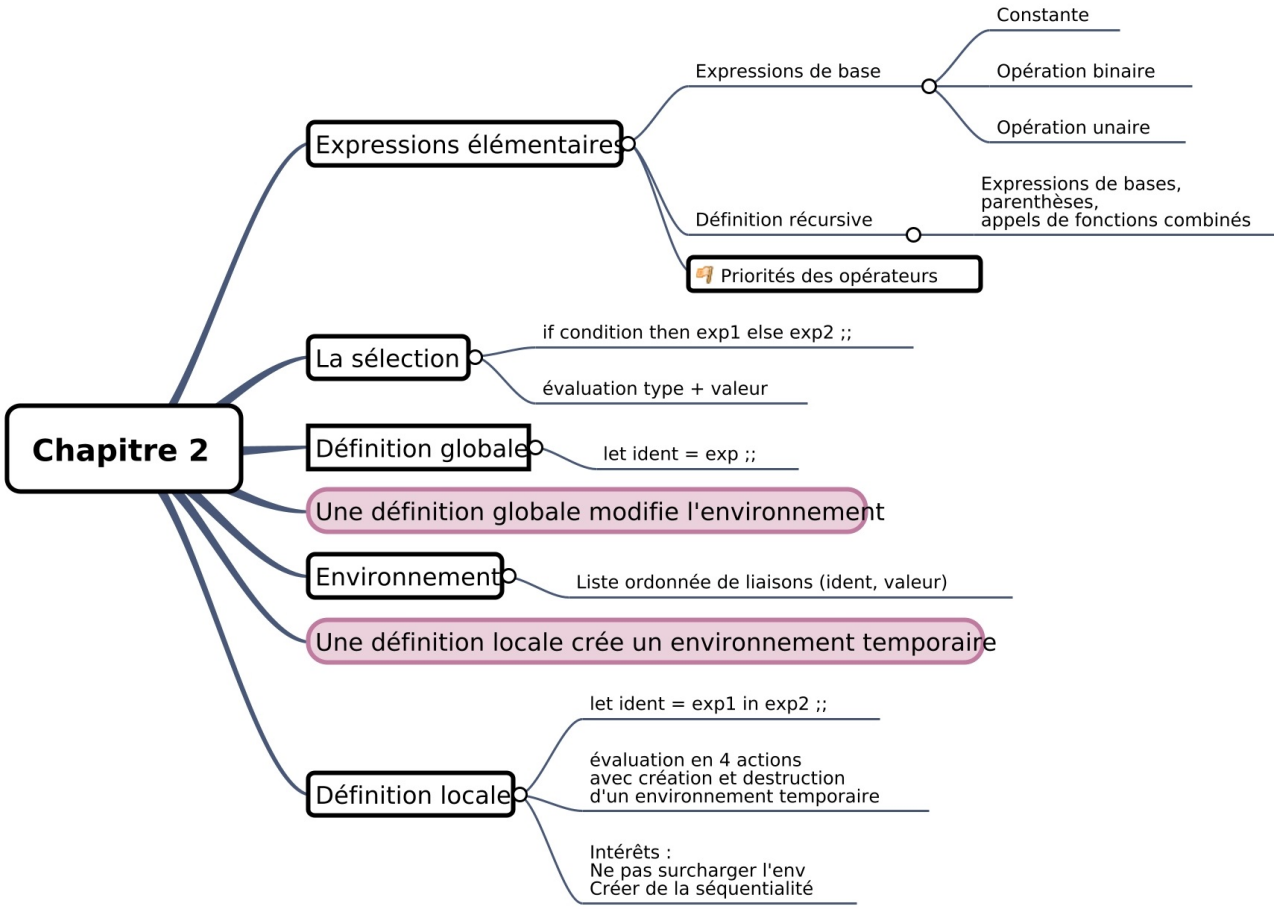
Institut National Universitaire Champollion

Chapitre 2 : Expressions élémentaires, sélections, définitions et environnement

En bref



Constructeur ou opérateur
appel de fonction
opérateurs unaires − et −.
les "produits" ∗, ∗. , /, /., *mod*
les "sommes" +, +., −, −.
les comparaisons : >, <, =, <>, <=, >= e
not
&
or

