Projet : Compilateur de C vers Bytecode Java

1 Contexte

1.1 Motivation

Le but de ce projet est de développer un compilateur qui traduit des programmes du langage de programmation C vers le bytecode de Java.

Le langage d'entrée est un sous-ensemble réaliste, mais restreint, du langage C. Vous pourrez donc compiler les programmes source de ce langage aussi avec des compilateurs C traditionnels (par exemple `gcc`), pour vérifier leur correction syntaxique et pour tester le bon fonctionnement de votre traducteur.

Votre compilateur produit un fichier en Java Bytecode qui peut être exécuté (presque) directement par la machine virtuelle de Java. Plus de détails sur Java et le bytecode se trouvent en annexe A, la manipulation des fichiers sous Caml en annexe B.

Les analyses lexicale et syntaxique sont fournies, vous pourrez donc appeler un parser qui prend comme entrée un fichier contenant un programme C et qui construit un arbre syntaxique de ce programme – ou le rejette s'il est mal formé.

L'essentiel du projet consiste donc à générer du Java Bytecode, en passant par quelques étapes intermédiaires : vérification de types ; analyse statique du code ; enfin génération des instructions du bytecode. C'est le résultat de ce travail que vous devez rendre dans le cadre du projet de "Types de données et Preuves".

1.2 Travail demandé

Structure du projet : Le projet se déroule en deux phases :

1. Vérification de types et génération de code pour expressions. Cette partie est à rendre le **vendredi 10 mars 2017 à 23h** au plus tard. Les détails des éléments à fournir sont décrits en sect. 2.4.
2. Le projet complet, à rendre le **mardi 11 avril 2017 à 23h** au plus tard. Les détails des éléments à fournir sont décrits en sect. 3.4.

Faites les exercices des sections suivantes. Dans la plupart des cas, il s'agit d'écrire du code Caml dans les fichiers désignés. Parfois, il s'agit d'exercices d'évaluation ou de réflexion. Dans ce cas, mettez vos observations comme commentaire dans le fichier Caml ou dans le `README`.

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 (sects. 2.4 et 3.4).
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*. L'archive devra contenir :

- Les fichiers source, qui doivent compiler (par un simple `make` à la console) ;
- un fichier `README` contenant une description courte 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 fournie contient les fichiers suivants :

- `lexer.mll` et `parser.mly` : Le lexer/parser pour le langage source.
- `comp.ml` : le fichier principal, contenant la fonction `main` qui permet d'exécuter le code de la console, voir annexe B.
- `lang.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` et `print_instr.ml` : le type de donnés des instructions du langage cible (Java Bytecode), et leur affichage textuel. A priori, vous n'avez pas besoin de toucher ces fichiers.
- `gen.ml` : le fichier qui devra contenir vos fonctions de génération de code (et qui contient actuellement des versions préliminaires).
- `analyses.ml` et `typing.ml`, contenant des fonctions d'analyses statiques et de typage.
- `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.
- Dans le sous-répertoire `Tests` :
  - `jasmin.jar` : le logiciel jasmin
  - `Wrapper.java` : la fonction principale pour exécuter le bytecode.
  - `even.c` : un programme simple

Souvent, les fichiers contiennent des définitions incomplètes, et qui existent uniquement pour permettre une compilation des sources Ocaml sans erreur.

2 Typage et compilation pour expressions

Le travail décrit dans cette section correspond à la première phase du projet : la vérification de types, et la génération de code pour des expressions du langage C. A ce moment, vous n'avez pas encore besoin d'un parser, vous pouvez écrire les expressions de constructeurs en Caml vous-mêmes.

Le fichier `lang.ml` contient les types qui représentent des expressions, instructions et programmes de C. Prenons un exemple : L'expression `n - 2` sera transformée par le parser en

```
BinOp (0,
       BArith BAsub,
       VarE (0, Var (Local, "n")),
       Const (0, IntV 2))
```

Cette représentation interne dit qu'il s'agit d'une opération binaire, plus particulièrement d'une opération arithmétique qui est une soustraction (`BArith BAsub`) dont le premier argument est la variable `n` et la deuxième la constante `2`. Nous revenons sur le premier argument, (le `0`) des constructeurs dans quelques instants.

