

TP n°2

On se concentre ici sur l'évaluation des expressions parsées par le code développé dans le TP 1. Pour plus d'homogénéité, on vous propose de repartir de l'archive `inter_tp2.zip`.

1 Exploration du projet

Cette suite est composée des mêmes fichiers que précédemment, dont les principaux.

- `lexer.mll` : spécification de l'analyseur lexical
- `parser.mly` : spécification de l'analyseur syntaxique
- `ast.ml` : définition de l'arbre de syntaxe abstraite **et de l'évaluateur**

C'est donc sur le fichier `ast.ml` que nous nous concentrerons (vous devrez tout de même mettre à jour `lexer.mll` et `parser.mly` lorsque des constructions syntaxiques seront rajoutées au cours du TP).

Tout d'abord, on remarque que l'interpréte se comporte différemment.

```
$ ./tp
1 + 2 ;;
(1) + (2) => 3
1 + 2.0 ;;
(1) + (2.) => 3.
1.0 + 2.0 ;;
(1.) + (2.) => 3.
true ;;
true => true
"une chaine" ;;
"une chaine" => "une chaine"
"une chaine " + "de caractères" ;;
("une chaine ") + ("de caractères") => "une chaine de caractères"
Bye
```

Les différents types du TP1 sont bien gérés mais l'interpréte est maintenant capable d'effectuer l'évaluation de certaines expressions. Le fichier `ast.ml` contient l'essentiel pour réaliser cette évaluation (ces éléments sont appelés dans `main.ml` pour afficher l'évaluation).

1.1 Domaine d'évaluation

Les expressions sont évaluées en une valeur de type `value`, dont la définition est demandée dans le TP précédent. Des opérations doivent être programmées pour manipuler ces `values`.

```
type value =
| VInt of int
| VFloat of float
```

```

| VBool of bool
| VStr of string

...

let rec string_of_value v = match v with
| VInt i -> string_of_int i
| VFloat f -> string_of_float f
| VBool b -> string_of_bool b
| VStr s -> Printf.sprintf "\"%s\"" s
(* FIXME: transformer les caractères spéciaux '\n', '\r' et '\t' *)

...

let add_value v1 v2 = match (v1, v2) with
| VInt i1, VInt i2 -> VInt (i1 + i2)
| VInt i1, VFloat f2 -> VFloat (float_of_int i1 +. f2)
| VFloat f1, VInt i2 -> VFloat (f1 +. float_of_int i2)
| VFloat f1, VFloat f2 -> VFloat (f1 +. f2)
| VStr s1, VStr s2 -> VStr (s1 ^ s2)
| VStr s1, v2 -> VStr (s1 ^ (string_of_value v2))
| v1, VStr s2 -> VStr ((string_of_value v1) ^ s2)
| _, _ -> failwith
  (Printf.sprintf "Type mismatch: cannot add '%s' and '%s'"
    (string_of_value v1) (string_of_value v2))

```

Dans le code proposé, l'addition entre `values` est définie avec la fonction `add_value: value -> value -> value`. Elle permet d'additionner des valeurs numériques entre elles (avec une conversion des entiers vers les valeurs flottantes le cas échéant) et de concaténer des chaînes de caractères (la somme entre une chaîne et tout autre type de valeur convertit la valeur arbitraire en chaîne de caractères par `string_of_value`). Pour tout autre type de valeur, la fonction échoue.

Question 1. Programmer la fonction `neg_value: value -> value` qui retourne l'opposé d'une valeur numérique (cas `VInt` et `VFloat`) et échoue pour tout autre type.

Question 2. Programmer la fonction `mul_value: value -> value -> value` qui retourne la multiplication de valeurs numériques (avec les mêmes conversions implicites que l'addition de valeurs numériques) ; lorsque la multiplication est faite entre un entier et une chaîne de caractères, celle-ci est concaténée à elle-même autant de fois que spécifié par l'entier : `3 * "toto"` s'évalue en `"totototototo"` (on pourra utiliser la définition récursive `0 * s = ""` et `n * s = s + (n-1) * s`, l'addition étant donnée par `add_value`).

