Université Bordeaux **Licence Informatique 2024-2025**
4TIN603U **Projet de Compilation : interpréteur mini-ml**

Modalités pour rendre le travail

Ce projet est à réaliser à 2 personnes et à remettre le 4 mai 2025 sur le site Moodle du cours. Le dépôt devra contenir (où "nom" est remplacé par le nom d'un ou de plusieurs membres du groupe):

- Un fichier `<vos_noms>.pdf` qui ne contiendra aucun code mais qui précisera:
  - Qui sont les auteurs du projet en détaillant qui a fait quoi.
  - Les réponses aux questions de chaque partie du projet.
  - Pour chaque partie du projet, les difficultés rencontrées s'il y en a, et qui fera un bilan de ce qui est fonctionnel et ce qui ne l'est pas.
- Un fichier `<vos_noms>.tar.gz` ou `<vos_noms>.tgz` qui contiendra le projet (incluant la partie extension telle que décrite précédemment).

Attention, le projet initial vous étant fourni compilant, un rendu qui ne compilerait pas aura automatiquement une note inférieure à la moyenne. Si vous n'arrivez pas à implémenter une fonctionnalité, on préfèrera une version compilable mais non fonctionnelle, avec une explication dans le rapport qu'un code approchant mais ne compilant pas. Rien ne vous empêche de mettre en commentaire des propositions ne compilant pas.

Dans la suite, on présente d'abord la structure du code de ce qui vous est fourni, puis on introduit pas à pas les caractéristiques du langage **Mini-ml** dans la partie où elles sont utiles – chaque partie correspondant à une partie de l'interpréteur. Le but de ce projet est de réaliser un interpréteur pour le langage **Mini-ml**. **Mini-ml** est un langage de programmation fonctionnel qui est un sous-ensemble de OCaml. À quelques détails près, un programme mini-ml peut être compilé par le compilateur OCaml, et votre interpréteur mini-ml pourra interpréter des codes OCaml restreints. L'annexe 6 récapitule la syntaxe et la sémantique du langage (cette dernière étant déjà implémentée pour vous), ainsi que le système de types.

Vous aurez à réaliser (pas forcément dans l'ordre):

- un parseur de syntaxe étendue (vous avez un parseur de syntaxe restreinte fourni)
- un typeur naïf (avec polymorphisme faible)
- un typeur avec polymorphisme fort
- répondre à des questions de compréhension/analyses de ce qui est mis en œuvre dans ce projet.

1 Base de code et bureaucratie

Si vous travaillez chez vous, pensez à installer les paquets nécessaires pour le cours. Au CREMI, tout est installé correctement et vous n'aurez à faire que les manipulations habituelles (voir page du cours).

Le programme fourni contient :

- Un programme principal qui met en lien les différents éléments du compilateur. Il est dans le fichier `bin/main.ml` et produit l'exécutable `interpreter.out`. Ce fichier devrait être laissé tel quel[1].
- Un programme de visualisation du parseur dans le fichier `bin/visualiser.ml` qui produit l'exécutable `visualiser.out`. Il ne doit pas être modifié[2].
- Une définition du langage **Mini-ml** sous forme d'arbre de syntaxe abstraite. Cette définition peut être trouvée dans `language/ast.mli`. Elle n'est pas à modifier.
- Un interpréteur du langage **Mini-ml** (dans `language/interpreter.ml`). Il est fonctionnel pour la partie principale du projet.
- Une bibliothèque `Util` dans le dossier `util` qui contient deux modules utilitaires pour votre interpréteur.
- Un ensemble de programmes exemples dans le dossier `programs`. Ce dossier comprend quelques exemples corrects (dans `good_examples`) et quelques exemples avec des erreurs de sémantique (dans `semantic_incorrect`).
- Un fichier `language/Lexer.mll` qui contient le lexeur de **Mini-ml** (fourni).
- Plusieurs fichiers `.mly` qui contiennent les différents parseurs de **Mini-ml**. La partie 1 du projet consiste à implémenter le parseur complet dans `Parser.mly`.
- Un ensemble de fonctions utilitaires sur les types regroupées dans `language/type_system.mli` qui vous aideront pour la partie typeur du projet.
- Les fichiers `language/typer_naive.ml`, `language/typer_util.ml` et `language/typer.ml` qui sont à implémenter pour la partie typeur du projet.

Vous pouvez compiler le projet avec la commande `make`. Les exécutables fournis affichent un message d'aide si vous les lancez sans argument. Les fichiers d'explication de menhir sont également généré. Vous pouvez générer la documentation du code fourni avec `make doc`.

Une documentation vous est fournie pour les modules implémentés que vous avez à utiliser, et quelques commentaires d'aide sont placés dans les fichiers à implémenter.

