Projet de Compilation Avancée
Compilation d'un langage impératif vers la Mini-ZAM

Version du 15 février 2022

Objectif Ce projet consiste en la réalisation d'un compilateur produisant du bytecode interprétable par une implémentation simplifiée de la machine virtuelle OCaml, appelée Mini-ZAM. Les sources de la Mini-ZAM sont fournis ainsi qu'un ensemble de jeux de tests. Le langage d'implantation du compilateur est libre.

Rendu Le rendu du projet se fera via la plateforme Moodle. Vous déposerez une archive au format `.tar.gz` qui devra comporter :

- l'implémentation du compilateur demandé, incluant un *Makefile* et un fichier *README* expliquant la marche à suivre pour installer et exécuter votre programme en combinaison avec la Mini-ZAM,
- un rapport (en français) décrivant la structure générale du projet, vos schémas de compilation, vos choix d'implantation et les extensions que vous aurez choisies de traiter,
- de nouveaux jeux de tests, couvrant notamment les extensions traitées.

Langage source Le langage source du compilateur est un mini-langage impératif dont la syntaxe est définie figure 1. Ce langage partage des similitudes avec le noyau impératif d'OCaml. Le projet n'aborde pas la question du typage (statique ou dynamique) des programmes sources, mais cet aspect peut impacter l'exécution du bytecode engendré par le compilateur. C'est pourquoi vous expliquerez dans votre rapport la manière [1] dont vous traitez les programmes sources incohérents du point de vue du typage, tels que **print** (**not** 42) ou **while** 42 **do print** 0 **done**.

Barème provisoire L'implantation du compilateur sans extension est notée sur 12 points, pour une note totale sur 20. Chaque extension rapporte un certain nombre de points supplémentaires. Les extensions réalisées peuvent (en tout) rapporter d'avantage que 8 points. Le nombre d'extensions à implémenter est libre.

Plan de ce document La section 1 présente succinctement le jeu d'instructions de la Mini-ZAM. La section 2 propose une approche permettant d'implémenter le compilateur de façon progressive. Vous êtes libres de suivre une autre approche ou définir des schémas de compilation différents de ceux proposés. La section 3 liste des extensions optionnelles. L'annexe 4 est une présentation détaillée de la Mini-ZAM. Cette présentation est reprise du projet de Compilation Avancée de 2018.

1 Jeu d'instructions de la Mini-ZAM

La Mini-ZAM est une machine virtuelle OCaml comprenant un interprète de bytecode et une bibliothèque d'exécution. L'interprète comprend une pile ainsi que plusieurs registres dont pc (le pointeur de code), accu (l'accumulateur) et env (l'environnement). La bibliothèque d'exécution implémente un tas avec un récupérateur automatique de mémoire (GC).

1. On pourrait supposer que les programmes sources sont tous bien typés (et dans ce cas décrire informellement les règles de typage utilisées) ou encore formuler des hypothèses sur la représentation des valeurs du langages (par exemple, coder la valeur **false** par l'entier 0 et imposer que toute valeur qui n'est pas 0 est *vraie*).

π ::=		Programme
	$\mid s$	instruction
s ::=		Instruction
	$\mid$ **print** e	affichage d'un entier
	$\mid$ **begin** b **end**	bloc
	$\mid$ **let** $x = e$ **in** s	liaison locale
	$\mid x := e$	modification physique d'une référence
	$\mid$ **if** e **then** s **else** s'	conditionnelle
	$\mid$ **while** e **do** b **done**	boucle
b ::=		Bloc
	$\mid s$	instruction
	$\mid s \ ; \ b$	séquence
e ::=		Expression
	$\mid c$	constante
	$\mid x$	variable locale
	$\mid (\ominus e)$	application d'opérateur unaire
	$\mid (e_1 \oplus e_2)$	application d'opérateur binaire
	$\mid$ (**ref** e)	allocation et initialisation d'une référence
	$\mid (! \ x)$	déréférencement d'une référence
c ::=		constante
	$\mid n$	entier
	$\mid$ **true** $\mid$ **false**	booléen
$\ominus$ ::=		opérateur unaire
	$\mid$ **not**	
$\oplus$ ::=		opérateurs binaires
	$\mid$ `+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `==` $\mid$ `!=`	
	$\mid$ `<` $\mid$ `>` $\mid$ `<=` $\mid$ `>=`	

FIGURE 1 – Syntaxe du language source

Chaque valeur manipulée par la Mini-ZAM est un mot de 32 bits :

— Soit une valeur immédiate, c'est-à-dire un entier sur 31 bits suivi d'un *bit de marque* à $\underline{1}$,

— Soit l'adresse (sur 32 bits) d'un mot mémoire – le *bit de marque* étant alors à $\underline{0}$ puisque l'adresse d'un mot est toujours un multiple de 4.

