Projet : Compilateur de Mini-ML vers Java

1 Contexte

1.1 Motivation

Le but de ce projet est de développer un compilateur qui traduit des programmes de Caml vers Java.

Le langage d'entrée est un sous-ensemble réaliste, mais restreint, du langage Caml. Les programmes acceptés par votre compilateur le seront aussi par le compilateur / évaluateur de Caml (mais l'inverse n'est pas vrai). Ceci permet notamment d'utiliser Caml pour vérifier la syntaxe et le typage de vos programmes.

Votre compilateur produira un fichier en Java qui contient une séquence d'instruction de la machine abstraite de Caml (la CAM). Ce fichier sera compilé avec d'autres fichiers qui simulent le fonctionnement de la CAM.

Les étapes essentielles du projet sont :

— Mise à jour et extension de l'analyse lexicale et syntaxique du langage Caml, à l'aide des outils *ocamllex* et *ocamlyacc* (sect. 2.1).
— Écriture d'un simulateur de la CAM en Caml (sect. 2.2).
— Écriture d'un simulateur de la CAM en Java (sect. 2.3).
— Écriture du générateur de code de Caml vers Java (sect. 2.4).

Ces étapes concernent tout d'abord un fragment fonctionnel pur. Dans une deuxième phase, nous rajoutons des fonctions récursives, ce qui nécessitera d'itérer sur les étapes précédentes (sect. 2.5). Vous pourrez ensuite tester vos fonctions sur des fonctions "realistes" (sect. 2.6).

1.2 Travail demandé

Structure du projet : Le projet est à rendre jusqu'au **10 mai 2019 à 23h** au plus tard.

Format des fichiers à rendre : Vous avez deux possibilités :

1. Vous pouvez déposer une archive (`tarfile` ou `zip`, pas de format `rar` ou autres formats propriétaires) sur Moodle, *avant la date limite*, avec le contenu indiqué en bas.
2. Pour vous inciter à travailler avec des systèmes de gestion de version, vous pouvez aussi nous communiquer les coordonnées d'un dépôt (seuls formats admis : `git` ou Mercurial) où nous pouvons récupérer votre code. L'adresse du dépôt doit être envoyée à martin.strecker@irit.fr avant la date limite, et le dépôt doit rester stable pendant au moins 24h après la date limite, pour nous permettre de récupérer le contenu. Évidemment, le dépôt doit être accessible en mode lecture. Si vous avez des doutes, envoyez l'adresse du dépôt au moins 24h avant la date limite pour permettre des tests. Nous nous engageons à évaluer le code uniquement après la date limite.

Le travail doit être un travail individuel qui ne doit pas être la copie (partielle ou intégrale) du code d'un collègue. Bien sûr, on ne vous interdit pas un travail collaboratif, et si vous avez travaillé avec des collègues et que vous supposez une trop grande similarité du code, indiquez vos collaborateurs dans le `README` à joindre au projet.

Fichiers à rendre : Déposez une archive (`tarfile` ou `zip`, pas de format `rar` ou autres formats propriétaires) sur Moodle, *avant la date limite* ; ou alternativement un dépôt Git ou Mercurial. L'ensemble des fichiers doit comporter :

- un répertoire avec des fichiers source (en Caml), qui doivent compiler par un simple `make` à la console ;
- un répertoire avec des fichiers Java qui implantent le fonctionnement de la CAM ;
- un répertoire avec des fichiers tests (en Caml) qui doivent se compiler et s'exécuter avec le compilateur que vous avez écrit ;
- un fichier `README` avec des instructions comment compiler et exécuter vos tests. Ce fichier doit en plus contenir un résumé du travail effectué, éventuellement des difficultés rencontrées, et aussi des tests que vous avez faits, éventuellement avec vos observations (par exemple sur la taille du code, son optimalité etc.).

