

# PROGRAMMATION FONCTIONNELLE AVANCÉE

<http://www-lipn.univ-paris13.fr/~saiu/teaching/PFA-2010/>

## SYNTAXE SIMPLIFIÉE (ET IDÉALISÉE) DE OCAML AVEC SÉMANTIQUE INFORMELLE (WRITTEN BY JEAN-VINCENT LODDO)

### Constantes et motifs correspondants (données prédéfinies)

|                    | Expressions<br>de type<br>$t ::=$  | Expressions<br>de valeur<br>$e ::=$  | Expressions<br>de motif<br>$p ::=$   | Sémantique informelle<br>de l'expression  | Identificateurs<br>prédéfinis<br>associés au type   |
|--------------------|--|--|--|---|---|
| <i>int</i>         | <code>int</code>   | <code>k</code>   | <code>k</code>   | l'entier $k$  | <code>+ * - / mod</code>  |
| <i>float</i>       | <code>float</code>   | <code>k.   k<sub>1</sub>.k<sub>2</sub></code>                                | <code>k.   k<sub>1</sub>.k<sub>2</sub></code>  | le rationnel $k$ ou $k_1.k_2$   | <code>+. *. -. /.<br/>**</code>   |
| <i>char</i>        | <code>char</code>  | <code>'c'</code>   | <code>'c'</code>   | le caractère $c$  | <code>int_of_char</code>  |
| <i>string</i>      | <code>string</code>  | <code>"c<sub>1</sub>...c<sub>n</sub>"</code>                                 | <code>"c<sub>1</sub>...c<sub>n</sub>"</code>   | la chaîne de caractères $c_1...c_n$   | <code>^</code>  |
| <i>bool</i>        | <code>bool</code>  | <code>true   false</code>  | <code>true   false</code>  | le booléen vrai ou faux correspondant   | <code>not &amp;&amp; (ou &amp;)<br/>   (ou or) =<br/>== &lt;&gt; (ou !=)<br/>&lt; &gt; &lt;= &gt;=</code> |
| <i>unit</i>        | <code>unit</code>  | <code>()</code>  | <code>()</code>  | l'unique élément de $unit = \{\bullet\}$ (résultat des commandes provoquant des effets de bord)   | <code>print_int<br/>print_float<br/>print_char<br/>print_string<br/>..</code>                             |
| <i>product</i>     | <code>t<sub>1</sub> * ... * t<sub>n</sub> (n ≥ 2)</code>   | <code>e<sub>1</sub>, ..., e<sub>n</sub></code>                               | <code>p<sub>1</sub>, ..., p<sub>n</sub></code>   | l' $n$ -plet $(v_1, \dots, v_n)$ des valeurs représentées par $e_1 \dots e_n$   | <code>fst snd (n=2)</code>  |
| <i>disjonction</i> | <code>  X<sub>1</sub> [ of t<sub>1</sub> ]<br/>:<br/>  X<sub>n</sub> [ of t<sub>n</sub> ]</code> | <code>X<sub>i</sub> [ e ]</code>   | <code>X<sub>i</sub> [ p ]</code>   | le terme ayant $X$ à la racine et possédant éventuellement le sous-terme représenté par $e$ comme unique fils   |   |
| <i>list</i>        | <code>t list</code>  | <code>[]<br/><br/>e<sub>1</sub>; e<sub>2</sub></code>                        | <code>[]<br/><br/>p<sub>1</sub>; p<sub>2</sub></code>  | la liste vide<br><br>la liste composé de la valeur $v$ (représentée par $e_1$ ) suivie des valeurs de la liste $\ell$ (repr. par $e_2$ )  | <code>@ hd tl nth<br/>rev<br/><br/>map iter<br/>exists find<br/>mem assoc<br/>sort ..</code>              |
| <i>record</i>      | <code>{x<sub>1</sub>:t<sub>1</sub>; ...; x<sub>n</sub>:t<sub>n</sub>}</code>                     | <code>{x<sub>1</sub>=e<sub>1</sub>; ...; x<sub>n</sub>=e<sub>n</sub>}</code> | <code>{x<sub>i<sub>1</sub></sub>=p<sub>i<sub>1</sub></sub>; ...; x<sub>i<sub>k</sub></sub>=p<sub>i<sub>k</sub></sub>}</code> | l'enregistrement composé des champs $x_1 \dots x_n$ affectés respectivement aux valeurs $v_1 \dots v_n$ représentés par les expressions $e_1 \dots e_n$ . Dans la construction du motif $\{i_1, \dots, i_k\} \subseteq \{1, \dots, n\}$ . |   |
| <i>exception</i>   | <code>exn</code>   | <code>raise X [ e ]</code>   | <code>X [ p ]</code>   | exception représentée par le constructeur $X$ du type somme <code>exn</code> (extensible par le <code>oplevel</code> )  |   |
| <i>(.)</i>         | <code>(t)</code>   | <code>(e)</code>   | <code>(p)</code>   | la valeur représentée par $e$   |   |