Chaque instruction de bytecode peut être précédée par une étiquette ℓ afin d'y faire référence en d'autres points du programme (par exemple, comme cible d'un branchement **BRANCH** ℓ). De ce point de vue, le compilateur peut traiter la pose d'étiquette comme une instruction spéciale **LABEL** ℓ telle que l'impression Print⟦**LABEL** ℓ; C⟧ produise le texte "ℓ: Print⟦C⟧". Une limitation de la Mini-ZAM impose que le bytecode interprété ne comporte jamais deux étiquettes successives ℓ_1: ℓ_2: C. Le compilateur devra donc contourner cette difficulté en réalisant une passe de « nettoyage » de la liste d'instructions de bytecode engendrée par compilation de chaque programme source. La liste d'instructions peut ensuite être imprimée sur la sortie standard ou dans un fichier.

Les instructions supportées par la Mini-ZAM sont décrites en annexe 4. Voici les principales. **STOP** interrompt l'exécution du programme. **CONST** n place la valeur immédiate n dans accu. **PUSH** empile la valeur située dans accu. **POP** dépile (et ignore) une valeur. **ACC** i place dans accu la i-ème valeur stockée dans la pile (l'indice $i = 0$ correspond au sommet de pile). **PRIM** *unop* applique la primitive *unop* à la valeur de accu et place le résultat dans accu. **PRIM** *binop* dépile une valeur v_0 puis applique la primitive *binop* à la valeur de accu et à v_0. **PRIM** `print` imprime la valeur de accu et place une valeur spéciale () dans accu. **BRANCH** ℓ place dans pc la position de l'étiquette ℓ dans le bytecode. **BRANCHIFNOT** ℓ place dans pc la position de l'étiquette ℓ si la valeur de accu est 0. **MAKEBLOCK** n crée un bloc de taille n, place son adresse dans accu ; si le bloc n'est pas vide (c'est-à-dire si $n > 0$), son premier champ est initialisé à la valeur de accu tandis que les $n - 1$ champs suivants prennent chacun une nouvelle valeur dépilée. **GETFIELD** n place dans accu la valeur n-ème champ du bloc dont l'adresse est située dans accu. **SETFIELD** n dépile une valeur v_0 puis assigne v_0 aux n-ème champ du bloc dont l'adresse est située dans accu. Lors de l'exécution de chaque instruction, pc est incrémenté à l'exception des branchements vers une étiquette.

L'implémentation C de la Mini-ZAM comprend un *mode debug* qui peut-être activé manuellement en dé-commentant la directive `#define DEBUG` dans interp.c ligne 14. Cela permet de visualiser le contenu des registres et de la pile après chaque instruction exécutée.

2 Implémentation du compilateur : approche proposée (12 points)

Cette section suggère une marche à suivre pour implémenter le compilateur.

1. On définit d'abord la compilation des expressions arithmétiques (sans variables locales) ainsi que la compilation des instructions **print** e et **begin** b **end**. Cela permettra de tester rapidement de premiers programmes.
2. On demande alors de définir une règle de compilation de l'instruction **if** e **then** s **else** s'.
3. On définit la compilation de l'instruction de liaison locale **let** $x = e_1$ **in** e_2 et on demande de définir une règle de compilation des occurrences de variables locales.
4. On définit la compilation de l'expression (**ref** e) allouant une référence (*i.e.*, un pointeur vers une cellule mémoire mutable) et on demande de définir une règle de compilation de l'accès à la valeur pointée (! x) et la modification physique de la valeur pointée $x := e$.
5. Enfin on demande de définir une règle de compilation de l'instruction **while** e **do** s **done**.

Expressions arithmétiques La compilation d'une constante c, notée Const⟦c⟧, engendre une suite d'instructions de bytecode OCaml permettant de placer ladite constante dans le registre accu de l'interprète de bytecode de la Mini-ZAM. Par convention, on encode la constante **true** (respectivement **false**) par la valeur immédiate 1 (respectivement 0).

$$\begin{array}{lcl} \text{Const⟦}n\text{⟧} & = & \textbf{CONST}\ n \\ \text{Const⟦}\textbf{true}\text{⟧} & = & \textbf{CONST}\ 1 \\ \text{Const⟦}\textbf{false}\text{⟧} & = & \textbf{CONST}\ 0 \end{array}$$