1.3 Fichiers fournis

L'archive de *fichiers Caml* fournie contient les fichiers suivants :

- `lexer.mll` et `parser.mly` : Le lexer/parser pour le langage source.
- `interf.ml` : une interface avec le parser.
- `comp.ml` : le fichier principal, contenant la fonction `main` qui permet d'exécuter le code de la console, voir annexe A.
- `miniml.ml`, qui contient les types du langage source. Vous avez le droit de modifier ces types, mais il n'est pas nécessaire de le faire, et si vous le faites, il faut documenter et motiver ces modifications.
- `instrs.ml` : le type de donnés des instructions du langage cible (de la CAM). Nous recommandons de programmer toutes les fonctions Caml dans ce fichier ; alternativement, vous pouvez créer un nouveau fichier avec vos fonctions.
- `use.ml` : Permet de charger les fichiers compilés, dans l'interpréteur de Caml.
- un `Makefile` pour compiler le projet. Ne modifiez pas ce fichier sauf si vous savez exactement ce que vous faites.

L'archive de *fichiers Java* contient :

- `Main` : le fichier principal, à exécuter (après compilation) avec `java Main`
- `Config` : Gestion d'une configuration d'une CAM.
- `Instr` et sa sous-classe `Quote` : des instructions de la CAM, correspond au type `instr` du fichier `instrs.ml`, respectivement au constructeur `Quote` de ce type. Voir aussi sect. 2.3.
- `StackElem` : composants de la pile, correspond au type `stackelem` du fichier `instrs.ml`.
- `Value` et ses sous-classes `IntV` et `NullV` : des valeurs manipulées par la CAM, correspond au type `value` du fichier `instrs.ml`, respectivement aux constructeurs `IntV` et `NullV` de ce type.
- `LLE` : extension de la classe *LinkedList* qui permet de construire une liste de manière fonctionnelle (liste vide, `empty` ; et liste composée, `add_elem`) comme en Caml.
- `Gen` : classe contenant une liste d'instructions qui doit être exécutée par la CAM. Le fichier `Gen.java` est censé être généré par votre compilateur ; il contient la liste des instructions qui est le résultat de la compilation d'un programme source. Voir aussi sect. 2.4.

1.4 Ressources

- La description de la CAM (fragment fonctionnel pur) se trouve sur les transparents du cours, section "CAM".
- La fonction de traduction d'expressions Caml vers la CAM est décrite dans les transparents, section "Fragment fonctionnel vers CAM".
- La CAM pour les fonctions récursives et la traduction des fonctions récursives sont décrites dans les transparents, section "Appels récursifs".
- La CAM et la traduction pour le fragment fonctionnel suivent assez exactement l'article : Cousineau, Curien, Mauny : *The Categorical Abstract Machine*, Science of Computer Programming 8

(1987), pp. 173-202. Cet article est disponible sur Moodle. Par contre, le traitement des fonctions récursives diverge considérablement.

2 Description du travail demandé

2.1 Analyse lexicale et syntaxique

L'essentiel de la grammaire lexicale et syntaxique est fournie dans les fichiers `lexer.mll` respectivement `parser.mly`. Vous pouvez appeler le parser comme décrit en sect. A.2 pour obtenir un arbre syntaxique (c'est à dire, une expression de type `prog`) à partir d'un programme Caml écrit sur un fichier.

Typiquement, vous n'aurez pas besoin de modifier l'analyse lexicale.

Modifications nécessaires : Les extensions suivantes sont nécessaires dans `parser.mly`, pour avoir une gamme suffisamment large d'opérations pour écrire des programmes réalistes :

— Rajout de règles de grammaire pour des expressions avec opérateurs additifs (addition, soustraction), de comparaison et d'expressions booléennes.

— Rajout d'une règle pour des paires.

Les règles pour les opérateurs doivent respecter leur priorité et associativité. Ainsi, les règles pour des opérateurs de comparaison doivent faire appel aux règles pour les opérateurs additifs, qui font appel aux opérateurs multiplicatifs.