1.2 Fonction d'évaluation

L'objectif est une sémantique pour le langage du précédent. On rappelle que la grammaire des expressions considérées dans `parser.mly` est la suivante, dans une version

correspondant à la syntaxe abstraite.

$$\begin{array}{lcl} E & \rightarrow & V \\ | & & E + E \\ | & & E * E \\ | & & - E \end{array}$$

$$\begin{array}{lcl} V & \rightarrow & \underline{\text{int}} \\ | & & \underline{\text{true}} \\ | & & \underline{\text{false}} \\ | & & \underline{\text{float}} \\ | & & \underline{\text{str}} \end{array}$$

Dans `ast.ml`, le non-terminal E correspond au type `expr` et V au type `value`. Ainsi, étant donnée un expression (non-terminal E , type `expr`), on cherche à construire une valeur de type `value`. Pour une présentation formelle de cette association, nous posons la **notation**

$$e \rightarrow v$$

qui se lit

l'expression e (dérivée du non-terminal E) s'évalue en la valeur v.

Cette notation met en évidence le passage de la syntaxe (ici e) au domaine d'évaluation (ici v). Pour faciliter la lecture, on considère également les notations naturelles suivantes pour faire référence aux valeurs et aux opérations sur le type `value`.

- v réfère à une valeur de type `value`
- i réfère à une valeur `VInt` (à distinguer du terminal `int` de la grammaire)
- f réfère à une valeur `VFLOAT` (à distinguer du terminal `float` de la grammaire)
- b réfère à une valeur `VBool` (à distinguer des terminaux `true` et `false` de la grammaire)
- s réfère à une valeur `VStr` (à distinguer du terminal `string` de la grammaire)
- $+$ réfère à `add_value` (à distinguer du terminal `+` de la grammaire)
- \times réfère à `mul_value` (à distinguer du terminal `*` de la grammaire)
- $-$ réfère à `neg_value` (à distinguer du terminal `-` de la grammaire)

Avec ces notations, les règles d'évaluation prennent la forme suivante.

$$\frac{}{v \rightarrow v} (1) \quad \frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2}{e_1 + e_2 \rightarrow v_1 + v_2} (2) \quad \frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2}{e_1 * e_2 \rightarrow v_1 * v_2} (3) \quad \frac{e \rightarrow v}{-e \rightarrow -v} (4)$$

Voici comment lire les deux premières règles.

1. Une expression qui est une simple constante v (cas $E \rightarrow V$) s'évalue en v .
2. La somme **syntaxique** $e_1 + e_2$ s'évalue en l'addition **de valeur** $v_1 + v_2$ à partir du moment où e_1 s'évalue en v_1 et e_2 s'évalue en v_2 .

Ces deux premières règles sont déjà implémentées dans le code à travers la fonction `eval_expr: expr -> value`. En d'autres termes, $e \rightarrow v$ équivaut au fait que `eval_expr e` retourne la valeur v . Voici le code correspondant.

```
let rec eval_expr e = match e with
| ECst v -> v
| EAdd (e1, e2) ->
```

```

let v1 = eval_expr e1 in
let v2 = eval_expr e2 in
add_value v1 v2
| _ -> failwith (
  Printf.sprintf "Evaluation of '%s' not yet implemented"
    (string_of_expr e))

```

Voici comment lire ce code.

- La règle (1) est implémentée par le cas

```
| ECst v -> v
```

une expression qui est une simple constante v s'évalue en la valeur v.

- La règle (2) est implémentée par le cas

```

| EAdd (e1, e2) ->
  let v1 = eval_expr e1 in
  let v2 = eval_expr e2 in
  add_value v1 v2

```

lorsque l'expression e₁ s'évalue en v₁ et que l'expression e₂ s'évalue en v₂, alors l'expression e₁ + e₂ s'évalue en v₁ + v₂.

Question 1. Rajouter les cas manquants pour les règles (3) et (4) pour gérer la multiplication et la négation respectivement.

Question 2. Tester l'implémentation.