La compilation d'une expression e, notée $\text{Expr}[\![e]\!]_{\rho,k}$, engendre une suite d'instructions de bytecode OCaml permettant de calculer la valeur de l'expression e et de placer cette valeur dans le registre `accu` de la Mini-ZAM. Par exemple, la compilation des expressions arithmétiques peut être définie par :

$$\begin{array}{lcl}
\text{Expr}[\![c]\!]_{\rho,k} & = & \text{Const}[\![c]\!] \\
\text{Expr}[\![(\ominus\ e)]\!]_{\rho,k} & = & \text{Expr}[\![e]\!]_{\rho,k};\ \textbf{PRIM}\ \ominus \\
\text{Expr}[\![(e_1 \oplus e_2)]\!]_{\rho,k} & = & \text{Expr}[\![e_2]\!]_{\rho,k};\ \textbf{PUSH};\ \text{Expr}[\![e_1]\!]_{\rho,k+1};\ \textbf{PRIM}\ \oplus
\end{array}$$

La fonction $\text{Expr}[\![.]\!]_{\rho,k}$ est paramètrée par un environnement de compilation ρ (intuitivement une suite d'identificateurs) et un entier k désignant le niveau de pile. Ces paramètres, ρ et k, serviront par la suite à définir la compilation $\text{Expr}[\![x]\!]_{\rho,k}$ des variables locales.

Séquences, affichages et conditionnelles On peut définir comme suit la compilation des instructions, des blocs, et finalement des programmes, vers du bytecode OCaml. Le symbole ε désigne l'environnement de compilation vide.

$$\begin{array}{lcl}
\text{Stat}[\![\textbf{print}\ e]\!]_{\rho} & = & \text{Expr}[\![e]\!]_{\rho};\ \textbf{PRIM}\ \texttt{print} \\
\text{Stat}[\![\textbf{begin}\ b\ \textbf{end}]\!]_{\rho} & = & \text{Block}[\![b]\!]_{\rho} \\
 & & \\
\text{Block}[\![s]\!]_{\rho} & = & \text{Stat}[\![s]\!]_{\rho} \\
\text{Block}[\![s;\ b]\!]_{\rho} & = & \text{Stat}[\![s]\!]_{\rho};\ \text{Block}[\![b]\!]_{\rho} \\
 & & \\
\text{Prog}[\![s]\!] & = & \text{Stat}[\![s]\!]_{\varepsilon};\ \textbf{STOP}
\end{array}$$

À l'aide des instructions de bytecode **BRANCH** ℓ et **BRANCHIFNOT** ℓ, et en représentant les étiquettes avec la pseudo-instruction **LABEL** ℓ, définir une règle de compilation des conditionnelles : **if** e **then** s **else** s'.

2.1 Liaisons locales

L'instruction **let** $x = e$ **in** s lie la valeur de l'expression e au nom x dans l'instruction s. On peut la compiler comme suit. La notation $(x :: \rho)$ désigne l'extension de l'environnement de compilation ρ avec la variable x.

$$\text{Stat}[\![\textbf{let}\ x = e\ \textbf{in}\ s]\!]_{\rho} \quad = \quad \text{Expr}[\![e]\!]_{\rho,0};\ \textbf{PUSH};\ \text{Stat}[\![s]\!]_{(x::\rho)};\ \textbf{POP}$$

Écrivez la règle de compilation $\text{Expr}[\![x]\!]_{\rho,k}$ des occurrences de variables. On pourra, pour cela, utiliser l'instruction de bytecode (**ACC** i) permettant d'accéder à la i-ème valeur dans la pile.

Conseil : tester avec la Mini-ZAM en mode debug pour voir le contenu de la pile et des registres à chaque pas d'exécution.
Indice : $\text{Expr}[\![x]\!]_{x::\varepsilon,0} = \textbf{ACC}\ 0$.

2.2 Références mutables

Une référence est un pointeur vers un case mémoire modifiable physiquement. L'expression (**ref** e) alloue une référence initialisée avec la valeur de e. On peut compiler cette construction comme suit :

$$\text{Expr}[\![(\textbf{ref}\ e)]\!]_{\rho,k} = \text{Expr}[\![e]\!]_{\rho,k};\ \textbf{MAKEBLOCK}\ 1$$

L'expression $(!\ x)$ retourne la valeur stockée dans la référence liée à la variable x. L'expression $(x := e)$ modifie physiquement la valeur stockée dans la référence liée à la variable x, la nouvelle valeur étant obtenue en évaluant e.

Proposez un schéma de compilation pour ces deux constructions.

2.3 Boucles

Proposez un schéma de compilation pour la construction **while** e **do** b **done**. On devrait alors être en mesure de compiler, puis faire exécuter par la Mini-ZAM, le programme source de la figure 2, calculant la factorielle de 10 et affichant le résultat.

```
let n = (ref 10) in
let acc = (ref 1) in
begin
  while ((!n) > 1) do
    acc := ((!n) * (!acc));
    n := ((!n) - 1)
  done;
  print (!acc)
end
```

FIGURE 2 – Exemple de programme source

3 Extensions

La suite du sujet propose des extensions au choix, indépendantes les unes des autres. Cette section du document peut être amenée à évoluer. Nous vous conseillons de choisir une extension de votre analyseur syntaxique, une extension sur les structures de données (listes ou tableaux) et une extension parmi les ruptures de calculs ou les fonctions.