Attention : les opérateurs booléens se traduisent directement en expressions conditionnelles, par exemple :

```
| compar_exp BLAND bool_exp
    Cond($1, $3, Bool(false))
```

Modifications facultatives : Mis à part cela, vous êtes libre à étendre la grammaire pour plus de comfort d'écriture de programmes. Quelques suggestions :

— Écriture de paramètres de fonctions de la forme `let rec f a1 a2 = ...`. Cette forme sera traduite directement en un arbre syntaxique de la forme `let rec f = fun a1 -> fun a2 -> ...`.

— Filtrage sur les paires, de la form `fun (x, y) -> ... x ... y`. La génération d'un arbre syntaxique, de la forme `fun p -> ... (fst p) ... (snd p)`, est plus difficile à effectuer.

Ces modifications sont un plus, mais ne sont pas exigées dans le cadre du projet.

2.2 Execution de la CAM en Caml

Le but de cette étape est d'écrire un simulateur de la CAM en Caml. Il est relativement facile à écrire parce que les règles de transition présentés en cours se transcrivent facilement en Caml, et parce qu'on peut facilement tracer son exécution.

Dans le fichier `instr.ml`, vous trouvez les définitions des types `instr`, `value`, `code` et `stackelem` présentés en cours.

Exercice 1 Définissez la fonction `exec` qui prend comme argument une configuration (un triplet (terme, code, pile)) et qui exécute un programme de la CAM selon les règles définies en cours. La fonction `exec` s'arrête si aucune règle n'est applicable.

Exercice 2 Tracez la fonction sur quelques exemples.

2.3 Execution de la CAM en Java

Nous proposons d'écrire des classes Java qui permettent d'exécuter la CAM sous Java. Vous avez une grande liberté de choix et pouvez concevoir d'autres structures que celles décrites en bas. Il faudra toutefois que votre code soit bien documenté et compréhensible. Nous décrivons dans la suite une structuration possible.

Nous proposons de créer les classes Java `Instr`, `Value` et `StackElem`, toutes les trois étant des sous-classes abstraites d'`Object` et qui correspondent aux types analogues sous Caml. En plus, nous aurons besoin d'une classe `Config` qui représente une configuration de la CAM. Il n'y aura pas de classe explicite pour le type `code` sous Caml. Nous proposons d'utiliser une classe de listes, par exemple `LinkedList<Instr>`.

Les sous-classes sont :

— `Instr` : les instructions, une classe par instruction. Les constructeurs des classes Java auront donc quasiment les mêmes arguments que les constructeurs Caml.

— `Value:` même remarque que pour `Instr` : une sous-classe par constructeur de valeur.

— `StackElem:` pareil, deux sous-classes `ValueSE` et `CodeSE`.

Ces classes ont des constructeurs et des méthodes `get` et `set` pour leurs champs privés. À part cela, les classes `Value` et `StackElem` n'ont pas de fonctionalité particulière.

La classe `Config` déclenche l'essentiel de la fonctionnalité de la CAM. Cette classe contient une méthode `exec_step` qui fait un seul pas de l'exécution de la CAM (correspondant à une seule transition $(t, c, st) \longrightarrow_c (t', c', st')$). Sur cette base, vous pouvez écrire une méthode `exec` qui exécute une configuration jusqu'à l'arrêt, et qui se comporte donc de manière analogue à la fonction `exec` écrite en Caml.

La méthode `exec_step` appelle des méthodes `exec_instr` implantés dans chacune des sous-classes de `Instr`. La méthode `exec_instr` d'une instruction i prend comme argument une configuration et effectue les modifications des trois composants de la configuration qui correspondent à la sémantique de l'instruction i. Par exemple, la méthode `exec_instr` de la classe `Fst` remplace le terme de la configuration (une `PairV`) par le premier composant de ce couple.