2.1 Typage d'expressions simples

Programmez vos fonctions dans le fichier `typing.ml`

Dans cette partie, il s'agit

— de vérifier que les expressions du langage C sont bien typées ;

— de les annoter avec des types.

Les types de nos programmes C sont limités : `int` et `bool`.[1] Nous utilisons le type *void* pour représenter le fait qu'une instruction est bien typée. Nous distinguons clairement les expressions (dont le résultat est une valeur, de type `int` ou `bool`) et des instructions (dont le résultat est un changement d'état, de type *void*). En Caml, nous avons donc un type des types :

```
type tp = BoolT | IntT | VoidT
```

Pour faire la vérification de types, nous utilisons un *environnement*, représenté en Caml par

```
type environment =
    {localvar: (vname * tp) list;
     globalvar: (vname * tp) list;
     returntp: tp;
     funbind: fundecl list}
```

Il sera nécessaire de distinguer entre variables locales (`localvar`) et globales (`globalvar`). Les deux sont des listes d'association de nom de variable `vname` et de type `tp`. La distinction entre variable locale et globale sera expliquée plus en détail en sect. 3.1. Pour l'instant, vous pouvez prétendre que toutes les variables sont locales. Les autres éléments de l'environnement sont le type renvoyé par la fonction actuelle (`returntp`) et une liste de liaisons de noms de fonctions (`funbind`), expliquées dans la sect. 2.2.

La fonction de typage des expressions, `tp_expr`, prend un environnement et une expression, et vérifie cette expression et l'annote avec des types. Plus en détail, la fonction `tp_expr` prend une expression de type `int expr` et la convertit en une expression du type `tp expr`, en remplaçant le `0` en première position de chaque constructeur par le type de la sous-expression en question. Si l'expression est mal typée, `tp_expr` lève une exception.

Comme illustration, prenons la représentation de l'expression booléenne `n == k + 1`. Le terme suivant est fourni par le parser :

```
BinOp (0, BCompar BCeq, VarE (0, Var (Local, "n")),
          BinOp (0, BArith BAadd, VarE (0, Var (Local, "k")),
                    Const (0, IntV 1)))
```

Pour l'environnement

```
 {localvar = [("k", IntT); ("n", IntT)]; globalvar = [];
  returntp = VoidT; funbind = []}
```

et ce terme, le résultat de `tp_expr` est

```
BinOp (BoolT, BCompar BCeq, VarE (IntT, Var (Local, "n")),
              BinOp (IntT, BArith BAadd, VarE (IntT, Var (Local, "k")),
                           Const (IntT, IntV 1)))
```

La fonction a donc reconnu que les sous-expressions `n`, `k`, `1` et `k + 1` sont de type `IntT`, tandis que la comparaison elle-même est de type `BoolT`.

Exercice 1 Programmez la fonction `tp_expr` ; omettez pour l'instant le cas de l'appel des fonctions (constructeur `CallE`). La fonction a à peine 16 lignes si vous introduisez des fonctions auxiliaires appropriées. Tenez surtout compte du fait que certaines classes d'opérateurs ont un comportement similaire en ce qui concerne le typage. Ainsi, tous les opérateurs arithmétiques prennent deux entiers et fournissent

1. En C, on peut bien utiliser un type `bool`, à l'aide de la librairie `stdbool` ; voir le fichier `even.c` joint comme test.

un entier, tandis que les opérateurs de comparaison prennent deux valeurs du même type et fournissent un booléen.