`interpreter.out` parse le programme en entrée, vous l'affiche, tente de le typer, puis de l'exécuter. Vous pouvez contrôler son comportement via un ensemble d'option (voir message d'usage).

1.1 Barème indicatif

La partie parsing comptera pour environ 8/20. Dans la partie typeur, on aura 5-6 points sur le typeur naïf ; 3-4 points sur la résolution des contraintes et 2-3 points sur le typeur fortement polymorphe. Les éventuelles extensions seront hors-barème. Dans chaque partie, ne négligez pas les questions de commentaire/compréhension, qui représenteront environ la moitié de la note.

On évaluera plus favorablement un projet qui s'est concentré sur quelques-uns des points à traiter, mais les a correctement réalisé et expliqué, qu'un projet qui s'est dispersé et a tout mal fait. C'est d'ailleurs pour s'adapter à ces différence que le barème ci-dessus est indicatif.

[1] du moins, si vous n'ajoutez pas d'extension

[2] sauf si vous voulez rajouter d'autres parseurs

2 Le langage Mini-ml

Mini-ml est un mini-langage fonctionnel (non-pur mais uniquement à cause de l'affichage) qui est presque un sous-ensemble strict de OCaml : à part la fonction d'affichage générique (absente en OCaml), et le fait que certaines fonctions de bases sont dans des modules en OCaml, la syntaxe et la sémantique d'un programme de mini-ml seront identiques à ce qu'elles sont en OCaml.

Mini-ml permet de manipuler des entiers, des booléens, des strings, la valeur `unit` (représentant l'absence de donnée), des fonctions et des listes (homogènes). Il dispose d'un ensemble de fonctions built-in permettant de manipuler les données susmentionnées (fonctions arithmétiques et booléennes, constructeurs et accesseurs de liste, fonction d'affichage). Il ne distingue pas les opérateurs des autres fonctions. Sa seule structure de contrôle est le if then else. C'est un langage statiquement (et strictement) typé, mais ce typage peut être réalisé de manière implicite. Il permet de définir des fonctions polymorphes.

L'annexe 6 contient l'ensemble de la syntaxe, sémantique et système de type du langage.

Le langage contient les constructeurs suivants :

- Des constantes (entières (`int`), booléennes (`bool`), chaînes de caractères (`string`), la liste vide `[]`, et l'action qui ne fait rien `()`).
- Des fonctions dites «built-in» qui peuvent être vues comme les fonctions de bases du langages. Elles incluent les opérations arithmétiques, booléennes, de comparaison, de manipulation de chaîne de caractères et de listes standards, ainsi qu'une fonction d'affichage générique. Ce sont des fonctions à un ou deux paramètres (en fonction du cas).
- Des noms («variables»), qui sont n'importe quelle chaîne de caractères alphanumériques commençant par une lettre minuscule.
- L'application d'une fonction à un argument. Ce constructeur suffit pour plusieurs arguments : si on veut passer deux arguments (mettons 1 et 2) à la fonction d'addition d'entiers, il suffit de l'utiliser deux fois :

  ```
  App(App(Cst_func(Add),Cst_i(1)),Cst_i(2))
  ```

  La stratégie de passage d'argument est le passage par valeur. L'argument est évalué avant la fonction, ce qui induit un ordre d'évaluation des arguments de droite à gauche s'il y en a plusieurs.
- Un if then else [3]
- Le constructeur de «liaison» `let` (récursif ou pas) qui permet de lier la valeur d'une expression à un nom (une «variable»). Le `rec` permet de définir des fonctions récursives (et donc au langage d'être aussi expressif qu'un langage impératif).
- Le constructeur de fonction qui permet de définir une fonction prenant un paramètre (pour des fonctions à plusieurs paramètres, il suffit d'utiliser ce constructeur plusieurs fois).

[3] En réalité, on aurait pu le remplacer lui aussi par une fonction, mais ça aurait causé quelques subtilités sur les effets de bords : contrairement à l'application de fonction, le if then else est paresseux (il n'évalue que la branche prise) – si ce point vous intéresse, parlez-m'en, je serais ravi de vous illustrer ce point.

- Le constructeur de séquencement qui permet d'ignorer la valeur d'une expression, puis de calculer la suivante. Dans ce langage, ce n'est utile que pour réaliser des affichages (puisque ce sont les seuls effets de bords possibles).

3 Parsing

Dans cette section, on va s'intéresser à réaliser plusieurs parseurs pour diverses syntaxes du langage. Dans `language/Parser_simple.mly`, vous avez un parseur qui implémente déjà une syntaxe minimaliste pour mini-ml (cf Annexe 6 pour le détail) qui correspond à expliciter tous les constructeurs du langage sans aucun sucre syntaxique.