Exercice 3 Implantez des classes qui réalisent une fonctionnalité comme décrite plus haut. Vous pouvez vous inspirer de la classe `Quote` (pour l'instruction du même nom) qui est un prototype des autres instructions.

Exercice 4 Vous pouvez rajouter d'autres méthodes, par exemple une exécution de n pas de la machine, avec affichage des configurations intermédiaires.

Avertissement : Il faut être conscient que quelques méthodes de Java sont non-fonctionnelles / destructives et peuvent avoir un effet de bord indésirable. C'est notamment le cas pour la fonction `pop()` appliquée à une liste d'instructions, qui peut modifier cette liste si elle est encore référencée ailleurs. Ceci est par exemple le cas pour l'instruction `Call` qui récupère la liste d'instructions associée à un nom de fonction. Lors de l'exécution de ces instruction, il faut éviter de détruire la liste d'instructions originale. Pour cela, il est nécessaire de faire une copie de la liste. Ceci peut être fait, entre autres, avec le constructeur `LinkedList`[1], comme dans :

```
LinkedList<Instr> fun_code_cloned = new LinkedList<Instr> (fun_code_orig);
```

2.4 Générateur de code de Caml vers Java

Le générateur de code est composé de deux parties :

1. d'une fonction de compilation `compile` telle que définie en cours ; elle prend un environnement et un terme à compiler et renvoie une liste d'instructions.

1. voir définition des constructeurs : https://docs.oracle.com/javase/10/docs/api/java/util/LinkedList.html

2. la génération d'un fichier qui contient la séquence d'instructions. Un exemple d'un tel fichier (`Gen.java`) est fourni dans l'archvie des fichiers Java. `Gen` défini une liste d'instructions `code` qui est utilisée dans la configuration initiale de la CAM. Ce fichier est régénéré lors de chaque compilation de code Caml vers Java. Les autres fichiers `*.java` ne sont pas modifiés, voir aussi sect. 2.6.

2.5 Fonctions récursives

Pour prendre en compte des fonctions récursives, il sera nécessaire :
— de rajouter trois nouvelles instructions à la CAM : `Call`, `AddDefs`, `RmDefs`, tant à l'implantation de la CAM en Caml qu'en Java.
— d'implanter la fonction de compilation pour le `let rec`, telle que décrite sur les transparents du cours.

A noter : La compilation de `let rec` avec ce mécanisme est uniquement correcte si les fonctions définies par un `let rec` ne dépendend pas de variables `e` liées à l'exterieur, de la forme `fun e -> (let rec f = fun v -> ... e ... in (f a))`. Même avec cette restriction, on peut écrire des fonctions suffisamment intéressantes. Si vous êtes ennuié par cette limitation, vous pouvez :
— ou bien reconnaître les programmes mal formés, émettre un message d'erreur et refuser de compiler un tel programme ;
— ou encore mieux, réparer de tels programmes, par exemple en rajoutant un paramètre supplémentaire. Le programme problématique serait alors transformé en `fun e -> (let rec f' = fun p -> fun v -> ... p ... in (f' e a))`, avec la fonction `f' e` équivalente à l'ancienne `f`, mais indépendante de variables globales. *L'implantation de cette transformation est relativement complexe ; elle est facultative pour ce projet.*

2.6 Tests

Le processus de compilation de code Caml et d'exécution sous Java est décrit dans la suite. Nous supposons que vous avez compilé le compilateur Caml, typiquement avec un `make` dans le répertoire qui contient vos fichiers `*.ml`. Ceci devrait produire un exécutable `comp` de votre compilateur. Cet exécutable prend comme entrée deux paramètres : le nom d'un fichier source (un fichier `.ml`) et un fichier cible (un fichier `.java`).