3.1 Analyse syntaxique (3 points)

Enrichir votre analyseur syntaxique, en gérant par exemple :

— des commentaires dans les programmes sources,

— des règles de priorités pour éviter de parenthéser systématiquement les expressions,

— une instruction **skip** (c'est-à-dire « *ne rien faire* »),

— un *if* unilatère : **if** e **then** s,

— des liaisons locales multiples : **let** $x_1 = e_1$ **and** $\cdots x_n = e_n$ **in** s,

— une variante syntaxique pour les liaisons locales : (s **where** $x = e$),

3.2 Structures de données (3 points)

Étendre le langage avec des listes ou des tableaux.

— À la manière des références mutables, étendre le langage avec des tableaux :

$$s ::= \cdots \mid x[e_2] \coloneqq e_3$$
$$e ::= \cdots \mid \{e_1, \cdots e_n\} \mid x[e] \mid (\textbf{length}\ e)$$

— Étendre le langage avec des listes. Une liste est soit une valeur spéciale **nil** (qui peut-être être représentée par la valeur immédiate 0), soit une bloc alloué (**cons** v_0 v_1) à deux champs v_0 et v_1. On accède à ces champs respectivement par les constructions (**hd** e) et (**tl** e) qui produisent une erreur si e s'évalue en une liste vide. La construction (**empty** e) teste si une liste est vide.

$$e ::= \cdots \mid (\textbf{cons}\ e_1\ e_2) \mid (\textbf{hd}\ e) \mid (\textbf{tl}\ e) \mid (\textbf{empty}\ e)$$
$$c ::= \cdots \mid \textbf{nil}$$

En cas d'accès mémoire erroné comme x[(**length** x)] ou (**hd nil**), le programme devrait s'arrêter immédiatement (avec l'instruction de bytecode **STOP**), sans produire d'erreur d'exécution (e.g. accès hors des bornes).

3.3 Ruptures de calculs (2 points)

Ajouter les instructions **break** et **continue**.

$$s ::= \cdots \mid \textbf{break} \mid \textbf{continue}$$

3.4 Fonctions (6 points)

Ajouter des valeurs fonctionnelles au langage. Le corps d'une fonction est un bloc. L'instruction **return** e retourne la valeur de l'expression e dans le contexte de l'appelant.

$s ::= \cdots \mid$ **return** e

$e ::= \cdots \mid$ **fun** x `->` $b \mid (e_1\ e_2)$

Dans un premier temps, on peut se limiter aux fonctions closes (sans variables libres, l'environnement lexical est vide).

Vous pouvez être amené à modifier la signature des schémas de compilation $\mathrm{E}[\![]\!]_{\rho,k}$, ... afin de distinguer, dans l'environnement de compilation, les variables provenant de l'environnement lexical [2] d'une part, et les variables stockées dans la pile d'autre part. Au lieu d'enclore l'environnement d'exécution entier dans chaque fermeture créée, on peut ne conserver que les variables libres du corps de la fonction.

3.5 Extension libre (à valider avec l'enseignant)

Étendre le compilateur ou l'implémentation C de la machine virtuelle.

Par exemple, ajouter de nouvelles primitives C dans la bibliothèque d'exécution de la Mini-ZAM et adapter le compilateur pour exploiter ces primitives dans les programmes sources.

4 Annexe : Description de la machine virtuelle

Cette annexe est reprise du projet CA de 2018.

Valeurs : La machine virtuelle que vous réaliserez manipulera des valeurs d'un type que nous nommerons dans ce sujet `mlvalue`. Au départ, ces valeurs correspondront à deux catégories de valeurs distinctes :

1. Des entiers, qui représenteront toutes les valeurs immédiates manipulées par le programme (on représentera par conséquent `true` par l'entier 1, `false` par 0, et `()` par 0, tandis qu'une valeur entière quelconque sera représentée tel quel).
2. Des fermetures, qui sont des couples (*pointeur de code*, *environnement*) qui représentent les valeurs fonctionnelles manipulées par le langage. Un pointeur de code pourra être représenté par un entier, tandis qu'un environnement sera une collection de valeurs.

   Dans la suite, nous représenterons une fermeture associée à un pointeur de code `pc` et un environnement `e` de la façon suivante : { `pc , e` } .

Le type `mlvalue` pourra être plus tard étendu pour représenter d'autres types de valeurs. Vous ferez donc attention à ce que votre implémentation soit suffisamment souple pour gérer sans trop de difficulté ces ajouts.