Vous avez un fichier intitulé `Parser_sandbox.mly` qui est initialement identique à `Parser_calc.mly`. Vous pouvez l'utiliser pour tester des trucs sans détruire le reste du projet. Si vous le souhaitez, vous pouvez même rajouter d'autres fichier de parseurs. Mais pour cela, il vous faudra modifier le fichier dune du dossier `language`, ainsi que les deux exécutables dans le dossier `bin`. Essentiellement il vous faudra y dupliquer et adapter tout ce qui concerne le parseur sandbox.

3.1 Parseur simple : compréhension

On va d'abord analyser quelques points intéressants de ce parseur, puis on en réalisera des extensions (qui permettent d'ajouter du sucre, et de typer explicitement le langage).

On va se servir d'une version simplifiée du parseur qui se trouve dans `language/Parser_calc.mly`, qui contient uniquement les constantes entières, les variables, la définition de fonction, l'application de fonction, et deux fonctions de base (l'addition et la soustraction unaire), qui permet de se concentrer sur le phénomène que l'on souhaite observer.

La grammaire la plus naturelle pour ce sous-langage est la suivante :

```
expr:
| FUN x = ID ARROW e = expr { Fun(x,e,Annotation.create $loc)
    }
| e1 = expr e2 = expr { App(e1,e2,Annotation.create $loc) }
| i = INT { Cst_i(i,Annotation.create $loc) }
| f = built_in { Cst_func(f,Annotation.create $loc) }
| x = ID { Var(x,Annotation.create $loc) }
| L_PAR e = expr R_PAR { e }
```

Vous noterez que ce n'est pas ce qu'il y a dans `Parser_calc.mly`.

1. Donnez l'automate LR0 associé à la grammaire ci-dessus (on considèrera pour cette question que `built_in`, `INT` et `ID` sont un seul terminal (qu'on notera `T`), car le phénomène à observer ne les concerne pas vraiment). Vous pouvez évidemment vous servir de menhir pour vérifier que vous avez le bon automate. La grammaire simplifiée est donc la suivante (il faut évidemment ajouter le non-terminal initial technique $expr'$ pour avoir un résultat similaire à menhir):

$$expr \rightarrow \texttt{FUN T ARROW } expr$$
$$expr \rightarrow expr\ expr$$
$$expr \rightarrow \texttt{T}$$
$$expr \rightarrow \texttt{(}\ expr\ \texttt{)}$$

2. Où sont les conflits sur cet automate, si on le considère comme un automate SLR ?

3. Donnez un exemple de séquence de tokens sur laquelle deux arbres de dérivations sont possibles avec cet automate. Qu'en déduisez-vous sur cette grammaire naturelle ?

4. Quel est le choix fait par le parseur implémenté dans `Parser_calc.mly` sur votre séquence de tokens ?

5. Quelles priorités peut-on ajouter à la grammaire ci-dessus pour retrouver le comportement de `Parser_calc.mly` (utilisez menhir pour vérifier que vous avez la bonne réponse).

6. Si on ajoute toutes les fonctions `built_in` dans le parseur, qu'est-ce que cela donne en terme du nombre de priorités à écrire.

7. À votre avis, pourquoi donc utilisons-nous plutôt des non-terminaux distincts dans ce cas plutôt que des priorités ?

3.2 Syntaxe étendue

Vous allez maintenant réaliser un parseur qui accepte la syntaxe étendue de mini-ml. Réalisez ce parseur dans `language/Parser.mly` (qui contient pour le moment la même chose que `Parser_simple.mly`).

Il faut y ajouter le sucre suivant :

- la notation infixe des opérateurs binaires : `e1 + e2` est du sucre pour la syntaxe standard `(+) e1 e2`. On utilisera les priorités usuelles des opérateurs binaires (cf plus bas).
- la notation du `let` avec arguments : `let nom a1 a2 = e` est du sucre pour `let nom = fun a1 -> fun a2 -> e`.
- le - unaire : `- 4` est du sucre pour `neg 4`.
- les listes constantes : `[1;2;3]` est du sucre pour `(::) 1 ((::)2 ((::)3 []))`.

Vous expliquerez dans le document de réponses pour chaque point la solution que vous avez adopté pour le traiter, ainsi que les éventuelles difficultés rencontrées (i.e., qu'est-ce qui ne fonctionne pas correctement).

Si vous n'arrivez pas à traiter un des points, fournissez un exemple de programme étant correctement parsé avec notre version et pas avec la vôtre.

Priorité des opérateurs Les opérateurs les plus prioritaires sont les opérateurs unaires (et ne sont pas associatifs).

Ensuite viennent les opérateurs arithmétiques, qui sont associatifs à gauches, et parmi lesquels les opérations multiplicatives sont prioritaires sur les additives.

Puis on a les opérateurs de listes, qui eux sont associatifs à droite.

Ensuite on a la concaténation de chaînes de caractères (associative à gauche).