Pour compiler et exécuter, vous procédez comme suit :

1. *Génération du fichier cible*, avec la commande `comp Tests/test.ml Java/Gen.java`.
2. *Compilation des fichiers Java*, avec `javac *java` dans le répertoire `Java`. On peut se limiter à compiler `javac Gen.java` si les autres fichiers ont déjà été compilés.
3. *Exécution*, en lançant `java Main`

Nous fournissons quelques fichiers tests :
— `test.ml` : test d'une expression du fragment fonctionnel pur.
— `even_odd.ml` : les fonctions `even` et `odd` définies par récursion récursive ;
— `ackermann_trad.ml`, la fonction d'Ackermann définie de manière traditionnelle (récursion imbriquée sur deuxième argument). Vous savez que la fonction a une croissance très forte ; un appel `ack 4 3` peut faire déborder la pile d'appel de l'évaluateur Caml.
— `ackermann_iter.ml`, la version d'ordre supérieur de la même fonction.

A Compilation et utilisation des fichiers Caml

A.1 Compilation de fichiers

Après téléchargement et extraction de l'archive, allez dans le répertoire contenant le code.

La **compilation des sources Caml** s'effectue en lançant `make` dans la console. Ceci crée des interfaces Caml (`*cmi`), des fichiers objet (`*cmo`), et surtout un exécutable, `comp`.

Lors du **développement de vos fonctions** Caml, nous récommandons la procédure suivante :

1. Avant de modifier des fonctions, assurez-vous que toutes les fonctions compilent correctement (faire un `make` sur la ligne de commande).
2. Lancez l'interpréteur interactif de Caml, de préférence dans un deuxième terminal. Évaluez le fichier `use.ml`, avec `#use "use.ml";;` [2]
3. Commencez à programmer dans les fichiers individuels.
4. Évaluez les fonctions individuellement pour les tester. Éventuellement, il est nécessaire d'ouvrir les modules avec `open` (voir plus bas).
5. Recompilez vos fichiers de temps en temps avec un `make`. Seulement les fichiers modifiés sont recompilés.

Si vous avez l'impression que votre compilateur est stable, vous pouvez l'**exécuter à partir de la console** :

1. Faites un dernier `make` pour être sûr que tout le code est compilé. Ceci devrait produire un fichier exécutable `comp`.
2. Procédez comme décrit dans la sect. 2.6 pour la traduction des fichiers Caml vers Java.

Problèmes typiques :

— Modules : après compilation et chargement avec `#load "nom de fichier.cmo";;`, les fonctions sont définis dans un module. Vous pouvez ou bien les appeler avec leur nom complet, par exemple `Interf.parse`, ou bien ouvrir le fichier pour omettre le préfixe : `open Interf;;` et ensuite appeler `parse`.

— En cas de blocage complet : Sortez de l'interpréteur interactif de Caml (avec `Ctrl-d`). Recompilez tous les fichiers avec `make`. Évaluez de nouveau le fichier `use.ml`.

A.2 Utilisation du parser

Pour tester vos fonctions de typage et de génération de code, il est utile d'utiliser le parser en mode interactif, pour récupérer l'arbre syntaxique correspondant au code que vous avez écrit dans un fichier, par exemple `test.ml`.

Pour utiliser le parser, procédez comme suit :

1. Assurez vous que les fichiers Caml sont correctement compilés, avec un `make`.
2. Lancez une session interactive de Caml et évaluez le fichier `use.ml`, avec `#use "use.ml";;`
3. Pour avoir accès au parser : `open Interf;;`
4. Ensuite, lancez `parse "Tests/test.ml" ;;` (ou choisissez le nom de fichier approprié)
5. Pour éviter le préfixe `Miniml` devant les constructeurs : `open Miniml;;`

Aussi la génération de code et écriture du code sur fichier peut se faire à partir de la session interactive, par exemple avec `generate "Tests/test.ml" "Java/Gen.java";;`

2. Pour `#use` comme pour `#load`, le `#` fait partie de la commande et n'est pas l'invite de Caml.