2. on peut y accéder avec l'instruction de bytecode `ENVACC i`

Registres de la VM : La *Mini-Zam* est composée de 5 registres :

1. `prog`, un tableau de couples $(label, instruction)$ qui représente le programme en cours d'interprétation.
2. `stack`, la pile dans laquelle seront placées les paramètres des fonctions, des pointeurs de code, des fermetures, etc : c'est une structure LIFO (*Last In First Out*) qui contient des `mlvalue` (au début du programme, la pile est vide).
   Dans la suite, nous représenterons les éléments dans la pile entre crochets `[...]`, avec la tête de pile correspondant à la valeur la plus à gauche.
3. `env`, l'environnement de la fermeture courante : c'est un collection de `mlvalue` qui représente les valeurs accessibles pour la fonction en cours d'exécution (au début du programme, l'environnement est vide).
   Dans la suite, nous représenterons les éléments dans l'environnement entre chevrons `<...>`.
4. `pc`, le pointeur de code vers l'instruction courante (au début du programme, `pc = 0`).
5. `accu`, un accumulateur utilisé pour stocker certaines valeurs (`mlvalue`) intermédiaires (au début du programme, l'accumulateur vaut `()`).

Le rôle de chaque instruction bytecode est de modifier tout ou partie des valeurs contenues dans ces registres. L'état de ces registres représentera ainsi à tout instant de l'exécution l'état de la machine virtuelle. Vous devrez fournir des fonctions permettant d'afficher lors de l'interprétation d'un fichier bytecode le contenu de ces registres.

5 Description des fichiers bytecode

Instructions : Le *bytecode* associé à un programme de notre langage « *mini-ML* » vous sera fourni sous la forme d'un fichier texte, dans lequel chaque ligne représente une instruction à exécuter. L'instruction est représentée par un caractère de tabulation (→), suivi par son nom, en lettres majuscules. Par exemple, la ligne suivante correspond à l'instruction `PUSH` qui ajoute une valeur en tête de la pile :

```
→PUSH
```

Arguments : Certaines instructions du bytecode doivent être paramétrées par une ou plusieurs valeurs. Ces arguments sont ajoutés à la suite du nom de l'instruction (après un espace), séparés par des virgules. Par exemple, la ligne suivante représente l'instruction `CLOSURE` (création de fermeture) avec comme argument un label L1 et une valeur 0 :

```
→CLOSURE␣L1,0
```

Labels : Enfin, certaines instructions sont identifiées par un label, afin d'y faire référence depuis un autre point du bytecode. Une instruction labellisée commence par une chaîne de caractères qui se termine par le symbole ':'. Ce label est suivi d'un caractère de tabulation, puis de l'instruction bytecode associée. Par exemple, la ligne suivante représente l'instruction d'application de l'opérateur `+`, labellisée par un label `N` :

```
N:→PRIM␣+
```

Exemple : L'exemple suivant est le contenu d'un fichier bytecode qui correspond au programme (**if** `true` **then** `2` **else** `3`) :

```
→CONST␣1
→BRANCHIFNOT␣L
→CONST␣2
→BRANCH␣M
L:→CONST␣3
M:→STOP
```

6 Instructions de base et fonctions unaires

Nous vous demandons en premier lieu d'implémenter un interprète de bytecode Mini-ML qui puisse traiter des programmes faisant usage d'opérateurs de base (comme les opérateurs arithmétiques ou le `if then else`) et permettant l'application de fonctions non-récursives qui n'ont pour le moment qu'un argument. La gestion de tels programmes entraîne la définition d'une douzaine d'instructions distinctes.

Nous présentons dans la suite le détail du fonctionnement de ces dernières [3] , ainsi que leur effet sur les registres de la machine virtuelle.

Attention : *Pour chaque instruction, seuls les registres ayant été modifiés (ou ayant un rôle dans l'instruction) seront mentionnés. En plus des modifications explicitement décrites, le pointeur de code (`pc`) sera incrémenté à la fin de chaque instruction (sauf dans les cas où il est précisé que `pc` prend une valeur différente).*

1. **CONST n** (Valeur constante)
   Description :
   — L'accumulateur (`accu`) prend pour valeur la constante n.

