Mémo CAML*

Info-sup S1

Nathalie "Junior" Bouquet

2023

Les éléments du langage

Caractères spéciaux, séparateurs et commentaires

Certains caractères et combinaisons de caractères ont une signification spéciale pour le langage. Ce sont les opérateurs, les séparateurs, les espaces... Nous les verrons au fur et à mesure de leur utilisation.

Les commentaires sont délimités par `(*` et `*)`.

Les règles de constructions des identifiants (ou identificateurs)

- Les identifiants sont des suites de lettres `a..z A..Z`, chiffres `0..9` et `_` (underscore).
- CAML est sensible à la casse (pour tous les "mots").
- Un identifiant commence par :
  - une lettre minuscule [1] ;
  - le caractère `'_'` à condition que celui-ci soit suivi d'au moins un caractère.
- Les identifiants doivent être différents des mots-clés du langage.
- Les identifiants doivent être **explicites** !

Les types de base et leurs opérateurs

Les entiers : `int`

Intervalle	64-bits platforms : $[-2^{62}, 2^{62}-1]$ (32-bits : $[-2^{30}, 2^{30}-1]$)
Valeurs :	`1 45 -69`
Opérateurs :	`+ - * /` (division entière [2]) `mod = <> < > <= >=`
Quelques fonctions :	`succ pred abs`
Valeurs particulières :	`max_int min_int`

Les nombres flottants : `float`

Codés sur 64 bits :	mantisse sur 53 bits et exposant $\in [-1022, 1023]$
Valeurs :	`12.5 -0.5 3. -3. 3e2 5.75e-2`
Opérateurs :	`+. -. *. /. = <> < > <= >=`
Quelques fonctions :	`sqrt ceil floor abs_float log cos` ...
Valeurs particulières :	`max_float min_float infinity neg_infinity nan`

Les booléens : `bool`

Valeurs :	`true false`
Opérateurs :	`not || && = <>`

Les caractères : `char`

Code ASCII :	$[0, 127]$
ISO 8859-1 standard :	$[128, 255]$
Valeurs :	`'a' 'A' '$' '9' '\065' '\n'` [3]
Opérateurs :	`= <> < > <= >=`
Quelques fonctions :	`Char.code Char.chr Char.escaped`

*Tous les exemples CAML donnés ici ont été évalués (interprétés) avec CAML 4.07.

1. Sauf pour les constructeurs de types et les modules (hors programme).
2. Uniquement dans $\mathbb{N}$ pour l'instant.
3. `'\065'` : le caractère de code ASCII 65. `'\n'` : retour à la ligne.

Les chaînes de caractères : `string`

Séquences finies de caractères.

Longueur :	$[0, 2^{57} - 9]$ (32-bit : $2^{24} - 5$)
Valeurs :	`"a string"` `"a"` `""` (chaîne vide)
Opérateurs :	`^` (concaténation) `=` `<>` `<` `>` `<=` `>=`
Accès à un caractère :	`s.[i]` est le caractère (de type `char`) au rang `i` ($0 \leq i <$longueur(s)) de `s`
Quelques fonctions :	`String.length` `String.sub`

Quelques fonctions de conversion

`float_of_int : int -> float`	entier → flottant
`int_of_float : float -> int`	flottant → entier (tronqué)
`int_of_char : char -> int` = `Char.code`	caractère → code ASCII
`char_of_int : int -> char` = `Char.chr`	code ASCII → caractère
`Char.escaped : char -> string`	caractère → chaîne
`string_of_int : int -> string`	entier → chaîne
`int_of_string : string -> int`	chaîne → entier
`string_of_float : float -> string`	flottant → chaîne
`float_of_string : string -> float`	chaîne → flottant

1 Les phrases : expressions et définitions

Pour l'instant nous connaissons deux types de phrases : les **expressions** et les **définitions**. Dans le système interactif, une phrase est toujours terminée par `;;`

1.1 Les expressions

Une expression peut être :

atomique :	une valeur simple	`15`	`x`
structurée :	une opération	`a + 15`	`"Hello " ^ "world"`
	parenthésée	`(3*a)`	`(true || false)`
	une application de fonction	`cos x`	`float_of_int 15`

```
# (666 * 42 = 27972) && ("Hello" < "World");;
 - : bool = true
```

Une expression peut aussi contenir une(des) définition(s) locale(s) (voir ci-dessous), des alternatives, du "pattern matching" (étudiés plus tard)...

