Programmation impérative en OCaml

En Ocaml, ce que l'on appellera instructions sont des expressions de type `unit`*.*

1 Expression de type `unit`

On rappelle que le type `unit` a une unique valeur, `()`. Les expressions de type `unit` ne portent donc aucune information mais sont intéressantes pour l'effet que produit leur évaluation, appelé **effet de bord**.

Exemple :

```
1 assert (1 == 2);
2 print_string "Hello world!";
```

Remarque : `let` n'est pas une expression de type `unit`.

Remarque : On ne peut pas juxtaposer des expressions de type `unit`. Il faut les séparer par points-virgules.

Exemple : `e1;e2` est une expression de type et de valeur qui sont ceux de `e2`. L'évaluation de cette expression consiste en l'évaluation de `e1` puis l'évaluation de `e2`. C'est équivalent à `let _ = e1 in e2`. Le `";"` est appelé **séquence**.

Remarque : Attention : si `e1` n'est pas du type `unit`, cela déclenche un avertissement qui indique que la valeur de e1 a été perdue.

Remarque : Bien sûr, on peut généraliser à plus de 2 expressions...

Pour gagner en lisibilité, il est possible d'utiliser les mots-clés `begin` et `end` à la place des parenthèses.

2 Références

Jusque là, nous avions appelé (à tort) *variables* des **valeurs nommées** dont la valeur ne changeait pas au cours du temps. En Ocaml, lorsque l'on veut une "vraie" variable, c'est à dire une case mémoire dont on va pouvoir modifier la valeur, il faut le préciser grâce au mot-clé `ref`.

Exemple :

```
1 let i = ref 1 in while (!i<3) do (print_int !i; i := !i+1) done;;
```

`ref e`, où `e` est une expression de type `t` de valeur `v`, est une expression de type `t ref`, et de valeur `{contents = v}`, c'est à dire une case mémoire contenant `v` la valeur de `e`.

L'évalulation de cette expression consiste en :

- L'évaluation de `e` en une valeur `v`.
- La réservation d'une case mémoire pouvant stocker une donnée de type `t`.
- L'initialisation de la donnée stockée dans cette case mémoire à `v`.

Remarque : On ne peut pas déclarer/créer une variable sans lui donner de valeur ! La création d'une nouvelle variable se fait comme suit :

Déclaration d'une variable : Pour déclarer une variable en Ocaml, la syntaxe est la suivante :

```
let var = ref e
```

où `var` est un indentificateur et `e` est une expression de type `t` et de valeur `v`

Accès à la valeur d'une variable : Attention, dans l'expression précédente, `var` est un identificateur ! Si l'on s'intéresse à la valeur identifiée, il faut utiliser l'opérateur `![nom de la variable]`, équivalent au `*[nom de la variable]` en C :

```
!var
```

où `var` est une expression de type `t ref`, est une expression de type `t` et de valeur la valeur enregistrée dans la référence `var`.

Modification de la valeur d'une variable : Il faut utiliser `:=` à la place de `=`

```
var := e
```

où `var` est une expression du type `t ref`, `e` est une expression de type `t`, est une expression de type `unit` (et donc de valeur `()`). Son évaluation consiste en l'évaluation de `e` en une valeur `v`, puis la modification de la valeur contenue dans la référence `var` en `v`.

Remarque : Attention, n'écrivez pas `a := a+2`.

Remarque : Attention, pour copier la valeur d'une variable dans une nouvelle variable, il faut faire une création de variable avec `ref`. Le "`=`" teste l'égalité entre valeurs, le "`==`" teste l'égalité entre adresses mémoires.

```
var1 = var2
```

où `var1` et `var2` sont des expressions de type `t ref` est une expression de type bool et de valeur `true` si `var1` et `var2` contiennent la même valeur (i.e `(!var1) = (!var2)`)

3 Boucles

3.1 Boucle `while`

```
while CONDITION do
    CORPS
done
```

où `CONDITION` est de type `bool`, `CORPS` est une expression de type `unit`. Son évaluation consiste en l'évaluation de `CONDITION`, puis :

- Si la valeur obtenue est `false`, l'évaluation est terminée
- sinon, on évalue `CORPS`, et on recommence.

En pratique :

- On crée d'abord une référence (au moins), disons `x`.
- `CONDITION` fait apparaître `!x`.

- `CORPS` modifie la valeur de la référence `x` (via `x:=...`).

3.2 Boucle `for`

```
for i = INIT to FINAL do
    CORPS
done
```

où `i` est un identificateur (**attention** : et pas une référence), `INIT` et `FINAL` sont de type `int`, et `CORPS` une expression de type `unit`, dans laquelle peut apparaître `i`. Son évaluation consiste en l'évaluation de `INIT` et `FINAL` en $d \in \mathbb{Z}$ et $f \in \mathbb{Z}$, puis pour chaque $k \in [\![d, f]\!]$, l'évaluation de `CORPS` où `i` est évaluée à k.

Remarque : Pour un comptage décroissant, on pourra utiliser `downto` à la place de `to`.

Remarque : Contrairement à C, OCaml ne propose pas de rupture de boucle avec un mot clé `break`. On peut cela dit obtenir un comportement similaire en utilisant des exceptions (https://discuss.ocaml.org/t/how-to-break-a-for-loop-in-a-nice-way/9852) :

```
exception Break
try
    for i = 0 to 9 do
        if i = 4 then raise Break
        else ()
    done
with
| Break -> ()
```

3.3 Tableaux

Les `array` sont en Ocaml la même chose que les tableaux en C. Ils permettent de stocker un nombre fini d'éléments de même type **Création d'un tableau à une dimension :**

```
Array.make <taille du tableau> <valeur>
```

Création d'un tableau à deux dimensions

```
Array.make_matrix <nombre de lignes/tableaux> <nombre de colonnes/taille des tableaux>
↪  <valeur>
```