat à son endroit initial.

13. De la même manière qu'avec la fonction `scale-zone`, rajouter un paramètre facultatif indiquant l'origine de la rotation.
14. Avec notre définition de 'jeu de tests', il est difficile de tester si une fonction signale correctement une exception (ce qui est le cas de la fonction `make-disk` si la valeur du premier argument n'est pas un réel) simplement parce que si une condition est signalée, on considère que le jeu de tests ne passe pas. Étudier le fonctionnement (dans la `HyperSpec` et par expérimentation) de `ignore-errors`, et concevoir une façon de signaler une condition si et seulement si une condition n'est pas signalée par une expression.
15. Utiliser le résultat de l'exercice précédent pour rajouter dans votre jeu de tests, une vérification que `make-disk` signale une exception lorsqu'elle est appelée avec une valeur qui n' est pas un nombre réel.
16. Rajouter des assertions opportunes dans l'ensemble des fonctions du logiciel pour vérifier le bon type ou la bonne valeur des arguments. Se servir de `ignore-errors` pour rajouter des tests dans votre jeu de tests pour vérifier le bon fonctionnement de ces assertions.
17. Écrire une fonction `make-half-plane` qui construit une zone correspondant à tous les points à droite d'un vecteur d'un point p_1 à un point p_2. Un point p est situé à droite du vecteur de p_1 à p_2 ssi $\det(p_2 - p_1, p - p_1) \leq 0$. On pourra donc écrire une fonction auxiliaire `determinant`.
18. Écrire une fonction `make-triangle` qui construit une zone correspondant à un triangle défini par trois points. Se servir du fait qu'un triangle peut être considéré comme l'intersection de trois demi plans (fermés) dont les frontières sont les supports des côtés.
19. Écrire une fonction `make-sector` qui, étant donnés deux nombres réels α, β, fabrique une zone $S(\alpha, \beta)$ correspondant à un secteur de rayon infini : $z \in S(\alpha, \beta)$ ssi $z = 0$ ou ($\frac{z}{|z|}$ appartient à l'arc du cercle unité parcouru par un point qui part de $e^{i\alpha}$, tourne dans le sens direct, et s'arrête au point $e^{i\beta}$).
    Attention : $S(\frac{\pi}{4}, \pi)$ est une intersection de demi-plans mais $S(\frac{\pi}{4}, -\frac{\pi}{4})$ est une union de demi-plans.
20. Rajouter un paramètre facultatif à la fonction `make-sector` qui indique l'origine du secteur.
21. Rajouter un deuxième paramètre facultatif à la fonction `make-sector` indiquant le rayon du secteur (la valeur par défaut, `NIL`, indique un rayon infini).

```lisp
;;; A zone is represented as a function that takes a point in 2-dimensional
;;; space as a parameter (represented as a complex  number), and returns T
;;; if and only if the point is in the zone, and NIL otherwise.

;;; To determine whether a point is in a zone, just call this function
(defun point-in-zone-p (point zone)
  (funcall zone point))

;;; A zone that contains no points.  A point is never in this zone.
(defparameter +nowhere+
  (lambda (p)
    (declare (ignore p))
    nil))

;;; Create a circular zone with center in (0,0) with the indicated radius.
(defun make-disk (radius)
  (lambda (p)
    (<= (abs p) radius)))

;;; Given two zones, create a zone that behaves as the intersection of the two.
(defun zone-intersection (zone1 zone2)
  (lambda (p)
    (and (point-in-zone-p p zone1)
         (point-in-zone-p p zone2))))

;;; Given a zone, move it by a vector indicated as a complex number
;;; passed as the argument.
(defun move-zone (zone vector)
  (lambda (p)
    (point-in-zone-p (- p vector) zone)))
```