1.2 Les définitions

Définitions globales simples

`let` *ident* = *expression*

Définition : le nom *ident* est **lié** à la valeur de l'*expression*.

```
# let a = 1 + 2 ;;
 val a : int = 3
```

Une fois défini, un nom est toujours lié à la même valeur (ne peut être modifié), mais il peut être "masqué" par une nouvelle définition utilisant le même nom...

Définitions globales multiples

`let` $ident_1$ = $expression_1$
`and` $ident_2$ = $expression_2$
...
`and` $ident_n$ = $expression_n$

```
# let one = 1 and two = 2. and three = '3' ;;
 val one : int = 1
 val two : float = 2.
 val three : char = '3'
```

Definitions locales

Définition locale simple : `let` *ident* `=` $expression_1$ `in` $expression_2$

```
# let a = 1 + 2 in a * 3 ;;
 - : int = 9
```

Définitions locales multiples :

`let` $ident_1 = expr_1$ `and` ... `and` $ident_i = expr_i$
`in let` $ident_{i+1} = expr_{i+1}$ `and` ... `and` $ident_j = expr_j$
...
`in` $expression_n$

```
# let a = 1 and b = 3 in
    let x = a + b and y = a - b in
        x * y ;;
 - : int = -8
```

Une expression contenant une définition locale est une expression

2 Les fonctions

2.1 Fonctions à un seul paramètre

Définir une fonction : `let` $f\ x$ = *expression*
Type : `f :` $type_x$ `->` $type_{res}$

```
# let succ x = x + 1 ;;
 val succ : int -> int = <fun>
```

Application de la fonction f à la valeur x : $f\ x$

```
# succ 3 ;;
 - : int = 4
```

L'application de fonction est prioritaire sur tout autre opérateur.

$$f\ x + y \quad \equiv \quad (f\ x) + y \qquad\qquad f\ x + g\ y \quad \equiv \quad (f\ x) + (g\ y)$$

```
# succ 3*2 ;;
 - : int = 8
```

```
# succ (3 * 2) ;;
 - : int = 7
```

Si le paramètre n'est pas une expression atomique (simple), il faut donc le parenthéser.

```
# succ -1 ;;
 Characters 0-4: [4] Error: This expression has type int -> int
                            but an expression was expected of type int
# succ (-1) ;;
 - : int = 0
```

2.2 Fonctions à plusieurs paramètres

Définition : `let` $f\ x\ y$ = *expression*
Type : `f :` $type_x$ `->` $type_y$ `->` $type_{res}$

```
# let average a b = float_of_int(a + b) /. 2.;;
 val average : int -> int -> float = <fun>
```

Application à x et y : $f\ x\ y \quad \equiv \quad (f\ x)\ y$

```
# average 2 3 ;;  (* same as (average 2) 3 *)
 - : float = 2.5
# average ( 2  3) ;;
 Error: This expression has type int
        This is not a function; it cannot be applied.
```

L'application de fonction est "associative à gauche"

Donc : $f\ (g\ x) \quad \neq \quad f\ g\ x$

```
# succ succ 2 ;;
 Error: This function has type int -> int
        It is applied to too many arguments; maybe you forgot a ';'. [5]
# succ (succ 2) ;;
 - : int = 4
```

Attention : `let` *f (x, y)* = *expression* est une fonction à 1 seul paramètre (le couple *(x, y)*) de type $type_x$ `*` $type_y$ `->` $type_{res}$

4. L'indication des caractères concernés par l'erreur sera omise dans les exemples suivants.
5. Ceci (le ';') sera expliqué plus tard...

3 Analyses par cas

3.1 L'alternative

`if` *condition* `then` *expression*$_1$ `else` *expression*$_2$

— *condition* est une expression booléenne ;
— *expression*$_1$ et *expression*$_2$ sont du même type ;
— la valeur de l'alternative est *expression*$_1$ si *condition* a la valeur `true` et *expression*$_2$ si *condition* a la valeur `false`.

L'alternative est une expression.

```
# if 2 > 1 then
      "higher"
  else
      "lower" ;;
 - : string = "higher"
```

```
# let absolute_value x =
    if x >= 0 then
        x
    else
        -x ;;
val absolute_value : int -> int = <fun>
```