2.2 Typage avec fonctions

Programmez vos fonctions dans le fichier `typing.ml`

Pour pouvoir traiter le cas de l'appel de fonctions, il faut savoir comment les déclarations de fonction sont traitées en interne. Supposons que vous avez une fonction en C :

```
int f (int n, bool b) {
  int m;
  m = n + 1;
  if b
     return n;
  else
     return m;
}
```

La première ligne est la *déclaration* de la fonction. Tout ce qui suit entre accolades fait partie de la *définition* de la fonction, à savoir la déclaration d'une variable locale (`m`) et une séquence d'instructions. En Caml, nous avons les définitions suivantes :

```
type vardecl = Vardecl of tp * vname
type fundecl = Fundecl of tp * fname * (vardecl list)
type 'a fundefn = Fundefn of fundecl * (vardecl list) * ('a stmt)
```

La déclaration `fundecl` est donc composée du type de résultat de la fonction, son nom et la liste des paramètres. Dans le cas de notre exemple, nous avons

```
Fundecl(IntT, "f", [Vardecl(IntT, "n"); Vardecl(BoolT, "b")])
```

Il faut clairement distinguer les déclarations de fonction des appels de fonction, par exemple `f(3, true)`. En Caml, les appels sont représentés par les constructeurs `CallE` (pour des expressions) et `CallC` (pour des instructions). Dans le cas de `CallE`, il y a un argument supplémentaire, qui est le type inféré, comme pour toutes nos expressions.

Sans annotation de type, le terme qui correspond à l'expression `f(3, true)` est

```
CallE(0, "f", [Const (0, IntV 3); Const (0, BoolV true)])
```

Pour vérifier qu'un appel de fonction est correctement typé dans un environnement, il faut :

- récupérer la `fundecl` qui correspond à la fonction actuellement appelée (voir le composant `funbind` de l'environnement). Si cette déclaration n'existe pas, on essaie d'appeler une fonction non déclarée (ce qui est une erreur). Il ne devrait pas y avoir plusieurs déclarations pour le même nom de fonction (à vérifier lors de l'analyse des programmes, voir sect. 3).
- vérifier que les types des arguments (de la `CallE`) sont compatibles avec les types des paramètres (dans la `fundecl`).

Exercice 2 Rajoutez la clause pour les appels (`CallE`) à la fonction `tp_expr`.

2.3 Génération de code pour expressions

Programmez vos fonctions dans le fichier `gen.ml`

La génération de code a pour but de créer une séquence d'instructions de Java Bytcode, étant donné une expression bien typée (et, après le travail effectué en sect. 2.2, annotée avec des types).

Par exemple, l'epxression `x - (y + 2)` sera traduite vers la séquence `[iload 0; iload 1; sipush 2; iadd; isub]`.

En fait, nous utilisons un format intermédiaire pour les instructions de Java Bytecode. Vous trouvez ce type `instr` dans le fichier `instrs.ml`. Avec les méthodes définies dans `print_instr.ml`, nous générons le format textuel qui est le format manipulé par Jasmim. Dans la discussion qui suit, nous faisons l'amalgame entre ces deux représentations.

La compilation d'expressions présente deux difficultés :

— le bytecode n'utilise pas de noms symboliques, contrairement aux expressions. Il faut donc établir une correspondance entre les noms des variables (par exemple `x`) et les registres (identifiés par des nombres, ici `0`) qui tiennent les valeurs des variables. L'idée est simple : lors de la traduction d'une expression ou instruction, nous connaissons toutes les variables disponibles, et nous en construisons une liste (par exemple `[x; y]`). Le nombre du registre correspond à la position de la variable dans cette liste (à commencer avec 0).

— il faut maintenir certains invariants. Ainsi, après avoir évalué une expression par une séquence d'instructions du bytecode, la valeur de l'expression doit se trouver au sommet de la pile d'opérandes.

Il y a une troisième difficulté, qui est liée aux branchements conditionnels ou inconditionnels, et qui sera traitée plus tard. Pour l'instant, nous nous limitons aux expressions sans branchement.

Exercice 3 Définissez la fonction `position` qui prend un élément et une liste et qui détermine la position de l'élément dans la liste, comptant à partir de 0.

