Comparaison théorique: notion de complexité

estimer théoriquement l'efficacité d'une fonction

- efficacité en temps (nombre d'opérations élémentaires)
- efficacité en espace (espace alloué)
- temps $>=$ espace

Notation $\mathcal{O}$:
La fonction f est dite en $\mathcal{O}(g)$ ssi

$$\exists k \in \mathbb{N}, \exists c > 0, \forall n > k, f(n) \leq cg(n)$$

Exemple: $\mathcal{O}(n \mapsto log_2(n))$
Par abus de notation: $\mathcal{O}(log_2(n))$

Exemples: $\mathcal{O}(log_2(n))$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, ...

Comparaison théorique: exemple

```
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | h :: t -> h :: append t l2
```

1. Combien d'appels récursifs de `append` ?
2. Combien de fois l'opérateur `::` est-il utilisé?

La récursion se faisant sur `l1`, la complexité dépend uniquement de la longueur de la liste `l1`.
Soit $a(n)$ le nombre d'appels récursifs pour `l1` de longueur n.

$$\begin{cases} a(0)=0 \\ a(n)=1+a(n-1) \end{cases}$$

se résout en $a(n) = n$.
le nombre d'appels à `::` est égal au nombre d'appels récursifs.
La fonction est en $\mathcal{O}(len(l_1))$.

Comparaison théorique: reverse quadratique

```
let rec reverse l =
  match l with
    [] -> []
  | h :: t -> append (reverse t) [h]
```

Soit $c(n)$ le nombre d'utilisations de `::` lors d'un appel à `reverse l` pour une liste `l` de longueur n.

$$\begin{cases} c(0)=0 \\ c(n)=a(n-1)+c(n-1) \end{cases}$$

se résout en $n(n-1)/2$.
La fonction est en $\mathcal{O}(len(l)^2)$.

Comparaison théorique: reverse linéaire

```
let rec rev_append l acc = (* O(len(l)) *)
  match l with
    [] -> acc
    | h :: t -> rev_append t (h :: acc)

let reverse l = rev_append l []
```

On peut montrer que le nombre d'appels récursifs est égal à n si n est la longueur de la liste `l`.
La fonction est en $\mathcal{O}(len(l))$.

Programmation modulaire

La programmation modulaire consiste à découper d'un programme en plusieurs modules.

Un module est un morceau de code définissant des types de données et un ensemble d'opérations sur ces types.

On peut voir ça comme l'implémentation d'un type abstrait.

Les modules ne sont pas propres à OCaML.

Comme en C, on aura une partie interface (`.mli`/`.h`) et une partie implémentation (`.ml`/`.c`).

Intérêt des modules

Les modules permettent de résoudre un certain nombre de problèmes qui se posent en programmation.

- Découpage de la difficulté: une petite entité est plus facile à comprendre, à déboguer (diviser pour régner)
- Réutilisabilité: définir des petites entités réutilisables
- Masquage de l'implémentation: pouvoir changer d'implémentation sans perturber les clients du module
- Contrôle de la visibilité des éléments encapsulés
- Sous-espaces de noms
- Généricité (polymorphisme)

Modules OCaML

Les modules `OCaML` interviennent dans le cadre d'un langage statiquement typé.

Ceci entraîne des contraintes supplémentaires sur la façon de compiler les modules.

La partie implémentation d'un module se fait dans un fichier `.ml`.

La partie visible est l'interface (ou signature) est dans un fichier `.mli` ou est inférée automatiquement.

Définition automatique d'un module

Un fichier `nom.ml` définit automatiquement un module de type `nom`.
Par exemple, le code suivant placé dans le fichier `point.ml` définit un module `Point`. Les noms de module commencent toujours par une majuscule.

```
type point = P of float * float
let p_x point = let P(x, _) = point in x
let p_y point = let P(x, _) = point in y
let p_origin = make_point 0.0 0.0
let p_i = make_point 0.0 (-1.)
let distance x y x' y' = sqrt (x *. x' +. y *. y')
let p_dist p1 p2 = dist (p_x p1) (p_y p1) (p_x p2) (p_y p2)
let p_abs p = p_dist p_origin p
```

Si on charge ce code directement dans la boucle d'interaction, le module n'existe pas puisqu'on n'a pas accès au nom du fichier.
On peut compiler en dehors de l'environnement interactif depuis un terminal:

```
ocamlc -c point.ml
```

Cela crée un fichier d'interface compilée `point.cmi` et le fichier objet `point.cmo`.
Dans un Makefile on peut utiliser la règle

```
%.cmo: %.ml
ocamlc -c $<
```

Le fichier `.cmo` peut être chargé dans la boucle d'interaction grâce à la requête `#load "point.cmo"`.

Accès à un élément d'un module

Par défaut, **tous** les éléments définis dans le module sont accessibles depuis l'extérieur en préfixant par `Nom.element`. Mais on verra par la suite qu'il est possible de cacher une partie de l'implémentation.
Pour ne pas avoir à préfixer (dans un autre module ou dans la boucle d'interaction), utiliser `open Point`.

