Machine universelle

I. Introduction.

L'objectif de ce TP est de coder, en OCaml, la machine universelle $\mathscr{U}$ permettant de calculer la fonction `interprète` définie par :

$$\texttt{interprète}\,(\langle\mathscr{M}\rangle, w) = \begin{cases} w' \text{ si } w \underset{\mathscr{M}}{\longrightarrow} w' \\ \text{non défini sinon} \end{cases}$$

pour une certaine machine $\mathscr{M}$. De même que dans le TD Théorie n°1, on utilise la notation $\langle\cdot\rangle$ pour dénoter la sérialisation. La fonction `interprète` aura la signature OCaml ci-dessous :

```
interprete : string -> string
```

L'entrée w a été supprimée : seul le code $\langle\mathscr{M}\rangle$ de la machine $\mathscr{M}$ est donné à la fonction `interprete`. L'entrée de la machine est « ajoutée » au début du code, sous la forme d'une expression de la forme

```
let input = [on placera l'entrée de la machine ici] in
[on placera le code de la machine ici]
```

Ainsi définie, la fonction `interprete` aura le même fonctionnement que `utop` : il évalue l'expression, sans prendre en compte une certaine entrée.

Pour les deux premières sections, on utilisera le dossier `tp2-1`. Dans la troisième section (vérification de types), on utilisera le dossier `tp2-2`. On copiera les fonctions écrites dans le dossier `tp2-1` dans le dossier `tp2-2`.

Ce TP est long, on n'attendra pas qu'il soit fini en 2 heures. Mais, si vous voulez continuer ce TP par la suite, vous pouvez. Des informations sur l'environnement de développement sont disponibles en fin de sujet (Section V).

La chaîne de production d'un programme est divisée en plusieurs parties. Premièrement, l'*analyse lexicale* (ou *lexer* en anglais) décompose le code source en une liste de symboles (*tokens*). Puis, l'*analyse syntaxique* (ou *parser* en anglais) permet de construire un *arbre syntaxique* (*abstract syntax tree* en anglais, souvent abrévié AST). Et, l'*analyse sémantique* vérifie que l'arbre syntaxique a un sens : dans notre cas, ce sera la vérification des types (cette étape est optionnelle pour ce TP, mais elle est importante dans le cadre d'un véritable langage de programmation). Enfin, l'*interprétation* de l'arbre syntaxique exécute le code entré.

Dans ce TP, nous considérons un « sous »-langage de OCaml, nommé OCaml−− (« OCaml moins moins »). Il contient les structures fondamentales de OCaml :

- la définition de variables (par exemple `let x: int = 3 in`)
- les opérations `+`, `-`, `*`, `/` sur des entiers,
- les opérations `>`, `>=`, `<`, `<=`, `=` et `<>` sur des entiers (et aussi des booléens pour les deux derniers).
- les types `int` et `bool`,
- les fonctions anonymes `fun x -> 2 * x`.

II. Quelques notes sur l'analyse syntaxique et lexicale.

Dans la figure suivante (Fig. 1, page 2), on représente l'analyse lexicale et sémantique sur un exemple.