Exercice 4 (Expressions simples) Écrivez la fonction `gen_expr` qui génère du bytecode pour une expression ; omettez pour l'instant le cas de l'appel des fonctions (constructeur `CallE`), et des expressions qui génèrent des branchements dans le bytecode (surtout le `IfThenElse`). La fonction prend la liste des variables de la fonction actuelle, et l'expression à compiler. La fonction est facile à écrire, seulement le cas des variables demande quelque attention (voir l'annexe A.2). Il est recommandé d'utiliser la fonction `position` de l'exercice 3.

Exercice 5 (Tests) Proposez quelques expressions, générez du code et testez le code généré, en le compilant avec Jasmin et en l'exécutant sout Java.

2.4 Travail attendu à la fin de cette phase

Le code déposé (formats `tar`, `zip` sur Moodle ou dépôt de gestion de version) doit contenir l'ensemble des fichiers qui vous ont été fournis pour ce projet, et où vous avez complété les fichiers

— `typing.ml` comme indiqué plus haut ;

— `gen.ml` comme indiqué plus haut ;

— `README` avec un résumé court du travail effectué, éventuellement des difficultés rencontrées, des cas de test que vous proposez, et leur utilisation.

Tout modification des autres fichiers, et en particulier du `Makefile`, doit être documentée dans le `README`.

Le code doit se compiler sans erreur avec `make`. Tout projet qui ne satisfait pas cette exigence minimale sera rejeté immédiatement. Des "warnings" sont admissibles, sous condition qu'ils soient documentés et justifiés dans le `README`. Si vous n'arrivez pas à compiler votre code malgré vos efforts de localiser l'erreur, commentez les parties du code qui posent problème, avec une description dans le `README`.

3 Typage et compilation pour programmes

La description suivante concerne la deuxième phase du projet. Continuez à travailler avec vos fichiers de la première phase. Bien évidemment, vous pouvez / devez modifier les fonctions écrites avant si vous trouvez des erreurs, ou pour toute optimisation.

3.1 Typage

Programmez ces fonctions dans le fichier `typing.ml`
Le typage pour programmes ne présente plus de problème particulier.

Exercice 6 Écrivez la fonction `tp_stmt` qui type une instruction (`stmt`) selon les mêmes principes que la fonction `tp_expr`. Prenez en compte que le type d'une instruction correctement typée est `VoidT`.

Vous pouvez maintenant faire la vérification de type d'une définition de fonction.

Exercice 7 Écrivez la fonction `tp_fdefn` qui vérifie qu'une définition de fonction est bien typée. Il y a plusieurs aspects à prendre en compte - réfléchissez-y vous-même (par exemple : est-ce qu'une variable peut avoir n'importe quel type ? Est-ce que toutes les combinaisons de noms dans une liste de paramètre sont admissibles ? Quel est le rapport entre le type déclaré de la fonction et le type des valeurs effectivement renvoyées ?

Comme les autres fonctions de typage, `tp_fdefn` prend comme argument un environnement et une `fundefn`. Avant d'appeler la vérification de type du corps de la fonction, il faut initialiser correctement les composants `localvar` et `returntp`.

Exercice 8 Il reste à écrire la fonction `tp_prog` qui vérifie un programme, c'est-à-dire, une liste de déclarations de variables globales et une liste de définitions de fonctions :

```
type 'a prog =  Prog of (vardecl list) * ('a fundefn list)
```

Contrairement aux autres fonctions de typage, `tp_prog` ne prend pas d'environement comme argument, mais construit un environnement initial.

3.2 Génération de code

Programmez ces fonctions dans le fichier `gen.ml`
La génération de code pour des expressions ou instructions qui nécessitent un transfert de contrôle (`goto`) est plus complèxe : il y a un branchement vers une cible, il faut donc générer une étiquette (*label*) qui est unique. La difficulté est que dans un langage fonctionnel comme Caml, il n'est pas possible d'incrémenter, par un effet de bord, un compteur global à chaque fois qu'on a besoin d'une nouvelle étiquette.