## Expressions de calcul

|          | Expressions<br>de type<br>$t ::=$ | Expressions<br>de valeur<br>$e ::=$                                       | Sémantique informelle<br>de l'expression   |
|----------|-----------------------------------|---|--|
| function | $t_1 \rightarrow t_2$             | function<br>  $p_1 \rightarrow e_1$<br>:<br>  $p_n \rightarrow e_n$       | la fonction qui à tout $v \in t_1$ associe la valeur de $t_2$ représentée le premier $e_i$ dont le motif $p_i$ unifie avec la valeur $v$ qui sera donnée en argument. L'expression sélectionnée $e_i$ est évaluée dans l'environnement d'exécution augmenté par tous les liens résultant de l'unification (matching) de $p_i$ avec $v$ |
| apply    |                                   | $e_1 e_2$   | le résultat de l'application de la valeur fonctionnelle représentée par $e_1$ à l'argument représenté par $e_2$  |
| bloc     |                                   | let $d$ in $e_1$  | la valeur représentée par $e_1$ dans l'environnement d'exécution $\rho$ augmenté par tous les liens provoqués par les définitions dans $d$   |
| name     | 'a                                | $x$   | la valeur de l'identificateur $x$ dans l'environnement d'exécution $\rho$  |
| sel      |                                   | $x.e$   | la valeur du champ $x$ de l'enregistrement représenté par $e$ dans l'environnement d'exécution $\rho$  |
| try      |                                   | try $e_0$ with<br>  $p_1 \rightarrow e_1$<br>:<br>  $p_n \rightarrow e_n$ | la valeur de $e_0$ si la calcul se déroule normalement; sinon, l'exception levée par le calcul de $e_0$ sera unifiée avec le premier motif compatible et le résultat sera la valeur calculée par l'expression associée à ce motif  |

### Expressions impératives (commandes de type unit ou liées au type unit)

```

e ::= while e0 do e1 done
    | for x = e0 (to|downto) e1 do e2 done
    | if e0 then e1
    | e1 ; e2
    | ref e
    | !e
    | e0 := e1

```

### Autres types

$t ::= t \ x$  (application de l'expression de type  $t$  au type paramétré nommé  $x$ )

## Définitions de noms (de valeurs)

|                  | Définition de valeurs<br>$d ::=$ | Sémantique informelle de la définition   |
|------------------|----------------------------------|--|
| <i>simple</i>    | $x = e$                          | ajoute le lien $(x,v)$ à l'environnement d'exécution $\rho$ , où $v$ est le résultat de l'évaluation de $e$ dans $\rho$  |
| <i>pattern</i>   | $p = e$                          | ajoute à l'environnement d'exécution $\rho$ tous les liens résultants du matching entre le motif $p$ et le résultat $v$ de l'évaluation de $e$ dans $\rho$   |
| <i>parallèle</i> | $d_1 \text{ and } d_2$           | ajoute l'union des liens provoqués par les définitions $d_1$ et $d_2$ évaluées sur le même environnement d'exécution   |
| <i>récursive</i> | $\text{rec } d$                  | ajoute à l'environnement d'exécution les liens provoqués par $d$ considérant que les expressions contenues dans $d$ peuvent déjà utiliser tous les noms définis par $d$ , malgré qu'ils soient encore en cours de définition. ( <b>idéalisat</b> ion : le <i>rec</i> doit en réalité être écrit juste après le <i>let</i> , et ne peut donc se trouver dans une sous-définition) |

