## Jeux de test

```
let test_fact0 () = (* assert *)
  begin
    assert(fact 0 = 1);
    assert(fact 6 = 720);
  end

let unit_test name condition =
  begin
    print_endline ("Testing " ^ name);
    if condition then print_endline "Passed"
    else begin print_endline "Failed";
               flush stdout;
         end
  end
let test_fact () =
  begin unit_test "fact 0" (fact 0 = 1);
        unit_test "fact 6" (fact 6 = 720);
        unit_test "toto" (fact 6 = 720);
```

## Comparaison de langages de programmation

"Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Proto-typing Productivity de 1994, P. Hudak et M. P. Jones
comparaison entre différents langages de
Le logiciel implémenté manipule des zones géométriques
une version simplifiée (mais pas tant que ça) de ce logiciel

## Zones du plan représentées par leur fonction caractéristique

On représente une telle zone par sa fonction caractéristique, c'est-à-dire par la fonction booléenne qui prend un point en argument, et retourne `true` si le `point` appartient à la zone et `false` sinon.
Ainsi, la zone représentant le plan tout entier est représentée par la variable `everywhere` suivante:

```
# let everywhere = fun point -> true;;
val everywhere : 'a -> bool = <fun>
```

On peut forcer le type des variables et des paramètres en les faisant suivre de `:` suivi du nom du type.

```
# let everywhere : zone = fun point -> true;;
val everywhere : zone = <fun>
```

## Appartenance d'un point à une zone

```
# let c_origin = make_point 0. 0.
# point_in_zone_p c_origin nowhere;;
- : bool = false
# point_in_zone_p c_origin everywhere;;
- : bool = true
```

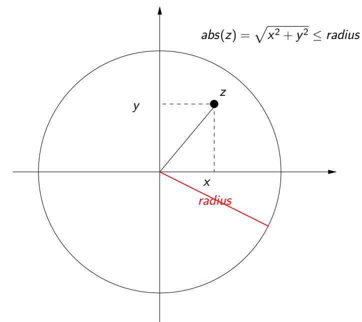
```
# let point_in_zone_p point zone = zone point;;
val point_in_zone_p : 'a -> ('a -> 'b) -> 'b = <fun>
```

En forçant les types:

```
let point_in_zone point (zone:zone) = zone point;;
val point_in_zone : point -> zone -> bool = <fun>
```

## Exemple de zone: disque

disque de rayon *radius* centré à l'origine.




```
let make_disk0 radius =
  fun point -> c_abs point <= radius
```

# Exemple de transformation: translation

Translation de vecteur $\overrightarrow{u}$ d'affixe $z_u$.
Le point $P'$ d'affixe $z'$ est dans la nouvelle zone si son antécédent $P$ d'affixe $z$ est dans la zone.

$$z' = z + z_u \Rightarrow z = z' - z_u$$

```
let translate c vector = c_sum c vector
```

Pour qu'un point soit dans la zone translatée, il faut que le point obtenu par la translation inverse soit dans la zone initiale:

```
let translate_zone zone vector =
  fun p -> point_in_zone_p
             (translate p (c_opp vector)) zone
```

## Visualisation des zones

À voir en TD machine.

## Déclaration d'un type enregistrement

Au lieu d'utiliser des tuples dans lesquels l'ordre des éléments a une importance, on peut utiliser des enregistrements dans lesquels les éléments sont nommés.

```
type <nom_de_type> =
{
  label1 : type1;
  label2 : type2;
  ...
  labeln : typen;
}
```

On peut ainsi redéfinir le type `complex` vu précédemment:

```
type complex = { real : float; imag : float; }
```

## Constructeurs et accesseurs

Une valeur du type `nom_de_type` s'écrit:

```
{
  label1 = valeur1;
  label2 = valeur2;
  ...
  labeln = valeurn;
}
```

**L'ordre des lignes n'a pas d'importance**.

## Exemple d'enregistrement

Par exemple, pour le type `complex` :

```
# { real = 1.0; imag = 0. };;
- : complex = {real = 1.; imag = 0.}
# let i = { real = 0.; imag = 1.0 };;
val i : complex = {real = 0.; imag = 1.}
# let c = { imag = 3.; real = 1.0 };;
val c : complex = {real = 1.; imag = 3.}
```

Étant donnée une valeur de type enregistrement, on *accède* à un des champs en suffixant la valeur par le champs voulu. Exemple:

```
# { real = 1.0; imag = 0. }.real;;
- : float = 1.
# i.imag;;
- : float = 1.
```

# Types récursifs (ou inductifs)

La récursivité est utilisable dans la définition des types. Ceci permet en particulier de construire des types infinis.
On peut par exemple représenter les listes d'entiers[4].

```
# type intlist = NI | CI of int * intlist;; (* NI: liste d
type intlist = NI | CI of int * intlist
# CI(1, CI(2, CI(3, NI)));;
- : intlist = CI (1, CI (2, CI (3, NI)))
# NI;;
- : intlist = NI
# type 'a truclist = NT | CT of 'a * 'a truclist;;
type 'a truclist = NT | CT of 'a * 'a truclist
# NT;;
- : 'a truclist = NT
# CT('a', CT('b', CT('c', NT)));;
- : char truclist = CT ('a', CT ('b', CT ('c', NT)))
```

[4] Définition en fait inutile, puis qu'il existe un type prédéfini pour les listes que nous verrons page 81

## Listes

Comment définir un type liste générique (liste d'éléments appartenant tous à un même type non fixé)?

```
type 'a mylist = Nil | C of 'a * 'a mylist
```

Exemples de fonctions utilisant ce type

- `length`
- `make_list`
- `concat`