1. Expressions élémentaires

1.1. Expressions simples

Pour l'instant une expression est

- **Soit une constante**, par exemple `72.63 ;;`
  Le type est automatiquement reconnu par Caml à l'issue de l'analyse lexicale, la constante est évaluée littéralement (sa valeur est elle-même !).
- **Soit le résultat d'une opération binaire**, par exemple `89.65 -. 72.63 ;;`
  Caml évalue d'abord les types `(t1, t2)` des deux opérandes. S'ils sont compatibles avec l'opérateur, c'est à dire si le type de l'opérateur est de la forme `t1 * t2 -> t`, alors l'expression est de type `t`. S'ils sont incompatibles, le système affiche un message d'erreur.
  Pour le calcul de la valeur, on effectue simplement l'opération.
- **Soit le résultat d'une opération unaire**, par exemples `-. 72.63 ;;`, `not true ;;`
  Caml évalue le type `t1` de l'opérande. Si l'opérateur est de type `t1 -> t` alors l'expression est de type `t`, sinon le système affiche un message d'erreur.
  Le calcul de la valeur de l'expression s'effectue simplement par application de l'opérateur.

1.2. Expressions élémentaires, définition

Les expressions élémentaires sont définies récursivement à partir de ces trois expressions de base :

Définition 1

Une **expression élémentaire** est une expression syntaxiquement correcte ne faisant intervenir que des constantes, des parenthèses, des opérations binaires ou unaires et des appels de fonctions.

Exemple :

```
# (3.0 +. 2.2) *. (sqrt 3.) ;;
```

1.3. Priorité dans l'évaluation des expressions élémentaires

En général, l'analyse lexicale fait apparaître plusieurs opérations, mais dans quel ordre faut-il les traiter ?
Les règles de priorité sont les suivantes :

- Le contenu des parenthèses les plus internes est évalué prioritairement.
- Les opérateurs sont ici listés du plus prioritaire au moins prioritaire :

Constructeur ou opérateur
appel de fonction
opérateurs unaires − et −.
les "produits" ∗, ∗. , /, /., *mod*
les "sommes" +, +., −, −.
les comparaisons : >, <, =, <>, <=, >= e
not
&
or

En cas d'égalité, l'expression est évaluée de gauche à droite, pour la plupart des opérateurs, sauf pour les appels de fonctions et nous verrons plus tard d'autres exceptions à cette règle.

Exemple :

```
4    *    5  +    6 mod 2
(eval 1)          (eval 2)
           (eval 3)
```

Il est conseillé d'utiliser les parenthèses de manière systématique pour lever toute ambiguïté !

Exercice

Pour les expressions ci-dessous, dans quel ordre vont être évaluées les différentes opérations ? Vérifier ensuite votre réponse en les évaluant avec Caml Light.

1+2 = 2+1 & 4>5 ;;
1+2 = 2+1 or 4>5 ;;
1+2 = 2+1 > 4 > 5 ;;
1+2 = 2+1 > (4>5) ;;
true or false = (1=1) & (4<5) ;;
(1+2 = 2+1) & 4>5 ;;
true or false = (1=1) & (4<5) ;;
4 + 1 < 6 & ('a'<'h' or "debut" = "fin") ;;

Exercice

Construire et évaluer des expressions booléennes permettant de prouver que
- les comparaisons sont prioritaires sur not
- not est prioritaire sur & et or
- & est prioritaire sur or

2. La sélection

2.1. Syntaxe

Pour élaborer des expressions plus complexes, commençons par introduire la **sélection**, permettant de choisir la valeur d'une expression en fonction de la valeur booléenne d'une condition.

Définition 2

La syntaxe générale de la sélection est

```
if condition then exp1 else exp2
```

où *condition* est une expression de type `bool` et *exp1* et *exp2*, deux expressions de même type `t`.

Attention ! À ne pas confondre avec le branchement conditionnel "si cette condition est vrai, fait si, sinon fait ça", nous sommes en programmation fonctionnelle !

2.2. Evaluation

- **Typage**
  L'évaluation du type d'une sélection demande en fait 3 évaluations de types : celle de la condition et celles des deux expressions.

Si la condition est de type `bool` et que les deux expressions sont du même type `t`, alors la sélection est de type `t`. Dans tous les autres cas, il y a une erreur de typage.

- **Valeur**
  On calcule la valeur de la *condition* ($true$ ou $false$) et la valeur de *exp1* si la *condition* est vraie, la valeur de *exp2* sinon. Cette dernière valeur calculée est la valeur de la sélection.

