Rappels de programmation

Programmation en C

`t` désigne un type générique, `v` une variable de type `t`, `ptr` un pointeur vers un objet de type `t`, `tab` un tableau d'éléments de type `t`, `x` une expression de type `t`, `n` un entier, `b` un booléen, `f` une fonction.

- Commentaires : après `//` ou entre `/*...*/`
- Entêtes : rajouter `#include <nom_entete.h>` en début de fichier. Entêtes usuelles : `assert.h`, `stdbool.h`, `stdio.h`, `stdlib.h`, `limits.h`.
- Types de base :
  - entiers : `int`, `unsigned int`, `uint_8/32/64` (dans `stdint.h`)
  - flottants : `double`
  - booléens : `bool`, valeurs `true` et `false`
  - caractères : `char`, valeurs `'a'`, `'b'`, ...
- Opérations élémentaires :
  - opérations arithmétiques : `+`, `-`, `*`
  - division entière ou flottante : `/`
  - modulo : `%`
  - opérateurs booléens : `&&`, `||`, `!`
  - comparaisons : `<`, `>`, `<=`, `>=`, `==`, `!=`
- Déclaration et initialisation : `t v = x;`
- Transtypage : `t v = (t) x;`
- Pointeurs :
  - pointeur d'un objet de type `t` : `t*`
  - déréférencement (pointeur → valeur) : `*ptr`
  - référencement (valeur → pointeur) : `&v`
  - allocation sur le tas : `malloc(`*nb octets*`)` renvoie un pointeur vers la zone
  - taille d'un type ou objet : `sizeof(t)`, `sizeof(v)`
  - libération d'une zone mémoire : `free(ptr)`
- Tableaux :
  - initialisation sur la pile : `t tab[N] = 0;` ou `t tab[N] = x0, x1, ...;`, `N` doit être une constante entière littérale
  - allocation sur le tas : `t* tab = (t*) malloc(n * sizeof(t));`
  - élément d'indice i : `tab[i]`
  - chaînes de caractères : type `char*`, valeurs `"hello"`,`"world"`, ..., ce sont des tableaux de `char` dont le dernier élément est `'\0'`
- Structures de contrôle : délimitées avec `{...}`, les variables déclarées sont locales
  - test : `if (b) ... else ...`
  - boucle : `for (`*init*`;`*fin*`;`*incr*`){...}` ou `while (b) ...`
  - sortie de boucle : `break`
  - passage à l'itération suivante : `continue`
  - fonction : `t f(t1 v1, ...)...`, `int main(void)`, ou avec des arguments `int main(int argc, char* argv[])`
- Types structurés :
  - déclaration : `struct T t1 champ1; ...;`
  - renommage : `typedef struct T t;`
  - initialisation sur la pile : `t v = {.champ1 = x1, ...};`
  - accès à un champ : `v.champ1` ou `ptr->champ1`
- Affichage et lecture :
  - affichage d'une chaîne formatée : `printf(chaîne, v1, v2, ...)`
  - lecture d'une entrée : `scanf(%xxx, ptr)`
  - formattage : `%d` (entier), `%lf` (double), `%s` (chaîne de caractères), `%p` (pointeur)
  - caractères spéciaux : `\n` (retour à la ligne), `\t` (tabulation)
- Compilation et exécution :
  - syntaxe générale : `gcc` *options* `source.c`
  - options :
    - nom d'exécutable : `-o nom_exec`
    - avertissements : `-Wall`, `-Wextra`
    - alerte mémoire : `-fsanitize=address`
  - exécution : `./nom_exec`

Programmation en OCaml

`t` désigne un type générique, `v` une variable de type `t`, `tab` un tableau d'éléments de type `t`, `lst` une liste d'éléments de type `t`, `x` une expression de type `t`, `n` un entier, `b` un booléen, `f` une fonction, `E` une exception.

- Commentaires : entre `(*...*)`
- Importation de bibliothèque : `#load` *biblio*
- Ouverture de module : `open` *Module*
- Types de base :
  - *void* : `unit`, valeur `()`
  - entiers : `int`
  - flottants : `float`
  - booléens : `bool`, valeurs `true` et `false`
  - caractères : `char`, valeurs `'a'`, `'b'`, ...
  - chaînes de caractères : `string`, valeurs `"hello"`, ...
- Opérations :
  - entiers : `+`, `-`, `*`, `/`
  - flottants : `+.`, `-.`, `*.`, `/.`, `**`
  - modulo : `mod`
  - opérateurs booléens : `&&`, `||`, `not`
  - comparaisons : `<`, `>`, `<=`, `>=`, `=`, `<>`
- Déclaration et initialisation : `let v = x` (`v` globale),
  `let v = x in e` (`v` locale à `e`)
- Tableaux :
  - type : `t array`
  - création : `Array.make n x`, `[|x0; x1; ...|]`
  - taille : `Array.length tab` (en $\mathcal{O}(1)$)
  - copie superficielle : `Array.copy tab`
  - initialisation : `Array.init n f` calcule `[|f 0; f 1; ...|]`
  - élément d'indice i : `tab.(i)`
  - modification : `tab.(i) <- x`
  - parcours : `Array.iter f tab` exécute `f tab.(0); f tab.(1); ...`
  - image par une fonction : `Array.map f tab` renvoie `[|f tab.(0); f tab.(1); ...|]`
- Listes :
  - type : `t list`
  - création : `[]`, `x :: lst`, `[x0; x1; ...]`
  - taille : `List.length lst` (en $\mathcal{O}(n)$)
  - tête et queue : `List.hd lst`, `List.tl lst`
  - parcours et image `List.iter` et `List.map`
- Uplets :
  - type : `t1 * t2 * ...`
  - création : `(x1, x2, ...)`
  - accès à la ième composante : `let (x1, x2, ...) = v in xi`
  - couples uniquement : `fst v`, `snd v`
- Types enregistrements :
  - définition (avec champs mutables) : `type t = {champ1 : t1; mutable champ2 : t2; ...}`
  - création : `{champ1 = x1; ...}`
  - accès à un champ : `v.champ1`
  - modification : `v.champ1 <- x`
- Références :
  - création : `let r = ref x`
  - accès à la valeur : `!r`
  - modification : `r := x`
- Types construits :
  - définition : `type t = Cons1 | Cons2 of t2 | Cons3 of t3 * t4 ...`
  - création : `Cons1`, `Cons2 x`, `Cons3 (x3, x4)`
  - type option (existe par défaut) : défini par `type 'a option = None | Some of 'a`
- Structures de contrôle : les variables déclarées sont locales
  - test : `if b then (...) else begin...end`, les `( )` et `begin end` sont interchangeables
  - filtrage : `match v with | motif1 -> ... | motif2 -> ...` un motif peut contenir types construits, enregistrements, variables distinctes ou constantes
  - boucle Pour : `for i = n1 to n2 do ... done`, `n1` et `n2` sont atteintes, on remplace `do` par `downto` pour une boucle décroissante
  - boucle Tant que : `while b do ... done`
  - fonction : `let f v1 v2 ... = ...`
  - fonction récursive : `let rec f v = ...`
- Exceptions :
  - définition : `exception E1`, `exception E2 of t`
  - levée : `raise E1`, `raise E2(v)`
  - rattrapage : `try ... with | E1 -> ... | E2(v) -> ...`
  - message d'erreur : `failwith` *erreur*
- Affichage :
  - types de base : `print_int`, `print_float`, `print_string`, `print_newline ()`
  - chaîne formatée : `Printf.printf`