Puis viennent les opérateurs de comparaisons, également associatifs à gauche.

Enfin, les opérateurs les moins prioritaires sont les opérateurs booléens (tous aussi associatifs à gauche).

Évidemment, nous n'avons listé que les priorités des opérateurs. Vous expliquerez comment elles se comparent aux priorités déjà présentes dans le parseur simple et pourquoi c'est cette solution qui est la bonne.

4 Typage

Dans cette section, on va implémenter le typeur de mini-ml, qui est en réalité l'algorithme de typage de ml[4]. À la fin de la section, vos programmes devraient être typés de la même manière par votre typeur et par OCaml !

On va procéder par étape : d'abord en faisant un premier algorithme qui explore l'AST, type les constantes, associe à chaque nom un type universel «frais» et génère les contraintes de types impliquées par l'AST. Ensuite, on rajoutera la résolution des contraintes, sans se soucier du polymorphisme des let (on aura ce qu'on nomme un polymorphisme faible). Enfin, on rajoutera ce qu'il faut pour avoir un véritable polymorphisme.

Pour ce faire, les fonctions de `language/type_system.mli` sont là pour vous aider dans cette tâche : elle réalisent pour vous la plupart des petites tâches techniques et rébarbatives, vous permettant de vous concentrer sur le cœur de l'algorithme de typage.

Le code présent dans `bin/main.ml` appelle les fonctions à réaliser ici en simplifiant pour vous une partie du travail : pour les deux premières partie, il appelle vos fonctions dans le bon ordre directement sur les expressions (vous n'avez donc pas à traiter les requêtes). Pour la seconde partie, il appellera uniquement la fonction `type_expr`, en remplaçant chaque requête `let f = e` par l'expression `let f = e in f` (qui se type évidemment de la même manière).

Les types valides en mini-ml sont les suivants:

- Les types constants (`TInt`, `TBool`, `TString` et `TUnit`)
- Les types universels `TUniv(n)`. Ils sont indexés par un entiers (pour différencier les différents types présents) et peuvent être remplacés par n'importe quel type concret. `TUniv(0)` sera affiché comme `'a` ; `TUniv(1)` comme `'b`, etc. À partir de `TUniv(26)`, on aura une notation de la forme `'_0`, etc.
- Les types fonctionnels `TFunc(generic,argument,result)` qui représentent un type de fonction de la forme `argument -> result`. La liste `generic` contient les indexes des types universels qui sont "génériques", c'est-à-dire locaux à la fonction et qui peuvent changer à chaque utilisation de la fonction.
- Les types listes `TList(generic,content)` qui représentent un type list de la forme `content list`. La list `generic` a le même rôle que dans le cas précédent.

La liste `generic` a pour rôle de permettre un vrai polymorphisme générique en «gardant» les variables universelles pour éviter que la première utilisation ne fixe son type définitivement.

Pour illustrer le problème, le code suivant :

```
let f = fun x -> x
let a = f 1
let b = f "coucou"
```

[4]Évidemment restreint à **Mini-ml**. En OCaml, il y a beaucoup plus de cas.

ne typera qu'avec un typage réellement polymorphe. Le premier let fixera le type de `f` comme étant `'a -> 'a`. Si on a un typage polymorphe faible (ce qui sera le cas à la seconde sous-section), alors dans la définition de `a`, on déterminera que `'a` doit être un `int`, et donc lors du typage de `b`, on aura une erreur (car `int` et `string` sont incompatibles). Au contraire, avec le polymorphisme fort, on aura deux instances différentes à chaque utilisation de `'a` qui pourront donc être différentes.

On ne se servira de ce concept que au niveau des `let`, et uniquement dans le dernier exercice de cette section.

4.1 Typage naïf et génération des contraintes

Le travail de cette section est d'implémenter le fichier `language/Typer_naive.ml` qui se contente de typer chaque sous-expression avec un type frais, et qui collecte les contraintes de type liées à l'AST. On ne se préoccupe pas de les résoudre pour le moment.

À ce stade, ce qu'on implémente ressemble donc beaucoup à un interpréteur (mais où au lieu d'exécuter le programme, on calcule son type et les contraintes associées).

Votre fonction `type_expr` doit explorer récursivement l'arbre et pour chaque nœud:

- Lui associer un type dans son annotation
- renvoyer le type en question et les éventuelles contraintes de type de l'AST.

Un *contrainte* est un couple de type qui représente le fait que les deux types le contenant doivent être égaux.

Par exemple `(TUniv(0),int)` représentera une contrainte disant que `'a = int` (ce qui est possible), et `TFunc([],TUniv(0),TUniv(1)),TInt` représentera que `('a -> 'b)= int` (ce qui est évidemment impossible).

