Variable anomyme

Le caractère `_` (souligné ou "tiret du 8" sur un clavier AZERTY) correspond à une variable anonyme.
On peut l'utiliser

- à gauche du `=` dans un `let` ou `let ... in`

```
# let x, _, z = 1,2,3 in x, z;;
- : int * int = (1, 3)
```

- dans un motif d'un `match`

```
let est_une_figure carte =
  match carte with
    As _ -> false
  | Numero(_, _) -> false
  | _ -> true
```

- dans un fichier `.ml` pour mémoriser une expression

```
let _ = couleur_carte (Numero(7, Pique))
let _ = couleur_carte (As Coeur)
```

Regroupement des clauses d'un `match`

```
match carte with
  As c -> c
| Roi c -> c
| Dame c -> c
| Valet c -> c
| Numero(_, c) -> c

match carte with
  As c | Roi c | Dame c | Valet c | Numero(_, c) -> c
```

```
match carte with
  As _ -> false
| Numero(_,_) -> false
| _ -> true
```

```
match carte with
  As _
| Numero(_,_) -> false
| _ -> true
```

Représentation d'un point du plan par un complexe

plan muni d'un repère orthonormé; au point M de coordonnées (x, y) on associe le nombre complexe $z = x + iy$, son affixe.

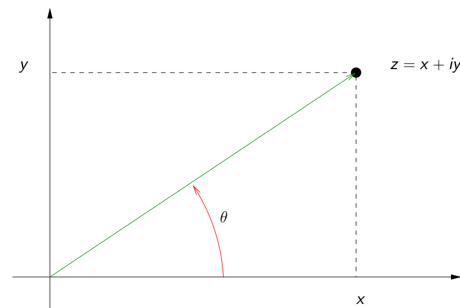


Figure: Plan complexe

Pour représenter un point du plan, on utilisera le type

```
type mycomplex = C of float * float
```

Opérations sur les complexes

On écrira en TM les fonctions et variables suivantes:

1. accesseur `realpart` qui permet de récupérer la partie réelle d'un complexe.
2. accesseur `imagpart` qui permet de récupérer la partie imaginaire d'un complexe.
3. variable `c_origin` contenant le point de coordonnées $(0, 0)$.
4. variable `p12` contenant le point $(1., 2.)$.
5. opérations
   `c_sum c1 c2`, `c_dif c1 c2`, `c_mul c1 c2`,
   `c_abs c`, `c_scal lambda c`, `c_exp c`
   qui permettent respectivement de calculer la valeur absolue d'un complexe, la somme, la différence, le produit de deux complexes, la multiplication d'un complexe par un scalaire (float), l'exponentielle complexe.

Transformations dans le plan complexe

Une transformation F du plan transforme tout point P en son image $P' = F(P)$. On peut décrire la transformation f par la fonction f qui appliquée à z l'affixe de P donne z' l'affixe de P'.

$$\begin{array}{rccc} f: & \mathbb{C} & \to & \mathbb{C} \\ & z & \mapsto & z' = f(z) \end{array}$$

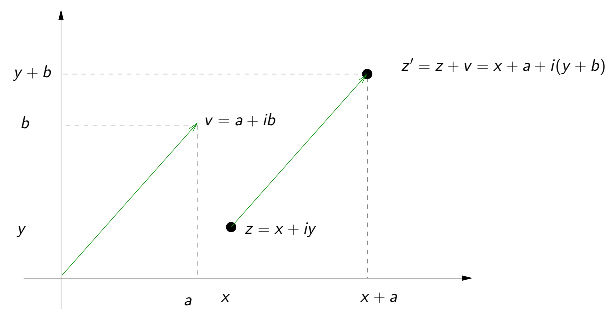


Figure: Translation de vecteur (a, b)

Transformations dans le plan complexe

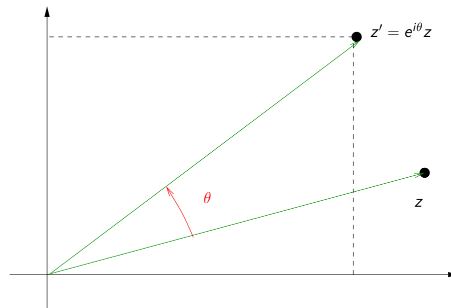


Figure: Rotation d'angle θ

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 programmation
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, une zone vide est représentée par la variable `nowhere` suivante:

```
# let nowhere = fun point -> false;;
val nowhere : 'a -> bool = <fun>
```

Appartenance d'un point à une zone

```
  # let c_origin = (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>
```

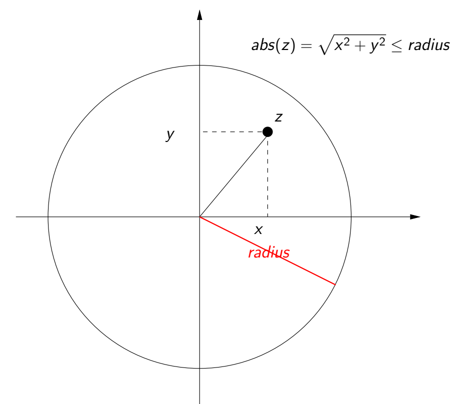
Pour forcer les types:

```
type zone = mycomplex -> bool

let point_in_zone (point : mycomplex) (zone : zone) =
   zone point;;
val point_in_zone : mycomplex -> 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 move_zone zone vector =
   fun p -> point_in_zone_p (c_dif p vector) zone
```

Visualisation avec visu-zones

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.
```