Exemple :
`if 2+3 = 5 then "oui" else "non" ;;` est une expression de type `string` de valeur "oui".

Exercice

Les expressions suivantes sont-elles bien typées ? Si oui, quelles sont leur valeur ? Vérifier avec Caml.

```
if 1 < 3 then "Hello world" ;;
if "A" < "B" then 1 else 3 ;;
if 2.0 < 3.5 then 2 else 4. ;;
```

Exercice

L'usage de la sélection dans les cas suivants est-il pertinent ? Nous supposons que A,B sont des expressions de type `bool`.

```
if A then true else false ;;
if A then true else B ;;
if A then false else B ;;
if A then B else true ;;
if A then B else false ;;
```

3. Définitions et environnements

Un langage fonctionnel prend tout son intérêt avec les notions d'environnements et de définition. La valeur d'une expression dépend de son environnement :

```
# let x = 1 ;  (* Ceci est une définition *)
x  : int = 1

# x + 3 ;;
-  : int = 4

# let x = 2 ;; (* Ceci est une autre définition *)
x  : int = 2

 #x + 3 ;;
 -  : int = 5
```

À deux reprises, on demande l'évaluation de l'expression `(x+3)` mais les valeurs des répliques sont différentes. **L'environnement** a changé du fait de nouvelles **définitions globales**.

3.1. Environnement

Définition 3

Nous appelons **environnement** une liste de liaisons ordonnée chronologiquement. On la note entre crochets :

$$Env = [(id_1, v_1), \ldots, (id_n, v_n)]$$

Par convention, la liaison la plus récente est à gauche, la plus ancienne à droite. Les liaisons sont composées d'une identificateur (id_i) et de sa valeur (v_i).

Dès le début d'une session Caml, il existe un environnement noté `Env0` contenant les liaisons prédéfinies par Caml (que nous n'avons pas à connaître). Par la suite, toute nouvelle définition globale ajoute une nouvelle liaison en tête de liste.

3.2. Définition globale

Définition 4

Nous appelons **définition globale** en Caml, l'association entre un identificateur et une expression, obtenue par la requête :

```
let ident = exp ;;
```

Cette association résulte en une `liaison` entre cet identificateur et la valeur de cette expression *au moment* de la définition notée `(ident, val)`.

Attention ! Une définition globale n'est pas une expression !
En fait en Caml, nous ne pourrons faire que deux choses :

1. Faire évoluer l'environnement courant par des définitions globales
   La réplique de Caml commence alors par le nom du nouvel identificateur lié
2. Demander l'évaluation d'une expression dans l'environnement courant
   La réplique de Caml commence alors par un tiret -

3.3. Une définition globale modifie l'environnement

Exemple : La définition `let x = 2 ;;` crée une liaison $(x, 2)$.
Si, à un instant donné, on est dans un environnement

$$Env = [(id_1, v_1), \ldots, (id_n, v_n)]$$

et que l'on définit

```
let x = 2 ;;
```

Cette définition va modifier l'environnement en

$$Env2 = [(x, 2), (id_1, v_1), \ldots, (id_n, v_n)]$$

que l'on note aussi

$$Env2 = [(x, 2) >< Env]$$

si l'on ne veut pas réécrire le contenu de Env (ou qu'on l'ignore !). On change la notation Env en $Env2$ pour faciliter la lecture et montrer l'évolution de l'environnement, mais c'est évidemment le même environnement qui est actualisé.

Important

Si on appelle Env l'environnement actif, une définition globale `let ident = exp ;;` provoque les actions suivantes :

1. L'expression `exp` est évaluée dans l'environnement Env, soit `v` sa valeur.
2. La liaison $(ident, v)$ est ajoutée en tête de liste, on passe donc dans l'environnement $Env2 = [(ident, v) >< Env]$

3.4. Évaluation d'une expression contenant des identificateurs

Important

Si l'analyse lexicale d'une requête fait apparaître des identificateurs, pour chacun d'eux on cherche à partir de la tête de l'environnement la première liaison (donc la plus récente, la dernière ajoutée) qui lie cet identificateur à une valeur.
Si une telle liaison est trouvée, l'identificateur est évalué par la valeur associée, sinon le système provoque une erreur.