Nous proposons deux démarches alternatives que vous pouvez élaborer plus en détail :

1. *Manipulation explicite d'un compteur :* Ce compteur peut simplement être un nombre naturel. Pour cela, il faut rajouter un argument supplémentaire aux fonctions de génération de code (par exemple `gen_expr`), qui représente le prochain nombre qui peut être utilisé. Ce nombre est incrémenté lors de la création d'une nouvelle étiquette. La fonction doit donc renvoyer le nouveau compteur, et en même temps le code généré. Cette solution est un peu lourde parce qu'il faut "séquentialiser" les appels de traduction de sous-expressions (utiliser des `let`), mais les étiquettes sont relativement lisibles (nombres consécutifs qui ne deviennent pas trop larges).
2. *Génération d'étiquettes uniques* en fonction de la position de la sous-expression dans l'arbre syntaxique. La position dans l'arbre syntaxique est une séquence de nombres naturels. Si vous vous plongez dans la n-ième sous-expression d'une expression donnée, vous étendez la séquence par le nombre n. Toutes les sous-expressions ont une position différente ; les étiquettes de ces sous-expressions sont donc uniques par construction. Cette solution est moins évidente, crée des étitquettes plus longues et mains lisibles, mais est plus facile à implanter une fois que le principe est compris : La fonction de génération de code prend un argument de plus (la position actuelle dans l'arbre), mais renvoie uniquement le code généré.

Exercice 9 Rajoutez la génération de code pour les expressions avec branchement (donc les cas qui manquent en exercice 4. Ceci nécessite une gestion des étiquettes, comme décrit en haut.

Le deuxième défi est la génération de code pour des appels de fonction (l'instruction `invokestatic` du Java Bytecode). Lisez la documentation du Java Bytecode pour comprendre la gestion de la pile d'opérandes - il faut empiler les arguments de la fonction appelée dans un certain ordre et ensuite appeler la fonction.

Exercice 10 Écrivez la fonction `gen_stmt` qui génère du code pour des instructions. En principe, la fonction n'est pas très différente de `gen_expr`.

Exercice 11 Écrivez la fonction `gen_fundefn` qui traduit toute une définition de fonction (l'en-tête et le corps). Vous constatez que vous avez besoin de définir les (types des) paramètres, et qu'il vous faut indiquer la taille maximale de la prile d'opérandes et le nombre de registres utilisés.

Pour pouvoir tester votre fonction, donnez d'abord des valeurs constantes pour la taille de la pile et du nombre de registres (qui produiront donc forcément des erreurs lors de la vérification du bytecode, pour certains programmes). Nous verrons plus tard (sect. 3.3) comment donner une estimation réaliste en fonction de la méthode à traduire.

Remarque : Vous constatez que nous générons uniquement des fonctions statiques, dans une classe prédéfinie, qui s'appelle `MyClass` (voir la définition dans le fichier `print_instr.ml`). Ceci est sans grande importance pour la génération de code.

3.3 Vérifications

Programmez ces fonctions dans le fichier `analyses.ml`.

Nous allons implanter quelques analyses statiques des programmes source.

Exercice 12 Écrivez la fonction `stmt_returns` qui vérifie si une fonction termine chaque branche d'exécution correctement avec un `return` ou un `return` e, où e est une expression. Le bon typage de la valeur renvoyée (en cas d'un `return`) est déjà assurée par la vérification de types, mais pas l'existence du `return`.

Exercice 13 Cet exercice vise à estimier le nombre de variables locales et la profondeur de la pile d'opérandes. Les deux sont essentiels pour la vérification de bytecode de Java.

Le nombre maximal de variables et facile à calculer et à intégrer dans la fonction `gen_fundefn`.

Pour déterminer la taille maximale de la pile d'opérandes, écrivez la fonction `stack_depth_c` pour les instructions (et vous aurez besoin d'une fonction similaire pour les expressions). Vous avez le droit de calculer une approximation qui n'est pas parfaitement précise, mais qui ne devrait pas surévaluer la profondeur grossièrement (par exemple par un facteur exponentiel).

Voici un test : quelle est la taille de la pile pour l'expression `(((n + 1) + 2) + 3) + 4` ? Et pour l'expression `n + (1 + (2 + (3 + 4)))` ?

Exercice 14 Implantez une vérification de *definite assignment*, voir la description dans la spécification pour le langage Java [2].