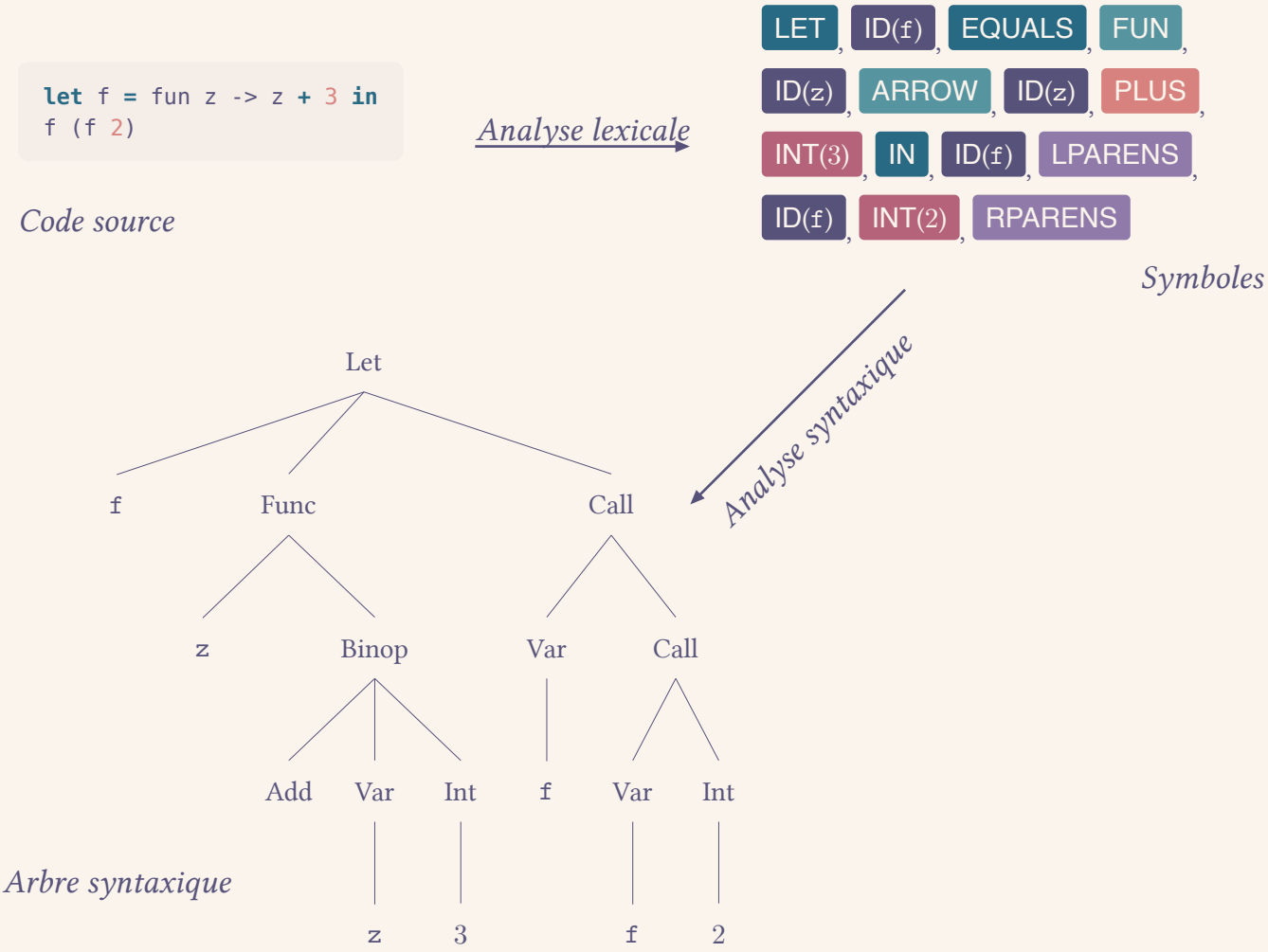

Fig. 1. Représentation de l'analyse lexicale et sémantique sur un exemple

L'analyse lexicale est réalisé à partir des automates : il suffit de définir des motifs à reconnaître, et comment les catégoriser (*i.e.* définir les symboles). L'analyse syntaxique est réalisée via des grammaires (notion vue plus tard dans l'année) : il suffit de définir les assemblages de symboles possibles, et quel impact cela aura sur l'arbre syntaxique. Cette partie n'est pas à réaliser, vous trouverez les fonctions nécessaires dans les documents associés au TP.