Définition manuelle d'un module

```
module <Nom> = struct
  ...
end
```

```
module Point =
struct
  type point = P of float * float
  let p_x point = let P(x, _) = point in x
  let p_y point = let P(x, _) = point in y
  let p_origin = make_point 0.0 0.0
  let p_i = make_point 0.0 (-1.)
  let distance x y x' y' = sqrt (x *. x' +. y *. y')
  let p_dist p1 p2 = dist (p_x p1) (p_y p1) (p_x p2) (p_
  let p_abs p = p_dist p_origin p
end
```

À l'intérieur de `struct end`, on peut mettre `type`, `let`, `module`, `module type`, `exception`, `include`.
Dans ce cours nous ne verrons pas `exception`, `include`.

Tous les éléments du module sont compilés **séquentiellement**. Chaque nouvel élément peut utiliser les liaisons précédentes. Le résultat global est lié à l'identificateur `Nom`.

Si cette déclaration de module est dans un fichier, le module `Point` sera un sous-module du module créé pour le fichier.

Par exemple, si le module est déclaré dans un fichier `plan.ml`, les éléments du module `Point` sont accessibles par `Plan.Point.`.

Signatures

Chaque module a un type appelé signature qui joue le rôle d'interface entre le module et les clients potentiels du module.
Il existe une signature par défaut qui peut être inférée à partir de l'implémentation du module dans laquelle tous les éléments définis par le module et leur type sont visibles.
Pour la voir: `ocamlc -i point.ml`

```
type point = P of float * float
val make_point : float -> float -> point
val p_x : point -> float
val p_y : point -> float
val p_origin : point
val p_i : point
val dist : float -> float -> float -> float -> float
val p_dist : point -> point -> float
val p_abs : point -> float
```

Si le module a été déclaré avec `Module ... struct ... end`

```
module Point :
  sig
    type point = P of float * float
    val make_point : float -> float -> point
    val p_x : point -> float
    val p_y : point -> float
    val p_origin : point
    val p_i : point
    val dist : float -> float -> float -> float -> float
    val p_dist : point -> point -> float
    val p_abs : point -> float
    end
```

Définition de signature

Par défault la signature expose TOUS les types et TOUS les noms.
Si on ne souhaite pas exposer tout, on peut définir la signature d'un module à la main.

- Soit a priori comme spécification d'un module par encore implémenté,
- soit comme une restriction de la signature par défaut pour cacher une partie de l'implémentation.

Ici on cache l'implémentation du type `point` ainsi que certaines fonctions internes au module:

```
sig
type point
val make_point : float -> float -> point
val p_x : point -> float
val p_y : point -> float
val p_origin : point
val p_dist : point -> point -> float
val p_abs : point -> float
end
```

Une signature doit être liée à un nom de signature (par convention en majuscules) à l'aide du mot-clé `module type`

```
module type POINT =
sig
...
end
```

On peut forcer le type de signature du module:

```
module Point : POINT =
struct
...
end
```

Compilation

Pour compiler en dehors de l'environnement interactif, on peut utiliser `ocamlc` qui produit du byte-code pour la machine virtuelle `Zinc` d'OCaML ou `ocamlopt` qui produit du code natif (plus volumineux, plus rapide, pas portable).

```
man ocamlc
```

La commande `ocamlc` ressemble à `cc` ou `gcc`; elle prend en plus en compte le problème des signatures.

Si un fichier `f.mli` existe, le compilateur vérifie que l'implémentation de `f.ml` est conforme à la signature.

Compilation et édition de liens

Cas d'un seul fichier `f.ml`.

`ocamlc f.ml`

produit `a.out`

`ocamlc f.ml -o f.out`

produit `f.out`

Compilation séparée

Chaque fichier `fi.ml` est compilé séparément:

```
ocamlc -c fi.ml
```

produit le fichier objet `fi.cmo` (byte-code) et l'interface compilée `fi.cmi` (sauf si le fichier `fi.mli` existe).

```
ocamlc -o main f1.cmo f2.cmo f3.ml
```

Exemple avec zones

```
ocamlc -c mycomplex.ml
ocamlc -c zones.ml
ocamlfind ocamlc -c -package graphics images.ml
ocamlfind ocamlc -c -package graphics visu_zones.ml
ocamlfind ocamlc -c main.ml
ocamlfind ocamlc -linkpkg -package unix,graphics -o main my
```

Systèmes de compilation automatisés

`make`, `OMake`, `ocamlbuild`, `Oasis`, `Dune`

Bibliothèque sur les graphes

- orientés ou non orientés
- implémentation “orientée sommet”
- ensemble de noeuds représenté par un dictionnaire
- un graphe est un ensemble de noeuds possédant des voisins avec un nom et un booléen indiquant si orienté ou pas
- ensemble de noeuds représenté par un dictionnaire
- ensemble d’arcs représenté par un dictionnaire
- un dictionnaire est implémenté par une table de hachage (donc mutable)