2 Opérateurs logiques, comparaisons et conditionnelles

2.1 Opérateurs logiques

On cherche à augmenter la syntaxe avec les productions

$$\begin{array}{l} E \rightarrow E \text{ and } E \\ E \rightarrow E \text{ or } E \\ E \rightarrow \text{not } E \end{array}$$

associées aux règles sémantiques

$$\frac{e_1 \rightarrow b_1 \quad e_2 \rightarrow b_2}{e_1 \text{ and } e_2 \rightarrow b_1 \wedge b_2} (5) \quad \frac{e_1 \rightarrow b_1 \quad e_2 \rightarrow b_2}{e_1 \text{ or } e_2 \rightarrow b_1 \vee b_2} (6) \quad \frac{e \rightarrow b}{\text{not } e \rightarrow \neg b} (7).$$

Domaine de valeurs.

Question 1. Ajouter dans le fichier `ast.ml` la fonction `and_value: value -> value -> value` effectuant la conjonction de valeurs uniquement lorsque celles-ci sont des VBool.

```

let and_value v1 v2 = match (v1, v2) with
  | VBool b1, VBool b2 -> VBool (b1 && b2)
  | _, _ -> failwith (

```

```
Printf.printf "Type mismatch: cannot and '%s' and '%s'"  

  (string_of_value v1) (string_of_value v2))
```

Question 2. Ajouter de la même manière la fonction `or_value`: `value -> value -> value` effectuant la disjonction de valeurs uniquement lorsque celles-ci sont des VBool.

Question 3. Ajouter de la même manière la fonction `not_value`: `value -> value` effectuant la négation d'une valeur uniquement lorsqu'elle est un VBool.

Syntaxe abstraite.

Question 4. Ajouter dans `ast.ml` les constructeurs `EAnd`, `EOr` et `ENot`.

Question 5. Mettre à jour `string_of_expr` en conséquence.

Évaluation.

Question 6. Ajouter les cas correspondant aux règles (5), (6) et (7) dans la fonction `eval_expr`.

Analyseur syntaxique.

Question 7. Ajouter dans le fichier `parser.mly` les tokens AND, OR et NOT ainsi que l'implémentation des productions

$$\begin{aligned} E &\rightarrow E \text{ and } E \\ E &\rightarrow E \text{ or } E \\ E &\rightarrow \text{not } E \end{aligned}$$

On considère que les opérateurs logiques sont moins prioritaires que les opérateurs arithmétiques, que la négation est prioritaire sur la conjonction, elle-même prioritaire sur la disjonction. Les opérateurs logiques binaires étant associatifs, on privilégiera les opérations à gauche (%left).

Analyseur lexicale.

Question 8. Ajouter dans le fichier `lexer.mll` la reconnaissance des unités lexicales `not`, `and` et `or`, correspondant respectivement aux tokens NOT, AND et OR.

2.2 Comparaison

On cherche à augmenter la syntaxe avec les productions

$$\begin{aligned} E &\rightarrow E == E \\ E &\rightarrow E != E \\ E &\rightarrow E \leq E \\ E &\rightarrow E \leq= E \\ E &\rightarrow E \geq E \\ E &\rightarrow E \geq= E \end{aligned}$$

associées aux règles

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2}{e_1 == e_2 \rightarrow v_1 = v_2} (8) \quad \frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2}{e_1 \leq e_2 \rightarrow v_1 < v_2} (9)$$

et au sucre syntaxique

- $e_1 \neq e_2 \equiv \underline{\text{not}}(e_1 == e_2)$
- $e_1 \geq e_2 \equiv \underline{\text{not}}(e_1 \leq e_2)$
- $e_1 \leq e_2 \equiv (e_1 \leq e_2) \underline{\text{or}} (e_1 == e_2)$
- $e_1 \geq e_2 \equiv \underline{\text{not}}((e_1 \leq e_2) \underline{\text{or}} (e_1 == e_2))$

Les opérations liées aux opérateurs = et < utilisés dans les règles (8) et (9) correspondent respectivement aux fonctions `equal_value`: `value -> value -> value` et `lt_value`: `value -> value -> value`. Elles comparent les valeurs en utilisant les opérateurs classiques ocaml (= et <) lorsque les types sont homogènes. Par exemple :

```
let eq_value v1 v2 = match (v1, v2) with
| VBool b1, VBool b2 -> VBool (b1 = b2)
| VInt i1, VInt i2 -> VBool (i1 = i2)
| VFloat f1, VFloat f2 -> VBool (f1 = f2)
| VStr s1, VStr s2 -> VBool (s1 = s2)
| _, _ -> failwith (
  Printf.sprintf "Type mismatch: cannot eq '%s' and '%s'"
    (string_of_value v1) (string_of_value v2))
```