Le principe est pour le reste similaire au typeur du langage du cours : lors des introductions de variables (qui sont ici les `let rec` et les fonctions), on introduit le nom de variable dans l'environnement (ici, avec un type universel frais, qui est fourni par le compteur en argument). Pour les `let n = e1 in e2` (récursif ou non), il faut associer `n` au type de `e1` lors de l'évaluation de `e2`. Les constantes reçoivent le type correspondant (attention, chaque fonction `built_in` a un type qu'il vous faudra définir explicitement dans la fonction dédiée). Les autres constructions inductives calculent leur type en fonction de leurs sous-expressions et introduisent des contraintes sur les types de leurs sous-expressions (la différence étant que dans le langage du cours, on pouvait résoudre immédiatement ces contraintes, et qu'ici, on se contente de les renvoyer).

À faire :

- Implémentez la fonction `type_of_built_in` du fichier `typer_util.ml` (attention à correctement attribuer les types universels génériques).
- Implémentez la fonction `type_expr` du fichier `typer_naive.ml`.
- Donnez 3 (ou plus) petits programmes mini-ml qui illustrent le fonctionnement de votre typeur naïf (toutes les sous-expressions doivent être représentées). Vous placerez ces programmes dans `examples/answers`, et expliquerez ce qu'ils illustrent (en commentaire de ces programmes, et dans votre rapport).
- Choisissez l'un de ces programmes, et illustrez le fonctionnement du typeur sur celui-ci sur papier. Dessinez l'arbre, et pour chaque nœud, donnez son type et dites quelles contraintes ce nœud introduit (évidemment, choisissez un exemples où il y a des contraintes).

- Si vous avez une différence entre votre typeur et le nôtre sur l'un de vos programmes ou l'un des nôtres, décrivez cette différence, et expliquez d'où vous pensez qu'elle vient.

Rappel: vous avez à votre disposition un exécutable compilé avec la solution du projet, qui affiche à chaque pas du typeur où il en est. Vous pourrez donc comparer votre résultat au notre dès cette phase.

Note sur les environements Pour vous simplifier la vie, et parce que le langage qu'on a ici est tout de même assez différent de celui du cours, on a modifié le module `Util.Environment` par rapport au langage du cours. Notamment, on a caché entièrement les références ici (aucun aliasing n'a de sens dans le typeur), et on a ajouté une fonction `remove`. Étant donné que `Mini-ml` force une gestion de portée stricte, il faut faire attention à gérer cette portée correctement dans le typage. Il faut donc bien penser à *retirer* de l'environnement un nom dont la portée se termine. Un exemple grossier, le programme suivant ne doit pas typer (parce que `x` n'existe pas en dehors de la déclaration de `a`):

```
let a = let x = 4 in
        x
let b = x
```

Pour cela, il vous faudra donc bien appeler la fonction `remove` dès qu'un nom sort du scope (la portée).

En réalité, grâce à cela, les environnements fonctionnent naturellement comme des piles, et il sera même donc possible d'avoir du masquage de nom qui fonctionne comme en Ocaml : le programme suivant est bel est bien correct :

```
let a = let x = true in
        if x then
          let x = 4 in x
        else
          let x = 12 in x
```

4.2 Résolution des contraintes et polymorphisme faible

Dans cette section, on va résoudre les contraintes introduites à la section précédente.

Le but de cette section est d'implémenter la fonction `solve_constaints` dont le but est de résoudre un système de contraintes et de calculer une *substitution de types* qui en découle.

Une substitution de types sera une liste de réécriture qui remplace (dans l'ordre) des types universels par d'autres types.

Par exemple, une substitution peut être `[(0,TInt); (1,TFunc([],TInt,TBool))]` qui remplace le type `'a` par `int`, puis le type `'b` par `int -> bool`, et donc réécrira le type `'a -> 'b list` en `int -> (int -> bool)list`.

L'ordre est important : la substitution `[(0,TUniv(1));(1,TInt)]` n'aura pas le même effet que `[(1,TInt);(0,TUniv(1))]` : dans le premier cas, `'a` deviendra `int`, dans le second, il deviendra `'b`.

La fonction `solve_constraints` prend en arguments une liste de contraintes, et renvoie la substitution associée.

Le principe est le suivant: la liste vide donne la substitution vide ; sinon, on regarde la première contrainte (t_1, t_2) de la liste (et on nomme r le reste de la liste) et :

- si t_1 est un type universel `TUniv(n)`, on définit la substitution (n, t_2), et on applique cette substitution à toutes les contraintes restantes, puis on calcule la substitution s correspondante aux contraintes restantes r. Le résultat est alors $(n, t_2) :: s$. Attention, si `TUniv(n)` apparaît dans `t_2`, on doit renvoyer une erreur (la résolution de la contrainte doit faire disparaître `TUniv(n)`). Dans le cas où t_1 et t_2 sont tous les deux des types universels, on réécrit celui d'indice le plus grand en celui d'indice le plus petit. Évidemment, le cas où t_2 est un type universel mais pas t_1, on fait la même chose en échangeant les rôles.