Le type `expr` représente l'arbre syntaxique, comme représenté sur la figure. Une fonction `parse` permet de convertir une chaîne de caractères en un arbre syntaxique. En cas d'erreur de syntaxe, une exception est levée ; on n'aura pas besoin de gérer ces erreurs de syntaxe.

```
type expr =
| Var    of string               (* Une variable *)
| Int    of int                  (* Un entier    *)
| Bool   of bool                 (* Un booléen   *)
| Binop  of bop * expr * expr    (* Une opération binaire *)
| Let    of string * expr * expr (* Une définition "let x = ... in ..." *)
| If     of expr * expr * expr   (* Une condition "if, then, else" *)
```

```
| Func   of string * expr        (* Une fonction "fun z -> ..." *)
| Call   of expr * expr          (* Un appel d'expression *)
```

Le type `bop` contient les différentes opérateurs binaires reconnus. Ainsi, il contient l'addition (`Add`), la soustraction (`Sub`), la multiplication (`Mult`), et la division (`Div`) de nombres entiers. Il contient également les opérateurs de comparaison (sur des entiers uniquement, retournant un booléen) $<$ (`Lt` , *less than*), $>$ (`Gt` , *greater than*), $\leq$ (`Leq` , *less than or equal to*), et $\geq$ (`Geq` , *greater than or equal to*). D'autres opérateurs pourront être définis, afin que OCaml−− se rapproche plus de OCaml.

Dans la suite du TP, le *parser* est la combinaison de l'analyseur sémantique et l'analyseur syntaxique.

Q1. Afin de vérifier le fonctionnement du *parser*, tester la fonction `parse` sur le code source de la Fig. 1. On comparera l'arbre obtenu à celui de la figure. On devrait obtenir le résultat ci-dessous.

```
Let ("f", Func ("z", Binop (Add, Var "z", Int 3)),
  Call (Var "f", Call (Var "f", Int 2)))
```

III. Interpréteur OCaml−−.

Réaliser un interpréteur peut se réaliser de plusieurs manières. Dans ce TP, nous prendrons l'approche de la *substitution*. Dans ce modèle, l'interpréteur s'exécute sur l'arbre de syntaxe du programme. Les variables sont stockées dans l'arbre de syntaxe, et non dans une autre mémoire.

Généralement, ce modèle n'est pas utilisé car il modifie l'arbre de syntaxe du programme. Mais, l'implémentation de cette méthode est assez simple.

III.1. Formalisation, notations.

Le type OCaml `expr` est représenté mathématiquement par un ensemble $\mathscr{E}$ défini par *induction nommée* à partir des règles **Var** représentant le variant `Var` , **Int** représentant le variant `Int` , **Bool** représentant le variant `Bool` , **Binop** représentant le variant `Binop` , **Let** représentant le variant `Let` , **If** représentant le variant `If` , **Func** représentant le variant `Func` , et **Call** représentant le variant `Call` . On identifiera, dans ces règles, le type `int` et l'ensemble $\mathbb{Z}$; le type `bool` et l'ensemble $\mathbb{B} = \{\boldsymbol{V}, \boldsymbol{F}\}$; le type `string` et un ensemble fini de variables $\mathscr{V} = \{\mathtt{x}, \mathtt{y}, \dots\}$; les opérateurs du type `bop` et leurs équivalents mathématiques, ainsi que la notation **?** quand l'opérateur est inconnu.

Les variables et les expressions seront différentiées dans leur notation. Les variables seront notées en police machine à écrire droite, par exemple $\mathtt{x}$. Les expressions seront notées en police roman italique, par exemple x.

III.2. Substitutions.