## Instructions Top Level

|                     | Définition de valeur/type $l ::=$   | Sémantique informelle  |
|---------------------|---|--|
| <i>valdef</i>       | $\text{let } d \ ; ;$   | ajoute à l'environnement d'exécution $\rho$ tous les liens de la forme (nom,valeur) provoqués par la définition $d$ .  |
| <i>typedef</i>      | $\text{type } x = t \ ; ;$  | enregistre dans l'environnement de <i>typage</i> que le nom $x$ est synonyme du type $t$ . Cette instruction est nécessaire pour toute utilisation successive d'un type <u>enregistrement</u> ou <u>somme</u> . Autrement dit, les types enregistrement et somme doivent être nommés de façon préliminaire à l'utilisation de leurs expressions ou motifs. |
| <i>parameter</i>    | $\text{type } 'a \ x = t \ ; ;$   | enregistre dans l'environnement de <i>typage</i> que le nom $x$ est synonyme (de la famille) du type $t$ paramétré par $'a$ .  |
| <i>parameters</i>   | $\text{type } ('a_1, \dots, 'a_n) \ x = t \ ; ;$  | enregistre dans l'environnement de <i>typage</i> que le nom $x$ est synonyme (de la famille) du type $t$ paramétré par $'a_1, \dots, 'a_n$ .   |
| <i>general form</i> | $\text{type}$<br>$['a ('a_1, \dots, 'a_n)] \ x_1 = t_1$<br>$\text{and}$<br>$\vdots$<br>$\text{and}$<br>$['a ('a_1, \dots, 'a_m)] \ x_k = t_k$<br>$ ; ;$ | forme générale de la définition de types : plusieurs types mutuellement récursifs sont définis, chacun éventuellement paramétré par un ou plusieurs paramètres. Cette définition enregistre dans l'environnement de <i>typage</i> tous les noms définis, leur type (avec paramètres éventuels) et leur relation mutuelle.                                  |
| <i>exn</i>          | $\text{exception } X \ [\text{of } t] \ ; ;$  | étend le type $\text{exn}$ par le nouveau constructeur   |
| <i>eval</i>         | $e \ ; ;$   | évalue l'expression $e$ dans l'environnement d'exécution courant.  |
| <i>séq</i>          | $l_1 \ l_2$   | composition séquentielle des deux instructions de top level $l_1$ et $l_2$   |

## Sucre syntaxique (macros ou abréviations)

### Expressions

|                | Abréviation d'expression <b>e</b>   | Développement   |
|----------------|---|---|
| <i>listes</i>  | <code>[e<sub>1</sub> ; ... ; e<sub>n</sub>]</code>  | <code>e<sub>1</sub> :: ... :: e<sub>n</sub> :: []</code>  |
| <i>match</i>   | <code>match e<sub>0</sub> with<br/>  p<sub>1</sub> -&gt; e<sub>1</sub><br/>:<br/>  p<sub>n</sub> -&gt; e<sub>n</sub></code> | <code>(function<br/>  p<sub>1</sub> -&gt; e<sub>1</sub><br/>:<br/>  p<sub>n</sub> -&gt; e<sub>n</sub>) e<sub>0</sub></code> |
| <i>cond</i>    | <code>if e<sub>0</sub> then e<sub>1</sub> else e<sub>2</sub></code>   | <code>match e<sub>0</sub> with true -&gt; e<sub>1</sub>   false -&gt; e<sub>2</sub></code>                                  |
| <i>fun</i>     | <code>fun p<sub>1</sub> ... p<sub>k</sub> -&gt; e</code>  | <code>function p<sub>1</sub> -&gt; function p<sub>2</sub> -&gt; ... -&gt; function p<sub>k</sub> -&gt; e</code>             |
| <i>let-fun</i> | <code>let x p<sub>1</sub> ... p<sub>k</sub> = e<sub>0</sub> in e<sub>1</sub></code>   | <code>let x = (fun p<sub>1</sub> ... p<sub>k</sub> -&gt; e<sub>0</sub>) in e<sub>1</sub></code>                             |

### Motifs

|               | Abréviation de motif <b>p</b>                      | Développement  |
|---------------|--|--|
| <i>listes</i> | <code>[p<sub>1</sub> ; ... ; p<sub>n</sub>]</code> | <code>p<sub>1</sub> :: ... :: p<sub>n</sub> :: []</code> |

### Top Level

|                | Abréviation de toplevel <b>l</b>                          | Développement   |
|----------------|---|---|
| <i>let-fun</i> | <code>let x p<sub>1</sub> ... p<sub>k</sub> = e ;;</code> | <code>let x = fun p<sub>1</sub> ... p<sub>k</sub> -&gt; e ;;</code> |

### Récapitulatif des méta-identificateurs (catégories syntaxiques) utilisés

|             |  |
|-------------|--|
| <b>l</b>    | instructions top level   |
| <b>e</b>    | expressions  |
| <b>p</b>    | motifs   |
| <b>t</b>    | types  |
| <b>d</b>    | définitions de noms (de valeurs)   |
| <b>k</b>    | entiers  |
| <b>c</b>    | caractères   |
| <b>x, a</b> | identificateurs (noms) commençant par une lettre minuscule (ou bien <code>_</code> ) |
| <b>X</b>    | identificateurs (noms) commençant par une lettre majuscule                           |