CHAPITRE 3 – BASES DU CAML

PRÉSENTATION DE CAML

- Caml = Langage de programmation
- OCaml = implémentation du langage Caml
  - Développé par Inria (Institut National de Recherche en Informatique et en Automatique)
  - Projet open-source
  - Anciennement : CamlLight (plus maintenu)

DESCRIPTION

- Caml *(Categorical Abstract Machine Language)* est un langage fonctionnel
  - Il a été inventé en 1985
  - Il est impur : il ne possède pas « que » des éléments fonctionnels (un peu comme Java n'est pas « que » objet)
  - Il a un typage statique : le typage est effectué avant l'évaluation (et non pas pendant l'évaluation)
  - Il a un typage fort : le type choisi défini exactement les données
  - Il est à inférence de type (avec reconnaissance de motifs) : c'est Caml qui déduit le type des expressions

COMMENTAIRES

- Un commentaire en Caml s'écrit entre « (* » et « *) »

3.1 – TYPES ET CONSTANTES

CONSTANTES ET TYPE INT

- Pour créer une constante, on utilise le mot-clé « let »
  - Exemple : « let c = 2;; » ➔ Caml crée une constante nommée « c » de valeur « 2 »
- Type entier : « int » représente le type entier (positifs ou négatifs)
  - Tout nombre écrit sans virgule est un entier
  - Exemple : « let c = 2;; » ➔ Caml répond : « val c : int = 2 » ; Caml a déduit le type de « c »
  - Opérations : « + », « - », « * », « / » (division entière), « mod » (modulo)

TYPE FLOAT

- Type réel : « float » représente les réels
  - Tout nombre écrit avec un « . » est un réel
  - Exemple : let « pi = 3.1415;; » ➔ Caml répond : « val pi : float = 3.1415 »
  - Opérations : « +. », « -. », « *. », « /. »
  - Il existe des fonctions prédéfinies : « sqrt », « cos », « sin », etc.

TYPES CHAR ET STRING

- Type caractère : « char »
  - Exemple : « let c = 'F';; »
- Type chaîne de caractères : « string »
  - Exemple : « let s = "Hello";; »
  - Opérations : « ^ » concatène des chaînes, « string_length » (en CamlLight) ou « length » (en OCaml), « s.[0] » retourne le premier caractère de la chaîne « s »
- Conversions de type
  - Il existe des fonctions de conversion : int_to_float, float_to_int, int_of_char, string_of_int, string_of_char, etc.

TYPES BOOL ET UNIT

- Type booléen : « bool »
  - Valeurs : « true » et « false »
  - Opérateurs : « not » représente l'opérateur « NOT », « & » et « && » représentent l'opérateur AND, « || » et « or » représentent l'opérateur OR ; ils ont une évaluation paresseuse
- Type unitaire : « unit »
  - Exemple : « let u = ();; »
  - « () » est la seule valeur de ce type ; cela correspond au type « void » en C

3.1.1 – TYPES CONSTRUITS

TYPE ÉNUMÉRÉ

- Type énuméré simple (équivalent du « enum » en C) :
  - Création : « type humain = Homme | Femme;; »
  - Utilisation : « let alice = Femme;; » ➔ « val alice : personne = Femme »
- Type énuméré paramétré
  - Création avec un paramètre : « type animal = Chien of string | Chat of string | Poisson;; »
  - Utilisation : « let a = Poisson;; » ou « let b = Chien("Medor");; »

TYPE ENREGISTREMENT

- Type enregistrement (équivalent du « struct » en C) :
  - Création : « type personne = { prenom : string; nom : string; age : int; };;
  - Utilisation : « let alice = { prenom="Alice"; nom="Dupont"; };; »
  - Puis : « alice.prenom;; » ➔ « - : string = "Alice" »

TYPE TUPLE

- Exemple de tuple :
  - Création : « let coord = (3.1, 2.9);; » ➔ « val coord : float * float = (3.1, 2.9) »
  - Utilisation : il faudra utiliser de la reconnaissance de motifs pour extraire les coordonnées, ou bien les fonctions « fst » et « snd » (voir plus loin)
- Une fonction Caml ne prend qu'un seul paramètre et ne renvoie qu'un retour
  - Mais ce paramètre et ce retour peuvent être des tuples

3.1.2 – TYPE LISTE

TYPE LISTE

- Les listes sont très utilisées en Caml
- Création de listes :
  - « [] » désigne la liste vide
  - « [1; 2; 3] » désigne la liste contenant les trois entiers 1, 2 et 3
- Opérations sur les listes :
  - « :: » ajoute un élement en tête de liste
  - « @ » concatène (= met bout à bout) deux listes
- Une liste ne contient que des éléments du même type
- Caml infère le type de la liste
  - « [1; 2; 3];; » ➔ « - : int list = [1; 2; 3] »
  - « [‘a’; ‘b’; ‘c’];; » ➔ « - : char list = [‘a’; ‘b’; ‘c’] »

LISTE DE LISTES

- On peut créer des listes de listes (ou des listes de listes de listes, etc.)
- Exemple :
  - « let l = [[11;10;12];[18;20];[11;9;19]];; » ➔ « val l : int list list = […] »

3.2 – DÉFINITION DE FONCTIONS ET INFÉRENCE DE TYPES

EXPRESSION « IF…THEN…ELSE »

- Caml dispose d'une expression conditionnelle du type « if…then…else »
  - Exemples : « if x=0 then true else false » ou « if age>=18 then "majeur" else "mineur" »
  - Remarque : le « else » est obligatoire