Nous proposons de procéder comme suit : Définissez d'abord la fonction `defassign_e` qui prend un ensemble de variables `vs` (à savoir les variables qui ont certainement une valeur) et une expression `e` et qui vérifie que `e` a certainement une valeur définie, ce qui est uniquement assuré si toutes les variables de `e` sont contenues dans `vs`. Cette fonction renvoie un booléen.

Définissez ensuite la fonction `defassign_c` qui fait la vérification pour les instructions. Contrairement à `defassign_e`, cette fonction renvoie l'ensemble des variables qui ont certainement une valeur définie après exécution de l'instruction. En cas d'évaluation d'une expression indéfinie (`defassign_e` renvoie "faux" pour la sous-expression en question) , la fonction `defassign_c` lève une exception.

2. http://docs.oracle.com/javase/specs/jls/se8/html/jls-16.html

Pour la manipulation des ensembles, vous pouvez utiliser le module `Set` de Caml[3]. L'ensemble `StringSet` (ensemble de chaînes de caractères) est déja prédéfini.

Par exemple, l'expression `StringSet.mem "a" (StringSet.of_list ["a"; "b"])` est vraie.

3.4 Travail attendu à la fin de cette phase

Comme pour la première phase (sect. 2.4), vous devez :

— fournir votre projet intégral, avec les modifications dans les fichiers `typing.ml`, `gen.ml` et `analyses.ml` telles que décrites plus haut.

— Vous devez documenter le fonctionnement de votre compilateur dans le `README`. Mettez en avant les points forts, ne cachez pas les faiblesses (même si nous allons les découvrir en tout cas ...).

— Ajoutez quelques tests (programmes C) dans le répertoire `Tests`.

Les exigences de la sect. 2.4 tiennent toujours : votre code `Caml` doit compiler avec un simple `make`. En plus, votre compilateur doit être utilisable à partir de la ligne de commande, avec un appel de la forme `comp source.c Cible.j`. Le fonctionnement exact est à documenter dans le `README`. Il est très fortement souhaitable que votre compilateur soit correct, dans le sens qu'il doit

— produire un code jasmin/ Java sémantiquement équivalent au code source ;

— ne pas provoquer des erreurs de vérification statique (par le *bytecode verifier*) lors du lancement du programme compilé. Entre autres, la taille des piles d'opérandes doit être dimensionnée correctement, il ne doit pas y avoir d'erreur de variable indéfinie (*definite assignment*). Ces erreurs doivent donc être détectées par votre compilateur lors de la compilation, et non lors de l'exécution.

Les messages d'erreurs de votre compilateur doivent indiquer clairement la source du problème, mais nous ne demandons pas une présentation sophistiquée ou enjolivée (la levée d'une exception en Caml est souvent suffisante).

A Java et Java Bytecode

Lisez l'intégralité de cette section avant de commencer à expérimenter (sect. B.1.)

A.1 Ecosystème Java

Java est up langage de programmation développé autour de 1995 par Sun Microsystems (maintenant racheté par Oracle). Comme presque tous les langages, Java ne peut pas être exécuté directement, mais doit être compilé. Pour assurer son indépendance vis-à-vis d'une plateforme matérielle, Java est compilé vers un "assembleur" de haut niveau, le *Java Bytecode*, qui de sa part est interprété par un environnement d'exécution, la machine virtuelle de Java (JVM). Java et la JVM sont documentés sur les pages d'Oracle [4]. Une introduction au bytecode se trouve en sect. A.2.

Ainsi, le programme `Wrapper.java` peut être compilé avec `javac Wrapper.java`, et ceci produit le fichier bytecode `Wrapper.class` qui peut être exécuté avec `java Wrapper`. La commande `java` lance donc la machine virtuelle qui interprète le fichier bytecode `Wrapper.class`.

Les fichiers `*.class` ne sont pas destinés à une lecture humaine, mais avec `javap -c -verbose *.class`, on peut afficher le bytecode des fichiers `*.class` produits par le compilateur `javac` (ou un autre).