- si t_1 et t_2 sont tous les deux de la forme `TList(l1,a1)` et `TList(l2,a2)`, alors le résultat est celui de la fonction sur `(a1,a2)::r` (i.e., les éléments des listes doivent être de même types).

- si t_1 et t_2 sont tous les deux de la forme `TFunc(l1,a1,r1)` et `TFunc(l2,a2,r2)`, alors le résultat est celui de la fonction sur `(a1,a2)::(r1,r2)::r` (i.e., les arguments et résultats des deux fonctions doivent être de même type).

- dans tous les autres cas, si t_1 et t_2 sont différent, on renvoie une erreur (on a une contrainte non-satisfaisable). S'ils sont égaux, alors cette contrainte est trivialement vraie, et la substitution est donc celle correspondant au reste de la liste r.

Un point à noter : en fait, sur les fonctions built-in, on a déjà la possibilité à ce niveau de les utiliser avec des types différents (ce qui ne serait pas possible avec du polymorphisme faible strict). C'est parce qu'en fait, let fonctions de substitutions qui vous sont fournies ne substituent pas les types génériques (et ça induirait du travail supplémentaire à la section suivante si on voulait que ce ne soit pas le cas – et puis sinon, on forcerait toutes les fonctions built-in à utiliser le même type, ce qui est tout de même très restrictif).

À faire :

- Implémenter la fonction `solve_constraints` du fichier `typer_util.ml`.

- Dans les exemples de la question précédente, lesquels sont typés correctement, lesquels ne le sont pas ? Expliquez pourquoi.

- Observez que l'exemple donné en début de chapitre n'est pas typé. Donnez les parties typées, et expliquez où se situe l'erreur.

- Proposez un autre programme qui n'est pas correctement typé alors que OCaml le type, et expliquez pourquoi. Vous le placerez dans `examples/answers`.

4.3 Typage fortement polymorphe

On va maintenant terminer le typage en ajoutant le polymorphisme fort. Pour cela, on a déjà quasiment tous les éléments avec les deux sections précédentes.

On rappelle que beaucoup de petites tâches techniques sont implémentées par les fonctions de `type_system.mli`. Lisez-en bien la documentation.

L'algorithme de typage est donc presque exactement le même que le typeur naïf, aux différences près listées dans la suite de cette section.

Les modifications à réaliser sont uniquement localisées sur les expressions `let` et les applications de fonction (et ne constituent que très peu de lignes de code par rapport à ce que vous avez déjà écrit) :

- Au lieu de résoudre les contraintes une fois le terme entier calculé, on résout les contraintes ≪internes≫ à chaque expressions `let`. Sur l'expression `let x = e1 in e2`, on résout d'abord les contraintes ≪internes≫ à `e1`, on applique la substitution obtenue sur `e1` et sur l'environnement, puis on généralise le type de `e1` (pour capturer les variables universelles locales, et donc à partir de la même valeur que celle qui détermine les contraintes internes). Attention, à chaque fois que des contraintes sont résolues, il faut appliquer la substitution en résultant sur l'expression correspondante (ici, `e1`), et sur l'environnement.

- Les contraintes ≪internes≫ à `e1` sont celles qui ne contiennent pas de type universel existant ailleurs que dans `e1`. Pour les détecter, on se servira de la valeur du compteur avant de type `e1` (les types universels n'existant que dans `e1` auront un indice supérieur). Une fonction vous est fournie pour couper votre liste de contraintes entre contraintes internes et externes.

- Généraliser un type `t` revient à considérer que, si ce type est une fonction ou une liste, alors tous les types universels apparaissant uniquement dedans sont génériques. Concrètement, on les place simplement dans la liste `generics` du type en question. Par exemple, si on a un type `TFunc([],TUniv(0), TUniv(1))` et qu'on le généralise à partir du type d'indice 1, on le remplacera par `TFunc([1],TUniv(0),TUniv(1))`. Encore une fois, on vous fournit une fonction vous aidant à faire cela.

- À chaque terme d'appel de fonction `App(f1,f2)`, avant de déterminer les contraintes applicables, on *instancie* les types de `f1` et `f2`, c'est-à-dire que pour chaque type présent dans la liste de types générique, on remplace le type par un type universel non encore utilisé. Ainsi, si on a `f1` de type `TFun([1],TUniv(0),TUniv(1))` et `f2` de type `TList([0],TUniv(0))`, et que le compteur vaut actuellement 12, on remplacera les types par respectivement `TFun([],TUniv(0),TUniv(12))` et `TList([],TUniv(13))`, et on génèrera donc la contrainte `(TUniv(12),TList([],TUniv(13)))`. Pour les types qui ne sont ni des fonctions ni des listes, il n'y a pas d'instanciation (ou plutôt, l'instanciation ne modifie pas le type).