CRÉATION D'UNE FONCTION SIMPLE

- Syntaxe 1 :

  let f x = x+1;;

- Syntaxe 2 :

  let f = function x->x+1;;

- Syntaxe 3 :

  let f = fun x->x+1;;

- Appel de la fonction f :

  « f(4);; » ➔ « - : int = 5 »

INFÉRENCE SIMPLE

- Caml détecte le type d'une fonction, et le note « -> »
  - Exemple : « let f x = x+1;; » ➔ « fun f : int -> int »

OPÉRATEUR « FAILWITH »

- Lorsqu'un paramètre incorrect est passé à une fonction, on peut utiliser « failwidth » en indiquant un message d'erreur
- Exemple :

```
« let divisionEntiere (a,b) = if b=0
        then failwith "Division par zéro"
        else a/b;; »
```

POLYMORPHISME

- Caml gère le polymorphisme, c'est-à-dire qu'il peut gérer des paramètres dont le type est quelconque
- Exemple de la fonction identité :
  - « let identite x = x;; » ➔ « fun identite : 'a -> 'a »
  - « 'a » signifie « n'importe quel type »
- Exemple d'une fonction qui renvoie le premier élément d'un couple :
  - « let premier (x,y) = x;; » ➔ « fun premier : ('a * 'b) -> 'a »
  - On remarque que « x » et « y » n'ont pas forcément le même type, donc Caml infère « 'a » pour « x » et pour le retour, et « 'b » pour « y »
  - Remarque : cette fonction est prédéfinie dans Caml et se nomme « fst » ; il existe aussi la fonction « snd » qui retourne le deuxième élément d'un couple

PARAMÈTRES MULTIPLES

- Pour Caml, chaque fonction a un seul paramètre
- Pour créer une fonction qui a deux paramètres (ou plus) :
  - Soit on utilise un couple (ou tuple) : exemple : « let somme (a,b) = a+b;; » ➔ « fun somme : int * int -> int »
  - Soit on crée une fonction qui prend en paramètre le premier paramètre, et qui retourne une fonction qui prend en paramètre le deuxième paramètre : « let somme = function a -> function b -> a+b;; » ➔ « fun somme : int -> int -> int »
  - Soit on crée une fonction qui prend en paramètre les deux paramètres (ou plus) : « let somme a b = a+b;; » ➔ dans ce cas, Caml fait comme pour la deuxième solution
- On remarque que l'inférence de type identifie bien la fonction « somme » comme prenant des entiers en paramètre

REMARQUES SUR LA PRIORITÉ

- Priorité des opérateurs de type :
  - « → » a une priorité inférieure à « * », donc « (int * int) → int » est synonyme de « int * int → int »
  - « → » est associatif à droite, donc « 'a → ('b → 'c) » est synonyme de « 'a → 'b → 'c »

CRÉATION D'UNE FONCTION RÉCURSIVE

- Pour créer une fonction récursive, on utilise le mot clé « rec »
  - Exemple :

```
« let rec fact n =
    if n=0 then 1
    else n*fact(n-1);; »
```

DÉCLARATIONS IMBRIQUÉES (1/3)

- La syntaxe « let x=expr1 in expr2 » est une déclaration imbriquée
  - Caml va remplacer les occurrences de « x » dans « expr2 » par « expr1 »
  - Cette syntaxe permet de factoriser le code (et donc de le rendre plus lisible)
  - Cette syntaxe permet aussi de « cacher » un paramètre dans une fonction, en le transformant en une constante
- Exemple de factorisation :
  - « let quadruple x =
    let double x = x+x
    in double (double x);; »

DÉCLARATIONS IMBRIQUÉES (2/3)

- Exemple de paramètre caché :
  - « let multiplication list valeur =
    let op x = x*valeur in
    map op list;; »
  - On remarque que la fonction « op » ne prend en paramètre que « x », qui est multiplié par « valeur »
  - Dans la fonction « op », « valeur » est une constante, mais dans la fonction « multiplication », « valeur » est un paramètre que l'on peut choisir
  - On n'aurait pas pu passer à « map » une fonction qui prenait deux arguments « x » et « valeur »

DÉCLARATIONS IMBRIQUÉES (3/3)

- On peut définir plusieurs constantes ou fonctions en parallèle avec « let … and … in … »
  - Exemple : « let a =15 and b=12 in a*b;; »

3.3 – RECONNAISSANCE DE MOTIFS

RECONNAISSANCE DE MOTIFS (1/2)

- Caml permet de faire de la reconnaissance de motifs grâce à « match … with »
- La reconnaissance de motifs fonctionne sur les types de base, les tuples, les listes, etc.
- Exemple avec un entier :

```
« let f x = match x with 0 -> 1
        | 1 -> 2
        | x -> x+1;; »
```

- Exemple avec un couple :

```
« let fst couple = match couple with (x,y) -> x;; »
```

RECONNAISSANCE DE MOTIFS (2/2)

- Exemple avec une liste :

```
« let suppressionFin l = match l with [] -> failwith "liste vide"
                | [x] -> []
                | _::xs -> suppressionFin xs;; »
```

- Remarque : « _ » désigne « tous les autres cas »
- Remarque : les motifs doivent être décrits de manière exhaustive

3.4 – AFFICHAGES

PROGRAMMATION IMPÉRATIVE EN CAML

- Caml supporte des instructions séquentielles, notamment pour l'affichage
  - Pour exécuter des instructions séquentielles, on utilise « ; » après chaque instruction
  - Pour afficher une chaîne de caractères « chaine », on utilise « print_string(chaine) »
  - Pour passer à la ligne, on utilise « print_newline() »