A.2 Java Bytecode

Le Java Bytecode est (essentiellement) une séquence d'instructions très simples. Nous nous concentrons ici sur quelques instructions de base ; une documentation de l'ensemble de toutes les instructions [5] et le

3. http://caml.inria.fr/pub/docs/manual-ocaml/libref/type_Set.html
4. http://docs.oracle.com/javase/specs/
5. http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html

résumé de toutes les instructions [6] se trouve sur la page d'Oracle.

De notre point de vue (la réalité est plus complexe), le bytecode manipule deux structures de données : un ensemble de *registres* et une *pile d'opérandes*. Il y a un nombre (en principe) non borné de registres, numérotés à partir de 0, et aussi la taille de la pile est en principe non borné. Une pile est une structure de données qui permet d'empiler plusieurs éléments dont typiquement seulement le premier (le sommet) est directement accessible. "Pousser" un élément sur la pile veut dire : mettre l'élément au sommet de la pile, et maintenir les autres éléments dans l'ordre.

Nous nous limitons à des opérations sur des entiers, et nous utilsons les opérations suivantes. Certaines de ces opérations ont des contraintes de bonne formation (pile non vide etc.), et les effets de non-respect de ces contraintes sont décrites dans la documentation.

Transfert entre pile et registres ; constantes

— `iload` n pousse le contenu du registre n sur la pile d'opérandes.
— `istore` n enlève le sommet de la pile d'opérandes et le met dans le registre n.
— `sipush` c pousse la constante c (un entier) sur la pile.
— `swap` permet d'inverser le sommet et le deuxième élément de la pile.

Opérations arithmétiques Les opérations arithmétiques s'effectuent uniquement sur la pile. A titre d'exemple, prenons `isub` : La pile doit contenir les valeurs v_1 (avant-dernier élément) et v_2 (sommet). L'effet de `isub` est d'enlever v_1 et v_2 de la pile et de les remplacer par la valeur $v_1 - v_2$.

Nous utilisons aussi les opérations `iadd` (addition), `imul` (multiplication), `idiv` (division), `irem` (reste de division / modulo).

Appel de fonctions Lors d'un appel d'une fonction (méthode statique), les n arguments de la fonction sont stockés dans les registres $r_0 \dots r_{n-1}$ et peuvent donc être utilisés dans le corps de la fonction appelée. L'instruction `ireturn` renvoie le sommet de la pile à la fonction appelante.

Taille maximale de la pile / nombre de registres Une définition de fonction contient une définition de la taille maximale de la pile, par exemple

```
.limit stack 3
```

et le nombre de registres utilisés :

```
.limit locals 3
```

Java vérifie (avant l'exécution !) que ces bornes sont respectées. Il faut en tenir compte lors de la génération de code.

Une exécution typique est montrée dans la fig. 1, à commencer avec une configuration initiale c_0 et en exécutant les instructions indiquées en bas de figure. Les trois registres r_0, r_1, r_2 sont en vert, les éléments de la pile en bleu.

B Compilation et utilisation des fichiers Caml

B.1 Premier test avec Caml, Java et Jasmin

Dans le répertoire `Tests`, vous trouvez les fichiers `Wrapper.java` et `jasmin.jar`. Pour vérifier que votre environnement est opértionnel, procédez comme suit :

— Extrayez l'archive. Vous obtenez le répertoire `Projet_Compil`. Descendez dans ce répertoire.
— Compilez les sources Caml (avec `make`), descendez dans le répertoire `Tests`