Les opérateurs de comparaisons sont de priorité inférieure aux opérateurs arithmétiques mais supérieure aux opérateurs logiques. Ces opérateurs ne peuvent être utilisés consécutivement ($e_1 == e_2 == e_3$ est interdit par exemple) ; il n'y a donc pas d'associativité à considérer (%nonassoc).

Question 1. Ajouter les fonctions de comparaisons (`eq_value` et `lt_value`).

Question 2. Étendre le type `expr` avec les constructions abstraites uniquement, ainsi que la fonction `string_of_expr`.

Question 3. Étendre la fonction `eval_expr` en ajoutant les cas pour les règles (8) et (9).

Question 4. Étendre l'analyseur syntaxique (tokens et productions, sucre syntaxique).

Question 5. Étendre l'analyseur lexical.

2.3 Conditionnelle

On cherche à augmenter la syntaxe avec la production

$$E \rightarrow \underline{\text{if}}\ E \underline{\text{then}}\ E \underline{\text{else}}\ E$$

associée aux règles

$$\frac{e_c \rightarrow \top \quad e_1 \rightarrow v_1}{\underline{\text{if}}\ e_c \underline{\text{then}}\ e_1 \underline{\text{else}}\ e_2 \rightarrow v_1} (10) \quad \frac{e_c \rightarrow \perp \quad e_2 \rightarrow v_2}{\underline{\text{if}}\ e_c \underline{\text{then}}\ e_1 \underline{\text{else}}\ e_2 \rightarrow v_2} (11).$$

Noter que la condition doit s'évaluer comme un booléen et que seule l'une des deux branche doit évaluée suivant la valeur de ce booléen. Tous les opérateurs vus jusqu'ici sont prioritaires sur les conditionnelles. La précédence de la conditionnelle est associée à son dernier token : `else`.

Question. Étendre l'évaluateur avec les conditionnelles.

3 Identifiants et environnement

On cherche dans cette partie à permettre l'évaluation d'expressions contenant des identifiants ; par exemple

```
let x = 1 in
let y = 2 in
x + y ;;
```

s'évalue en 3.

Pour cela, nous allons ajouter à l'évaluateur la possibilité d'associer des valeurs à des identifiants. Dans l'exemple précédent, l'expression `x + y` prise isolément s'évalue en 3 dans l'environnement où `x` est associé à 1 et `y` à 2. Cet environnement est construit par les deux constructions `let ... in ...`.

3.1 Les environnements

Mathématiquement parlant, un environnement est une fonction (partielle) associant des valeurs à des noms. Nous noterons les environnements Γ .

Informatiquement parlant maintenant, un environnement est un dictionnaire dont la clé est une chaîne de caractères (le nom des identifiants) et la valeur associée est de type `value`. Pour faire simple, nous allons traiter les environnements comme des listes de couples de type `(string * value) list`. Ainsi, la liste

```
[ ("x", 1) ; ("y", 2) ]
```

encode l'environnement de l'exemple précédent.

Les environnements sont manipulés à travers deux opérations principales.

- Obtenir la valeur associée à un identifiant dans un environnement. En `ocaml`, étant donné l'environnement `gamma` et l'identifiant `x`, on obtiendra la valeur associée ainsi.

```
List.assoc x gamma
```

- Ajouter une entrée dans un environnement. En `ocaml`, étant donné un environnement `gamma`, un identifiant `x` et une valeur `v`, on ajoute l'association (x, v) à l'environnement `gamma` ainsi.

```
(x, v) :: gamma
```

Noter un effet intéressant de cet ajout : si `x` est déjà présent dans `gamma`, la nouvelle association cache l'ancienne. En effet, la fonction `List.assoc` trouve la *première* occurrence. Cette propriété est très pratique pour coder la portée des définitions.