Étant données deux expressions e et $f \in \mathscr{E}$, et une variable $\mathtt{x}$, on définit la *substitution* de $\mathtt{x}$ par f dans e, notée $e[f \mapsto \mathtt{x}]$, par induction sur e :

- $(\textbf{Var}\ \mathtt{x})[\mathtt{x} \mapsto f] = f$,
- $(\textbf{Var}\ \mathtt{y})[\mathtt{x} \mapsto f] = \textbf{Var}\ \mathtt{y}$ avec $\mathtt{y} \neq \mathtt{x}$,
- $(\textbf{Int}\ n)[\mathtt{x} \mapsto f] = \textbf{Int}\ n$,
- $(\textbf{Bool}\ b)[\mathtt{x} \mapsto f] = \textbf{Bool}\ b$,
- $(\textbf{Binop}\ ?\ e_1\ e_2)[\mathtt{x} \mapsto f] = \textbf{Binop}\ ?\ e_1[\mathtt{x} \mapsto f]\ e_2[\mathtt{x} \mapsto f]$ pour une opération ?,
- $(\textbf{If}\ a\ b\ c)[\mathtt{x} \mapsto f] = \textbf{If}\ a[\mathtt{x} \mapsto f]\ b[\mathtt{x} \mapsto f]\ c[\mathtt{x} \mapsto f]$,

- $(\textbf{Call}\ e_1\ e_2)[\mathtt{x} \mapsto f] = \textbf{Call}\ e_1[\mathtt{x} \mapsto f]\ e_2[\mathtt{x} \mapsto f]$,
- $(\textbf{Func}\ \mathtt{x}\ e)[\mathtt{x} \mapsto f] = \textbf{Func}\ \mathtt{x}\ e$,
- $(\textbf{Func}\ \mathtt{y}\ e)[\mathtt{x} \mapsto f] = \textbf{Func}\ \mathtt{y}\ e[\mathtt{x} \mapsto f]$ si $\mathtt{x} \neq \mathtt{y}$ et que y n'est pas une variable libre de f,
- $(\textbf{Func}\ \mathtt{y}\ e)[\mathtt{x} \mapsto f] = \textbf{Func}\ \mathtt{y}\ e[\mathtt{x} \mapsto f']$ si $\mathtt{x} \neq \mathtt{y}$ et que y est une variable libre de f, et que l'on a renommé y dans f en f',[1]
- $(\textbf{Let}\ \mathtt{x}\ a\ b)[\mathtt{x} \mapsto f] = \textbf{Let}\ \mathtt{x}\ a[\mathtt{x} \mapsto f]\ b$,
- $(\textbf{Let}\ \mathtt{x}\ a\ b)[\mathtt{x} \mapsto f] = \textbf{Let}\ \mathtt{x}\ a[\mathtt{x} \mapsto f]\ b[\mathtt{x} \mapsto f]$ si $\mathtt{x} \neq \mathtt{y}$.

On définit l'ensemble des *variables libres* $\mathsf{FV}(e) \subseteq \mathscr{V}$ d'une expression $e \in \mathscr{E}$ par
- $\mathsf{FV}(\textbf{Var}\ \mathtt{x}) = \{\mathtt{x}\}$,
- $\mathsf{FV}(\textbf{Call}\ e\ f) = \mathsf{FV}(e) \cup \mathsf{FV}(f)$,
- $\mathsf{FV}(\textbf{Binop}\ ?\ e\ f) = \mathsf{FV}(e) \cup \mathsf{FV}(f)$,
- $\mathsf{FV}(\textbf{Int}\ n) = \emptyset$,
- $\mathsf{FV}(\textbf{Bool}\ b) = \emptyset$,
- $\mathsf{FV}(\textbf{Func}\ \mathtt{x}\ e) = \mathsf{FV}(e) \setminus \{\mathtt{x}\}$,
- $\mathsf{FV}(\textbf{Let}\ \mathtt{x}\ e_1\ e_2) = \mathsf{FV}(e_1) \cup (\mathsf{FV}(e_2) \setminus \{\mathtt{x}\})$,
- $\mathsf{FV}(\textbf{If}\ a\ b\ c) = \mathsf{FV}(a) \cup \mathsf{FV}(b) \cup \mathsf{FV}(c)$.

On représentera en OCaml un ensemble par une liste. On s'assurera que chaque élément n'apparait qu'une fois dans la liste représentant un ensemble.