Exemple :
$Env0$

```
# let x = 1 ;;
```

$Env1 = [(x,1) >< Env0]$

```
# let x = 2 ;;
```

$Env2 = [(x,2) >< Env1]$

```
# x + 3 ;;
-  : int = 5
# y + 3 ;;
^^^identificateur non défini
```

Les liaisons obtenues par une définition globale sont donc valables pour toute la session Caml, ou bien jusqu'à ce que le même identificateur soit lié à une autre valeur.
Remarque : Il est possible de regrouper plusieurs définitions dans une même requête en utilisant le mot-clef `and` : `let id1 = exp1 and id2 = exp2 and .. idn = expn ;;` Les définitions sont alors *simultanées*.

Exercice

Donner la valeur de chaque expression et décrire l'évolution de l'environnement :

```
# let annee = 2020 ;;
# annee ;;
# annee + 2 ;;
# let futur = 2021 ;;
# futur = annee + 1 ;;
# annee_future ;;
# let annee_future = annee + 1 ;;
```

```
# let annee = 2000  ; ;
# annee_future  ; ;
```

Exercice

Donner la valeur de chaque expression et décrire l'évolution de l'environnement :

```
# let val = 8  ; ;
# let double = val * 2 and triple = val*3  ; ;
# let moitie = val/2 and quart = moitie/2  ; ;
# moitie  ; ;
```

4. Définitions locales

Définition 5

Nous appelons **définition locale**, une définition provoquant une liaison dont la durée de vie est limitée au temps de l'évaluation d'une expression. Une telle définition est obtenue par la requête :

```
let id = exp1 in exp2  ; ;
```

L'intérêt d'une définition locale est de pouvoir utiliser une liaison juste le temps où on en a besoin, sans surcharger l'environnement.

Exemple :
On se place dans un environnement Env et on demande l'évaluation de l'expression suivante :

```
let x = 1 in 3 + x  ; ;
```

Cela va créer un environnement temporaire $Env_t = [(x,1) >< Env]$. L'expression $3+x$ est alors évaluée dans Env_t, donc elle vaut 4.
L'environnement temporaire est alors détruit et on retrouve l'environnement Env.

Important

Si nous notons Env l'environnement actif, une définition locale `let ident = exp1 in exp2` provoque 4 actions :

1. L'expression `exp1` est évaluée dans l'environnement Env, soit v_1 sa valeur
2. Un environnement temporaire est crée en ajoutant la liaison $(ident, v_1)$ en tête de liste, on passe donc dans l'environnement temporaire

$$Env_t = [(ident, v_1) >< Env]$$

3. L'expression `exp2` est évaluée dans Env_t, sa valeur v_2 est le résultat de l'évaluation
4. L'environnement Env_t est détruit, on revient donc à Env_t. $ident$ n'est donc plus lié à la valeur v_1.

La définition locale est une expression, contrairement à la définition globale.

Remarque : Il est possible d'imbriquer les définitions locales :

```
Env0
```

```
# let x = 1 in
                EnvT1 = [(x,1)><Env0]
    let y = x + 2 in
                EnvT2 = [(y,3),(x,1)><Env0]
        let z = x+y in
                EnvT3 = [(z,4),(y,3),(x,1)><Env0]
           x+y+z  ; ;
-  : int = 8
                retour à Env0
```

Exercice

Donner la valeur de chaque expression, et décrire l'évolution de l'environnement.

```
#let x = 2  ; ;
#let y = x+3  ; ;
#let x = y+5  ; ;
#let z = y*2 in x+z+y*y ; ;
#let x = 3 in x*x+2*x*y+4*y  ; ;
#let x = 1 in x=2*x  ; ;
#let x = 0 in x=2*x ; ;
```

Exercice

Même consigne pour ces définitions locales emboîtées.

```
# let x=5 in
let prod = x*x in
prod+prod*prod ; ;


#let resultat= let x=5 in
let prod = x*x in
prod +prod*prod ; ;


#let val= let x=3 and y=4 in
let x=x+y and y=x-y in
x*x+y*y ; ;

#let y=2 in val*val+2val*y  ; ;
```

Exercice

Même consigne, avec des sélections.

```
#if (1=1) then "salut" else "au revoir" ; ;

#let x= 3 in if(x<0) then x else x*x ; ;

#if (5>0) then 1 else "erreur" ; ;

#let x=3 and y=3 in
     let y=y*x in
     if y mod 2 = 0 then "pair"
          else "impair" ; ;
```