3.2 Mise à jour des règles de la sémantique

La notation $e \rightarrow v$ utilisée jusqu'ici n'est plus suffisante car elle ne nous permet pas de préciser un environnement dans lequel évaluer e . On fait évoluer la **notation** précédente pour tenir compte des environnements par

$$\Gamma \vdash e \rightarrow v$$

qui se lit

Dans l'environnement Γ , l'expression e s'évalue en la valeur v .

Cela paraît un changement plutôt petit. Néanmoins, il va nous demander de mettre à jour l'ensemble des règles de sémantique vues jusqu'ici.

$$\begin{array}{c}
 \frac{}{\Gamma \vdash v \rightarrow v} (1) \quad \frac{\Gamma \vdash e_1 \rightarrow v_1 \quad \Gamma \vdash e_2 \rightarrow v_2}{\Gamma \vdash e_1 \pm e_2 \rightarrow v_1 + v_2} (2) \\
 \\
 \frac{\Gamma \vdash e_1 \rightarrow v_1 \quad \Gamma \vdash e_2 \rightarrow v_2}{\Gamma \vdash e_1 * e_2 \rightarrow v_1 \times v_2} (3) \quad \frac{\Gamma \vdash e \rightarrow v}{\Gamma \vdash \underline{-} e \rightarrow -v} (4) \\
 \\
 \frac{\Gamma \vdash e_1 \rightarrow b_1 \quad \Gamma \vdash e_2 \rightarrow b_2}{\Gamma \vdash e_1 \text{ and } e_2 \rightarrow b_1 \wedge b_2} (5) \quad \frac{\Gamma \vdash e_1 \rightarrow b_1 \quad \Gamma \vdash e_2 \rightarrow b_2}{\Gamma \vdash e_1 \text{ or } e_2 \rightarrow b_1 \vee b_2} (6) \quad \frac{\Gamma \vdash e \rightarrow b}{\Gamma \vdash \underline{\text{not}} e \rightarrow \neg b} (7) \\
 \\
 \frac{\Gamma \vdash e_1 \rightarrow v_1 \quad \Gamma \vdash e_2 \rightarrow v_2}{\Gamma \vdash e_1 == e_2 \rightarrow v_1 = v_2} (8) \quad \frac{\Gamma \vdash e_1 \rightarrow v_1 \quad \Gamma \vdash e_2 \rightarrow v_2}{\Gamma \vdash e_1 \leq e_2 \rightarrow v_1 < v_2} (9) \\
 \\
 \frac{\Gamma \vdash e_c \rightarrow \top \quad \Gamma \vdash e_1 \rightarrow v_1}{\Gamma \vdash \underline{\text{if}} e_c \text{ then } e_1 \text{ else } e_2 \rightarrow v_1} (10) \quad \frac{\Gamma \vdash e_c \rightarrow \perp \quad \Gamma \vdash e_2 \rightarrow v_2}{\Gamma \vdash \underline{\text{if}} e_c \text{ then } e_1 \text{ else } e_2 \rightarrow v_2} (11)
 \end{array}$$

En gros, nous avons ajouté la mention de l'environnement Γ partout où une évaluation était faite. Côté implémentation, les conséquences vont naturellement concerter la fonction `eval_expr` dont le type va évoluer en

```
eval_expr: (string * value) list -> expr -> value
```

pour permettre d'établir la correspondance

$$\Gamma \vdash e \rightarrow v \quad \equiv \quad \text{eval_expr } \Gamma \ e == v.$$

La conséquence concrète sur le code est la suivante.

```
let rec eval_expr gamma e = match e with
| ECst v -> v
| EAdd (e1, e2) ->
  let v1 = eval_expr gamma e1 in
  let v2 = eval_expr gamma e2 in
  add_value v1 v2
| ...
```

Le nouvel argument `gamma` est ajouté à la fonction (`let rec eval_expr gamma e = ...`); chaque appel récursif fait appel à cet argument (par exemple, `... let v1 = eval_expr gamma e1 in ...`).

Question 1. Mettre à jour toute la fonction `eval_expr` pour prendre en compte un environnement d'évaluation `gamma`.