	Avant	Après
`accu`	_	n

2. **PRIM op** (Application de primitive)
   Description :
   — *op* est une primitive parmi :
   ◦ des opérateurs arithmétiques à deux arguments : `+`,`-`,`/`,`*`
   ◦ des opérateurs logiques à deux arguments : `or`,`and`
   ◦ l'opérateur booléen `not`
   ◦ des opérateurs de comparaison `<>`,`=`,`<`,`<=`,`>`,`>=`
   ◦ une primitive d'écriture (d'un caractère représenté par sa valeur ASCII) sur *stdout* : `print`
   — Pour les opérateurs à deux arguments, *op* est appliqué à l'accumulateur et à une valeur dépilée, et le résultat est mis dans `accu` :

	Avant	Après
`stack`	$[a_0; a_1; ...]$	$[a_1; ...]$
`accu`	x	$op(x, a_0)$

   — Pour les opérateurs unaires, *op* est appliqué à l'accumulateur, et le résultat est mis dans `accu` :

	Avant	Après
`accu`	x	$op(x)$

3. **BRANCH L** (Branchement)
   Description :

3. Il est à noter qu'à des fins de simplification les instructions décrites dans ce sujet diffèrent pour certaines des instructions de la ≪ vraie ≫ machine virtuelle OCaml

— `pc` prend pour valeur la position du label L.

— On utilisera une fonction *position* qui associe un label à une position dans le code (une valeur entière)

	Avant	Après
`pc`	_	$position(L)$

4. **BRANCHIFNOT L** (Branchement conditionnel)
   Description :
   — Si l'accumulateur vaut 0, alors `pc` prend pour valeur la position du label L :

	Avant	Après
`accu`	0	0
`pc`	_	$position(L)$

   — Sinon, `pc` est incrémenté normalement :

	Avant	Après
`accu`	1	1
`pc`	n	$n+1$

5. **PUSH** (Empilement)
   Description :
   — Empile dans `stack` la valeur située dans `accu` :

	Avant	Après
`stack`	$[\_]$	$[x;\_]$
`accu`	x	x

6. **POP** (Dépilement)
   Description :
   — Dépile la valeur située en tête de `stack` :

	Avant	Après
`stack`	$[x;\_]$	$[\_]$

7. **ACC i** (Accès à la i-ième valeur de la pile)
   Description :
   — `stack[i]` est mis dans `accu` :

	Avant	Après
`stack`	$[a_0; a_1; a_2; ...; a_i; ...]$	$[a_0; a_1; a_2; ...; a_i; ...]$
`accu`	_	a_i

8. **ENVACC i** (Accès à la i-ième valeur de l'environnement)
   Description :

— `env[i]` est mis dans `accu` :

	Avant	Après
`env`	$<e_0; e_1; e_2; ...; e_i; ...; e_n>$	$<e_0; e_1; e_2; ...; e_i; ...; e_n>$
`accu`	_	e_i

9. **CLOSURE L,n** (Création de fermeture)
   Description :
   — Si $n > 0$ alors l'accumulateur est empilé.
   — Puis, une fermeture dont le code correspond au label L et dont l'environnement est constitué de n valeurs dépilées de `stack` est créée et mise dans l'accumulateur.

   Par exemple, si $n > 0$:

	Avant	Après
`stack`	$[a_0; ...; a_{n-2}; a_{n-1}; ...]$	$[a_{n-1}; ...]$
`accu`	x	$\{\ position(L)\ ,\ <x; a_0; ...; a_{n-2}>\ \}$

10. **APPLY n** (Application de fonction)
    Description :
    — n arguments sont dépilés [4]
    — `env` puis `pc+1` sont empilés
    — les n arguments sont rempilés

    On se met dans le contexte de la fonction appelée :
    — `pc` reçoit le pointeur de code de la fermeture située dans `accu`
    — `env` reçoit l'environnement de la fermeture située dans `accu`

	Avant	Après
`stack`	$[a_0; ...; a_{n-1}; a_n; \_]$	$[a_0; ...; a_{n-1}; c+1; e; a_n; \_]$
`pc`	c	c'
`env`	e	e'
`accu`	$\{\ c'\ ,\ e'\ \}$	$\{\ c'\ ,\ e'\ \}$

11. **RETURN n** (Sortie de fonction)
    Description :
    — n valeurs sont dépilées

    On retourne dans le contexte de l'appelant :
    — les valeurs associées à `pc` et `env` sont dépilées

	Avant	Après
`stack`	$[a_0; ...; a_{n-1}; p; e; a_n; \ ...]$	$[a_n; ...]$
`env`	_	e
`pc`	_	p

12. **STOP** (Fin de programme)
    Description :
    — Fin de l'exécution du programme
    — La valeur calculée par le programme est alors située dans `accu`

4. pour le moment, on ne traitera que les fonctions unaires

Fichiers de test

Vous disposez dans l'archive qui vous est fournie de fichiers vous permettant de tester votre implémentation. Pour cette partie, ces fichiers se trouvent dans le dossier `unary_funs`. Chaque fichier "bytecode" (extension .txt) est fourni avec une représentation en syntaxe OCaml du programme à tester [5].