Q2. Coder, en OCaml, une fonction `union : 'a list -> 'a list -> 'a list` qui réalise l'union de deux ensembles, représentés par des listes. (On pourra utiliser la fonction `List.mem` qui teste l'appartenance d'un élément dans une liste.)

On pourra définir un opérateur binaire `@@` tel que `e @@ f` réalise l'union de `e` et de `f`. Pour cela, il suffit d'ajouter le code ci-dessous.

```
let (@@) = union
```

Pour plus d'informations sur les opérateurs binaires, regardez la documentation de OCaml, et la fiche « topo » de Mr Journault.

Q3. Définir une fonction `fv : expr -> string list` telle que `fv e` corresponde à l'ensemble des variables libres $\mathsf{FV}(e)$, où e est la représentation mathématique de `e`.

Q4. Définir une fonction `substitution : expr -> expr -> string -> expr` qui réalise la substitution $e[\mathtt{x} \mapsto f]$ lors de l'appel `substitution` $e\ f$ `x`. On pourra supposer qu'aucune variable ne serait libre, quitte à modifier le code de la fonction `substitution`.

III.3. L'interprétation.

On définit la fonction partielle $\triangleright : \mathscr{E} \to \mathscr{E}$ par induction :
- $\triangleright(\textbf{Var}\ \mathtt{x}) = \square$,
- $\triangleright(\textbf{Bop}\ ?\ a\ b) = \triangleright(a)\ ?\ \triangleright(b)$,
- $\triangleright(\textbf{Int}\ n) = \textbf{Int}\ n$,
- $\triangleright(\textbf{Bool}\ b) = \textbf{Bool}\ b$,
- $\triangleright(\textbf{Let}\ \mathtt{x}\ e\ f) = \triangleright(f[\mathtt{x} \mapsto \triangleright(e)])$,
- $$\triangleright(\textbf{If}\ a\ b\ c) = \begin{cases} \triangleright(b) & \text{si } \triangleright(a) = \textbf{Bool}\ \boldsymbol{V}, \\ \triangleright(c) & \text{si } \triangleright(a) = \textbf{Bool}\ \boldsymbol{F}, \\ \square & \text{sinon}, \end{cases}$$

[1] Dans un premier temps, on n'implémentera pas ce cas. On supposera que toutes les variables ne seront pas libres.

- $\triangleright(\textsf{Func}\ \mathtt{x}\ e) = \textsf{Func}\ \mathtt{x}\ e$,
- $\triangleright(\textsf{Call}\ e\ f) = \begin{cases} \triangleright(g[\mathtt{x} \mapsto \triangleright(f)]) & \text{si } \triangleright(e) = \textsf{Func}\ \mathtt{x}\ g, \\ \square & \text{sinon.} \end{cases}$

Le « $\square$ » est un élément inconnu, dans le cas où la fonction $\triangleright$ n'est pas définie. Informatiquement, c'est une exception, une erreur.