Question 2. Mettre à jour le code de `main.ml` qui appelle la fonction `eval_expr`. Il suffit pour cela de rajouter en paramètre à cet appel la liste vide, c'est-à-dire l'environnement vide où aucun identifiant n'a de valeur associée.

```
let _ =
(* Ouverture un flot de caractère ; ici à partir de l'entrée...*)
```

```

let source = Lexing.from_channel stdin in

(* Boucle infinie interrompue par une exception correspondant...
let rec f () =
  try
    (* Récupération d'une expression à partir de la source...
    let e = Parser.ansyn Lexer.anlex source in
    let v = Ast.eval_expr [] e in
    Printf.printf "%s => %s\n"
      (Ast.string_of_expr e) (Ast.string_of_value v);
    flush stdout;
    f ()
  with Lexer.Eof -> Printf.printf "Bye\n"
in

f ()

```

3.3 Identifiant

Maintenant que la notion d'environnement d'évaluation est prise en compte dans l'évaluateur, l'ajout des définitions et l'utilisation d'identifiants se font simplement.

Question 3. Augmenter la syntaxe avec les productions suivantes.

$$\begin{array}{l} E \rightarrow \underline{\text{id}} \\ E \rightarrow \underline{\text{let }} \underline{\text{id}} \underline{\equiv} E \underline{\text{ in }} E \end{array}$$

La première production dit qu'une expression peut être un identifiant. La seconde production augmente la grammaire pour permettre la définition de nouveaux identifiants (pour augmenter l'environnement initialement vide) dans un style proche du langage ocaml (construction `let ... in ...`).

Pour le token id nous considérerons *des suites de lettres (majuscules et minuscules), de chiffres et de underscores, commençant par une lettre, et n'autorisant pas deux underscores successifs et ne se terminant pas par un underscore*.

Pour ce qui est des priorités, la construction `let ... in ...` est proche des conditionnelles, l'expression `let x = 2 in 1 + x` doit être comprise `let x = 2 in (1 + x)`.

Les règles sémantiques pour les identifiants sont les suivantes.

$$\frac{\text{List.assoc } x \Gamma = v}{\Gamma \vdash x \rightarrow v} \quad (12) \qquad \frac{\Gamma \vdash e_1 \rightarrow v_1 \quad (x, v_1) :: \Gamma \vdash e_2 \rightarrow v_2}{\Gamma \vdash \underline{\text{let }} \underline{x} \underline{\equiv} e_1 \underline{\text{ in }} e_2 \rightarrow v_2} \quad (13)$$

4 Fonctions

On cherche enfin à étendre la syntaxe avec les constructions permettant de manipuler des fonctions

$$\begin{array}{l} E \rightarrow \underline{\text{fun }} \underline{\text{id}} \underline{\rightarrow} E \\ E \rightarrow E \underline{(} E \underline{)} \end{array}$$

dont les règles de sémantique sont

$$\overline{\Gamma \vdash \underline{\text{fun}}\ x \ \underline{->} e \rightarrow \langle x, e, \Gamma \rangle}^{(13)}$$

$$\frac{\Gamma \vdash e_1 \rightarrow \langle x, e, \Gamma' \rangle \quad \Gamma \vdash e_2 \rightarrow v \quad (x, v) :: \Gamma' \vdash e \rightarrow v'}{\Gamma \vdash e_1 \underline{(\ } e_2 \underline{)} \rightarrow v'}^{(14)}.$$

L'élément principal de cette extension est la *clôture* $\langle x, e, \Gamma \rangle$. En effet une fonction s'évaluera au moment de son appel (cf. les prémisses de la règle (14)). Cette évaluation demandera de restaurer l'environnement existant au moment de la création de la fonction. Une clôture contient donc ces deux informations : la fonction (*i.e.*, un nom d'argument et une expression constituant son corps) et l'environnement au moment de sa définition.

Question 1. Ajouter au type `value` le cas des clôtures et compléter la fonction `string_of_value`.

Question 2. Modifier le type `expr` pour prendre en compte les 2 nouveaux éléments de syntaxe et la fonction `string_of_expr`.

Question 3. Modifier l'évaluateur pour intégrer les règles (13) et (14).

Question 4. Compléter l'analyseur syntaxique dans `parser.mly` (proposer des solutions pour les conflits).

Question 5. Compléter l'analyseur lexical dans `lexer.mll`.