Une fois cela fait, le typeur supportera le polymorphisme fort.
Dans cette partie, on implémente le fichier `language/typer.ml`.
À faire :

- Implémentez la fonction `instantiate`, du fichier `typer_util` qui réalise l'instantiation d'un type décrite plus haut.

- Implémentez la fonction `type_expr` du fichier `typer.ml`, qui adapte votre fonction de la première sous-section en y ajoutant les modifications décrites plus haut.

- Observez que l'exemple décrit en début de section est maintenant correctement typé.

- Illustrez le fonctionnement du typage polymorphe en fournissant deux exemples supplémentaires qui typent différemment avec les deux algorithmes de typage. Choisissez-en un dont vous expliquerez soigneusement où se situe la différence (i.e., décrivez l'application des deux algorithmes à cet exemple). Vous les placerez dans `examples/answers`.

- Si vous avez des différence entre votre implémentation et le comportement de l'outil qui vous est fourni, décrivez-les, et dites d'où vous pensez qu'elles viennent.

5 Extensions

Si vous avez tous fini et souhaitez aller plus loin, parlez-en à votre chargé de TD pour déterminer une extension raisonnable parmi vos idées, et sur la marche à suivre pour le faire proprement. Dans tous les cas, créez une archive avec votre projet sans extension – ce qui en permettra l'évaluation sans que vous ne cassiez quoi que ce soit.

Le principe général sera tout de même de ne pas écraser les codes produits dans les sections précédentes, donc cela vous amènera très probablement à modifier les fichiers dune et les exécutables du projet.

Une idée assez raisonnable est la suivante : en OCaml, il est possible de typer explicitement les variable (dans les déclarations de fonctions ou dans les let). Cela se fait avec les syntaxes suivantes :

```
let a : int = 4
let f : int -> int = fun (n : int) -> n + 1
let f (n : int) : int = n + 1
```

Cette syntaxe permet à l'utilisateur de rajouter des types explicites et d'avoir une erreur si ce typage n'est pas cohérent avec le programme.

Pour étendre le projet, vous pouvez :

- créer un parseur supportant cette syntaxe (les deux premières sont plus facile, la troisième plus techniques), et qui en présence d'un tel type l'ajoutent dans l'annotation.
- adapter les deux typeur pour lorsqu'un type est déjà présent dans l'annotation, en tenir compte dans les contraintes.

Évidemment d'autres extensions sont possibles. Si vous avez une idée, parlez-en avec votre chargé de TD qui pourra vous donner son avis sur la faisabilité. Également, il se pourra que vos idées impliquent de modifier les fichiers de configuration du projet ou le fichier `bin/main.ml`. Ce n'est pas un problème, mais n'hésitez pas à demander de l'aide sur ce point si besoin.

6 Annexe : Références du langages Mini-ml

6.1 Éléments lexicaux

Mots-clé Les mots-clé de **Mini-ml** sont les suivants:

`if`, `then`, `else`, `+`, `-`, `*`, `/`, `%`, `mod`, `||`, `not`, `=`, `<>`, `<=`, `>=`, `<`, `>` `(`, `)`, `;`, `true`, `false`, `print`, `let`, `rec`, `fun`, `->`, `^`, `::`, `hd`, `tl`, `in`, `neg`, `[`, `]`, `@`

Dans la définition de la syntaxe du langage (section suivante), on les utilisera tels quels.

Identificateurs Les identificateurs commencent par une lettre minuscule non accentuée éventuellement suivie d'une série de lettres alphanumériques et d'underscore. Exemples d'identificateurs: `x`, `i`, `premier_Nombre`, `value12`

Dans la définition de la syntaxe du langage, on utilisera `{id}` pour désigner un tel identificateur.

Entiers Les entiers sont représentés en décimal (une suite non-vide de chiffres entre 0 et 9).

Dans la définition de la syntaxe du langage, on utilisera `{int}` pour désigner l'entier obtenu.

Chaînes de caractères Les chaînes de caractères sont une séquence de n'importe quelle suite de caractères entourée de guillemets `"`. Elles peuvent contenir des caractères échappés par un \.

Dans la suite, on les notera `{str}`

Commentaires Les commentaires sont n'importe quelle séquence de caractère commençant par `(*` et terminant par `*)`

6.2 Syntaxe de Mini-ml

On donne ici la grammaire du langage dans sa syntaxe simple.