Exemple. Avec l'arbre de la Fig. 1 (page 2), on a

$$
\begin{aligned}
&\triangleright(\textsf{Let}\ \mathtt{x}\ (\textsf{Func}\ \mathtt{z}\ (\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3)))(\textsf{Call}\ (\textsf{Var}\ \mathtt{f})(\textsf{Call}\ (\textsf{Var}\ \mathtt{f})(\textsf{Int}\ 2)))) \\
=\ &\triangleright((\textsf{Call}\ (\textsf{Var}\ \mathtt{f})(\textsf{Call}\ (\textsf{Var}\ \mathtt{f})(\textsf{Int}\ 2)))[\mathtt{f} \mapsto \triangleright(\textsf{Func}\ \mathtt{z}\ (\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3)))]) \\
=\ &\triangleright((\textsf{Call}\ (\textsf{Func}\ \mathtt{z}\ (\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3)))(\textsf{Call}\ (\textsf{Func}\ \mathtt{z}\ (\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3)))(\textsf{Int}\ 2)))) \\
=\ &\triangleright((\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3))[\mathtt{z} \mapsto \triangleright(\textsf{Call}\ (\textsf{Func}\ \mathtt{z}\ (\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3)))(\textsf{Int}\ 2))]) \\
=\ &\triangleright((\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3))[\mathtt{z} \mapsto \triangleright(\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3))[\mathtt{z} \mapsto (\textsf{Int}\ 2)]]) \\
=\ &\triangleright((\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3))[\mathtt{z} \mapsto \triangleright(\textsf{Binop Add}\ (\textsf{Int}\ 2)(\textsf{Int}\ 3))]) \\
=\ &\triangleright((\textsf{Binop Add}\ (\textsf{Var}\ \mathtt{z})(\textsf{Int}\ 3))[\mathtt{z} \mapsto \textsf{Int}\ 5])) \\
=\ &\triangleright(\textsf{Binop Add}\ (\textsf{Int}\ 5)(\textsf{Int}\ 3)) \\
=\ &\textsf{Int}\ 8
\end{aligned}
$$

Q5. Définir des exceptions pour les différents cas non définis de la fonction $\triangleright$.

Q6. Coder la fonction $\triangleright$ sous forme d'une fonction de signature `interp : expr -> expr`. On créera des fonctions mutuellement récursives pour gérer les différents cas de la définition de la fonction $\triangleright$: `interp_bop`, `interp_let`, `interp_if`, `interp_call`.
Cette question est bien plus longue que les autres.

III.4. Interface utilisateur.

Pour le moment, nous avons défini des fonctions OCaml, mais nous n'avons pas créé un outil prêt à être utilisé. C'est ce que nous allons réaliser dans cette sous-section.

Q7. Coder une fonction `string_of_value : expr -> string` qui associe une chaîne de caractère à une *valeur* (donc uniquement booléen, et entier).

Q8. Coder la fonction `interprete : string -> string`.

Passer ce point, nous avons codé la machine universelle $\mathscr{U}$. Mais, nous allons continuer quand même. Le programme `utop` est un *REPL* (*Read–Eval–Print Loop*) : il lit la commande de l'utilisateur, l'exécute, puis affiche le résultat, et recommence.

Q9. Définir une fonction `get_input : unit -> string` qui récupère l'entrée de la console. Attention, le programme peut être sur plusieurs lignes : on ne s'arrêtera qu'après `;;`.

Q10. Définir une fonction `run_interpreter : unit -> unit` qui lance l'interpréteur.

IV. Vérification de types.

Dans la suite de ce TP, nous allons vérifier les types dans le programme à exécuter.

IV.1. Formalisation mathématique, notations.

Nous définissions l'ensemble des *types* $\mathscr{T}$ en OCaml−− comme un ensemble défini par induction nommée à partir des règles **IntType** et **BoolType** d'arité 0, et du type **FuncType** d'arité 2. Ainsi, on définit en OCaml le type associé `typ` aux éléments de $\mathscr{T}$.

```
type typ
| BoolType
| IntType
| FuncType of typ * typ
```

La « vérification de types » consiste à vérifier que le type de toute expression inclue dans le programme a un type correct, en supposant les sous-expressions sont correctement typée.

Un *environnement de typage* est un ensemble de couples variable type, donc un élément de $\wp(\mathscr{V} \times \mathscr{T})$ tel que, pour chaque variable $v \in \mathscr{V}$, il n'existe qu'un unique $t \in \mathscr{T}$, ou aucun tel que (v, t) soit un élément de l'environnement. On le notera $\mathrm{env} = \{\mathtt{x} \mapsto t_{\mathtt{x}}, \mathtt{y} \mapsto t_{\mathtt{y}}, \dots\}$. On notera bool le type associé à la règle **BoolType**, int le type associé à la règle **IntType**, $e \to f$ le type associé à la règle **Func** $e\ f$. Comme env est un ensemble, on pourra noter $\mathrm{env} \cup \{\mathtt{x} \mapsto t\}$ pour ajouter un type. Si x avait déjà un type associé auparavant, il est redéfini par cet union.