Par exemple, considérons le programme `fun1.ml` :

```
let f x = 1 + x in (f 4) * 2
```

Le contenu du fichier bytecode correspondant, `fun1.txt`, est :

```
⟶BRANCH␣L2
L1:⟶ACC␣0
⟶PUSH
⟶CONST␣1
⟶PRIM␣+
⟶RETURN␣1
L2:⟶CLOSURE␣L1,0
⟶PUSH
⟶CONST␣2
⟶PUSH
⟶CONST␣4
⟶PUSH
⟶ACC␣2
⟶APPLY␣1
⟶PRIM␣*
⟶POP
⟶STOP
```

Et l'interprétation de ce fichier produit la suite suivante d'états :

```
au début : pc=0 accu=0 stack=[] env=<>
BRANCH L2          -> pc=6 accu=0 stack=[] env=<>
L2: CLOSURE L1,0   -> pc=7 accu={ L1, <> } stack=[] env=<>
PUSH               -> pc=8 accu={ L1, <> } stack=[{ L1, <> }] env=<>
CONST 2            -> pc=9 accu=2 stack=[{ L1, <> }] env=<>
PUSH               -> pc=10 accu=2 stack=[2;{ L1, <> }] env=<>
CONST 4            -> pc=11 accu=4 stack=[2;{ L1, <> }] env=<>
PUSH               -> pc=12 accu=4 stack=[4;2;{ L1, <> }] env=<>
ACC 2              -> pc=13 accu={ L1, <> } stack=[4;2;{ L1, <> }] env=<>
APPLY 1            -> pc=1 accu={ L1, <> } stack=[4;14;<>;2;{ L1, <> }] env=<>
L1: ACC 0          -> pc=2 accu=4 stack=[4;14;<>;2;{ L1, <> }] env=<>
PUSH               -> pc=3 accu=4 stack=[4;4;14;<>;2;{ L1, <> }] env=<>
CONST 1            -> pc=4 accu=1 stack=[4;4;14;<>;2;{ L1, <> }] env=<>
PRIM +             -> pc=5 accu=5 stack=[4;14;<>;2;{ L1, <> }] env=<>
RETURN 1           -> pc=14 accu=5 stack=[2;{ L1, <> }] env=<>
PRIM *             -> pc=15 accu=10 stack=[{ L1, <> }] env=<>
POP                -> pc=16 accu=10 stack=[] env=<>
STOP
```

7 Fonctions récursives

Ajoutons désormais la gestion des fonctions récursives. Pour qu'une fonction puisse faire appel à elle-même, elle doit avoir accès à un pointeur vers la première instruction de son code.

5. À ce propos, les fichiers bytecode fournis ressemblent à ce qu'affiche le compilateur `ocamlc` lorsqu'on lui donne l'option `-dinstr`

Vous devez ajouter alors deux nouvelles instructions :

1. L'instruction `CLOSUREREC L,n` permet de créer une fermeture récursive. Elle est très semblable à l'instruction `CLOSURE L,n`, à ceci près qu'elle stocke également en tête de l'environnement de la fermeture créée le pointeur de code correspondant au label L (afin de pouvoir se rappeler elle-même). De surcroît, pour faire comme dans la ZAM, la fermeture créée sera empilée à la fin de l'instruction.
2. L'instruction `OFFSETCLOSURE` qui met dans `accu` une fermeture dont le code correspond au premier élément de l'environnement courant (`env[0]`) et l'environnement correspond à l'ensemble de valeurs contenues dans `env`.

Fichiers de test

Des fichiers de test associés aux fonctions récursives sont situés dans le sous-dossier `rec_funs`.

8 Fonctions n-aires et application partielle

Il est possible de compiler une fonction d'arité n grâce à un processus de curryfication [6] qui consiste à transformer une fonction à n arguments en une fonction à un argument qui retourne une fonction à $n-1$ arguments. Par exemple, la fonction `(fun x y z -> x + y + z)` peut être réécrite (en curryfiant toutes les fonctions) en :

```
(fun x -> (fun y -> (fun z -> x + y + z)))
```

Cependant, ce procédé est assez coûteux, car avec notre mode d'application des fonctions, il serait alors créé n fermetures pour une fonction $n-$aire. Et ce surcoût semble encore plus inutile quand on réalise que, la plupart du temps, les fonctions sont appliquées avec tous leurs arguments.

Pour pallier à ce défaut, nous gérons alors l'application d'une fonction à n arguments en ajoutant à la machine *Mini-ZAM* un registre `extra_args` dont le rôle est de représenter le nombre d'arguments restant à appliquer à une fonction pour réaliser une application totale. Ce registre fera alors partie de l'état de la machine virtuelle, qui contiendra désormais 6 éléments.