On a les fonctions built-in suivantes :

- `unop` : `neg`, `not`, `hd`, `tl`, `print`
- `binop` : `+`, `-`, `*`, `/`, `mod`, `&&`, `||`, `=`, `<>`, `<=`, `>=`, `<`, `>`

Les *expressions* (ci-après `expr`) et leurs AST (sans tenir compte des annotations) correspondants sont les suivants :

expr	AST
{int}	Cst_i({int})
true	Cst_b(true)
false	Cst_b(false)
{str}	Cst_str({str})
(binop)	Cst_func(binop)
unop	Cst_func(unop)
[]	Nil
()	Unit
{id}	Var({id})
if expr1 then expr2 else expr3	IfThenElse(expr1,expr2,expr3)
expr1 expr2	App(expr1,expr2)
let {id} = expr1 in expr2	Let(false,{id},expr1,expr2)
let rec {id} = expr1 in expr2	Let(true,{id},expr1,expr2)
fun {id} -> expr	Fun({id},expr)
expr1 ; expr2	Ignore(expr1,expr2)
(expr)	expr

On fera attention à deux points :

- les opérations binaires *doivent* être entourées de parenthèses pour être appliquées comme des fonctions (attention à détacher les parenthèses de la multiplication pour ne pas être confondu avec des commentaires).
- Telle quelle, la grammaire est ambigüe à cause de l'application. Voir parseur fourni pour une résolution de ce point.

Le sucre syntaxique est décrit en section 3.2.

6.3 Sémantique du langage

La sémantique des expressions **Mini-ml** est la suivante :

expr	valeur : $[expr]_\eta$
`Cst_i(i)`	i
`Cst_b(b)`	b
`Cst_str(str)`	str
`Cst_func(f)`	$[\![f]\!]$
`Nil`	[]
`Unit`	()
`Var(id)`	$\eta(id)$
`IfThenElse(expr1,expr2,expr3)`	$[expr2]_\eta$ si $[expr1]_\eta = true$ $[expr3]_\eta$ sinon.
`App(expr1,expr2)`	$[body]_{\eta[x \leftarrow [expr2]_\eta]}$ si $[expr1]_\eta$ =`Func(x,body)`
`Let(b,id,expr1,expr2)`	$[expr2]_{\eta[id \leftarrow [expr1]_\eta]}$
`Fun(id,expr)`	`Func(id,expr)`
`Ignore(expr1,expr2)`	$[expr2]_\eta$ (après avoir évalué expr1)

Deux choses à noter :

- Le `Let` se comporte de la même manière qu'il soit récursif ou non (la seule implication du rec est sur le typage).
- Le `Ignore` ignore le résultat de la première expression, mais il faut tout de même l'évaluer pour ses éventuels effets de bord (ici de l'affichage).

On laisse la sémantique des fonctions built-in au lecteur. Elles sont toutes des fonctions classiques de langage de programmation (et on renvoie à l'implémentation dans `language/interpreter.ml`). Le seul point à noter : les comparaisons sont génériques, mais homogènes, et réalisent une comparaison *structurelle* des données comparées. Elles ne sont *pas* implémentées pour les valeurs fonctionnelles et renverront une erreur dans ce cas.

6.4 Typage de Mini-ml

Le type des expressions, ainsi que les contraintes correspondantes sont données ci-après. Dans le tableau, les types génériques notés `'a`, `'b`, etc, sont systématiquement des types *frais* (i.e., non-utilisés ailleurs). Dans la colonne contraintes, seules sont précisées les contraintes venant du nœud décrit. Il faut évidemment ajouter à celles-ci les contraintes générées par leurs sous-expressions.

expr	type(expr,η)	contraintes
`Cst_i(i)`	int	
`Cst_b(b)`	bool	
`Cst_str(str)`	string	
`Cst_func(f)`	le type de f	
`Nil`	'a list	
`Unit`	unit	
`Var(id)`	$\eta(id)$	
`IfThenElse(expr1,expr2,expr3)`	type(expr2,η)	type(expr1,η) = bool type(expr2,η) = type(expr3,η)
`App(expr1,expr2)`	`'b`	type(expr1,η) = `'a -> 'b` type(expr2,η) = `'a`
`Let(false,id,expr1,expr2)`	type(expr2,$\eta[id \leftarrow$t1$]$) où t1 = type(expr1,η)	
`Let(true,id,expr1,expr2)`	type(expr2,$\eta[id \leftarrow$t1$]$) où t1 = type(expr1,$\eta[id \leftarrow$ `'a`$]$)	type(expr1,$\eta[id \leftarrow$ `'a`$]$) = `'a`
`Fun(id,expr)`	`'a ->` type(expr,$\eta[id \leftarrow$ `'a`$]$)	
`Ignore(expr1,expr2)`	type(expr2,η)	

Pour le type des fonctions built-in, on renvoie le lecteur vers la documentation du module `Ast`.

Pour avoir le typage fortement polymorphe, on renvoie le lecteur à la section 4.3.