Pour $e \in \mathscr{E}$, $t \in \mathscr{T}$ et un environnement env, on notera $\mathrm{env} \vdash e : t$ dès lors que l'expression e a pour type t. On définit inductivement cette notation.

- On a toujours $\mathrm{env} \vdash$ **Int** n : int et $\mathrm{env} \vdash$ **Bool** b : bool.
- On a $\mathrm{env} \vdash$ **Var** x : t où $(\mathtt{x}, t) \in \mathrm{env}$.
- Si $\mathrm{env} \vdash e_1 : t_1$ et $\mathrm{env} \cup \{\mathtt{x} \mapsto t_1\} \vdash e_2 : t_2$, alors $\mathrm{env} \vdash$ (**Let** x $e_1\ e_2$) : t_2.
- Si $\mathrm{env} \vdash e_1$: int et $\mathrm{env} \vdash e_2$: int, alors $\mathrm{env} \vdash$ (**Binop** ? $e_1\ e_2$) : int pour ? $\in \{+, -, \times, /\}$.
- Si $\mathrm{env} \vdash e_1$: int et $\mathrm{env} \vdash e_2$: int, alors $\mathrm{env} \vdash$ (**Binop** ? $e_1\ e_2$) : bool pour ? $\in \{\leq, \geq, <, >\}$.
- Si $\mathrm{env} \vdash a$: bool, $\mathrm{env} \vdash b : t$ et $\mathrm{env} \vdash c : t$, alors $\mathrm{env} \vdash$ (**If** $a\ b\ c$) : t.
- Si $\mathrm{env} \cup \{\mathtt{x} \mapsto t_1\} \vdash f : t_2$, alors $\mathrm{env} \vdash$ (**Func** z f) : $t_1 \to t_2$.
- Si $\mathrm{env} \vdash f : t_1$ et $\mathrm{env} \vdash e : t_1 \to t_2$, alors $\mathrm{env} \vdash$ (**Call** $e\ f$) : t_2.

Pour stocker un environnement, on utilisera une liste de couples variables type. On définit le type `environment` comme ci-dessous.

```
type environment = (string * typ) list
```

Q11. Définir une fonction `find_type : environment -> string -> typ option` qui, lors de l'appel `find_type` v env, renvoie `Some` t si $(e, t) \in$ env, et `None` sinon.

On modifie le *parser* pour que les variants `Let` et `Func` contiennent également le type (de type `typ`) de la variable définie.

Q12. Ajuster le code défini précédemment avec cette modification.

Q13. Définir une fonction `type_of : environment -> expr -> typ` qui, à une expression e et un environnement env, associe un type t tel que $\mathrm{env} \vdash e : t$. On réalisera, comme pour la question , des fonctions mutuellement récursives nommées `type_of_bop`, `type_of_if`, ...

Q14. Définir une fonction `type_check : expr -> expr`. Cette fonction sera l'identité[2] pour toute expression correctement typée. Pour les expressions non typée, on renverra une erreur.

[2] On rappelle $\mathrm{id}_A : x \mapsto x$ est la fonction identité de A.

V. Environnement de développement

Ce TP utilise `dune`, un environnement de développement pour OCaml. On peut lancer `utop` via la commande `dune utop`. Une fois ouvert, il suffira d'exécuter `open Interpreter.Main;;` pour importer le fichier `main.ml`. Si la commande `dune` n'est pas reconnue, on pourra l'installer via `opam` (le gestionnaire de paquets d'OCaml) :

```
opam install dune
```

En cas de difficultés d'installation, consultez le site https://dune.build/. Si ces difficultés persistent, prévenez et un *Repl.it* sera mis en ligne.