6. http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html

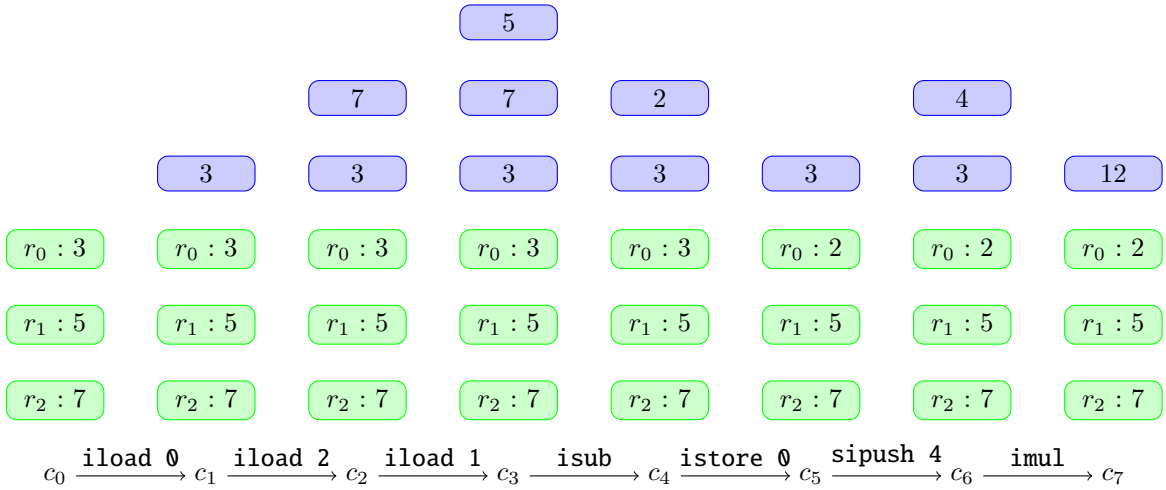

FIGURE 1 – Exécution typique

— Générez `Even.j` avec `../comp even.c Even.j`
— Générez `Even.class`, avec `java -jar jasmin.jar Even.j` Ceci génère la classe `MyClass.class`.
— Compilez `Wrapper.java`, avec `javac Wrapper.java`
— Exécutez le programme : `java Wrapper 3`

Le programme devrait afficher le résultat `0` (d'ailleurs pour n'importe quel argument de `Wrapper`). Quand votre compilateur sera opérationnel, vous aurez des résultats réalistes ...

B.2 Compilation de fichiers

Après téléchargement et extraction de l'archive, allez dans le répertoire contenant le code.
Pour **tester la compilation des sources Caml**, procedez comme suit :

1. Compilez tous les fichiers avec `make`. Ceci crée des interfaces Caml (`*cmi`), des fichiers objet (`*cmo`), et surtout un exécutable, `comp`.
2. Déscendez dans le sous-répertoire `Tests`, où vous trouvez le fichier `even.c`, qui est une fonction C standard et que vous pouvez compiler avec un compilateur C, par exemple avec `gcc -c even.c`
3. Testez l'exécutable, avec `../comp even.c Even.j` Ceci va créer le fichier `Even.j` (s'il n'existe pas encore) ou le remplacer, avec un nouveau contenu. Lors de la première exécution, `Even.j` contient un message banal. Vous écrirez un compilateur dont le code pourra être utilisé directement par l'assembleur `jasmin`.

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";;` [7]
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.

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

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. Descendez dans le répertoire `Tests`. La séquence typique est :
   (a) Compilation (utilisant votre compilateur) du fichier `even.c`, avec

   ```
   ../comp even.c Even.j
   ```

   (b) Traduction du fichier `Even.j` que votre compilateur vient de produire :

   ```
   java -jar jasmin.jar Even.j
   ```

   Ceci produit `MyClass.class`

   (c) Faites éventuellement des adaptations dans le fichier `Wrapper.java` ; ensuite, compilation avec `javac Wrapper.java`. Vous pouvez omettre cette étape si `Wrapper.java` n'est pas modifié.

   (d) Exécution, par exemple avec `java Wrapper 42` (nombre d'arguments en fonction de ce qui est attendu par votre `Wrapper`).

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 `Typing.tp_expr`, ou bien ouvrir le fichier pour omettre le préfixe : `open Typing;;` et ensuite appeler `tp_expr`.

— 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`.

B.3 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 `even.c`.

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/even.c";;` (ou choisissez le nom de fichier approprié)
5. Pour éviter le préfixe `Lang` devant les constructeurs : `open Lang;;`

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/even.c" "Tests/Even.j";;`