Le contexte stocké lors de l'appel d'une fonction contiendra donc un élément supplémentaire. En effet, on aura besoin, en plus du `pc` et de l'environnement, de la valeur de `extra_args` pour enregistrer toutes les informations du contexte de l'appelant. Le mécanisme d'application de fonction est alors modifié en conséquence :

1. L'instruction **APPLY n** empilera `extra_args` en plus de `pc` et `env`, et mettra ensuite la valeur $n-1$ dans `extra_args`.
2. L'instruction **RETURN n** est modifiée comme suit :
   - Dans tous les cas, on dépile d'abord n valeurs.
   - Puis, si `extra_args` $= 0$, on conserve le fonctionnement précédent (mais on fera attention à bien se remettre dans le contexte de l'appelant en tenant compte des modifications apportées à `APPLY`)
   - Sinon :
     - `extra_args` est décrémenté de 1.
     - `pc` reçoit le pointeur de code de la fermeture située dans `accu`.
     - `env` reçoit l'environnement de la fermeture située dans `accu`.

   Par exemple, si `extra_args` > 0 :

6. `https://fr.wikipedia.org/wiki/Curryfication`

	Avant	Après
`stack`	$[a_0; a_1; ...; a_{n-1}; a_n; \ldots]$	$[a_n; \ldots]$
`extra_args`	m	$m-1$
`pc`	_	c
`env`	_	e
`accu`	$\{\ c\ ,\ e\ \}$	$\{\ c\ ,\ e\ \}$

De plus, deux nouvelles instructions seront à traiter :

1. **GRAB n** (Gestion de l'application partielle)
   Description :
   - Si `extra_args` $\geq n$, alors on a assez d'arguments pour appliquer la fonction : décrémenter `extra_args` de n et continuer à l'instruction suivante.

	Avant	Après
`extra_args`	$m+n$	m
`pc`	c	$c+1$

   - Sinon, on génère une nouvelle fermeture :
     - Dépiler `extra_args`+1 éléments
     - Mettre dans `accu` une nouvelle fermeture $\{c', e'\}$ telle que :
       - c' = `pc`-1 (i.e une instruction `RESTART`)
       - $e'[0]$ = `env`
       - les autres éléments de e' correspondent aux `extra_args`+1 éléments qui ont été dépilés juste avant.
     - Trois valeurs sont dépilées et associées à `extra_args`, `env`, et `pc`.

	Avant	Après
`stack`	$[a_0; a_1; ...; a_{m-1}; a_m; a_{m+1}; a_{m+2}; a_{m+3}; a_{m+4}; \ldots]$	$[a_{m+4}; \ldots]$
`extra_args`	m	a_{m+1}
`pc`	c	a_{m+2}
`env`	e	a_{m+3}
`accu`	_	$\{\ c-1\ ,\ <e; a_0; a_1; ...; a_m>\ \}$

2. **RESTART**
   Description : Cette instruction précédera toujours une instruction `GRAB`
   - Soit n, la taille de l'environnement
   - Déplacer les éléments de env à partir du deuxième (donc de `env[1] à env[`$n-1$`]`) dans la pile
   - `env` prend pour valeur celle de son premier élément (`env[0]`)
   - `extra_args` est incrémenté de $(n-1)$.

	Avant	Après
`stack`	$[a_0; ...]$	$[e_1; ...; e_{n-1}; a_0; ...]$
`extra_args`	m	$m+(n-1)$
`env`	$<e_0; e_1; ...; e_{n-1}>$	e_0

Fichiers de test

Des fichiers de test associés aux fonctions n-aires sont situés dans le sous-dossier `n-ary_funs`.

9 Optimisations et nouvelles fonctionnalités

Pour étendre le fonctionnement de votre *Mini-ZAM*, nous vous demandons finalement d'ajouter *au moins* une des fonctionnalités décrites dans cette section.

9.1 Appels terminaux

Il arrive assez souvent que l'appel à une fonction soit la dernière chose qui est réalisée dans la fonction appelante. Ainsi, une instruction `APPLY` précède régulièrement une instruction `RETURN`, et il est inutile (et coûteux) de sauvegarder le contexte de l'appelant dans la pile et le restaurer après l'appel pour simplement en sortir juste après.

De ce fait, nous vous proposons d'ajouter une nouvelle instruction **APPTERM n,m** dont le comportement est identique à la suite d'instructions `APPLY n; RETURN m-n`, mais sans passer par une sauvegarde et restauration de la fonction appelante.

Le déroulé d'une instruction `APPTERM n,m` est le suivant :

— Dépilement des n arguments de l'appel.

— Dépilement des $m - n$ variables locales de la fonction appelante.

— Rempilement des n arguments de l'appel.

— Positionnement dans le contexte de la fonction appelée.

— Incrémentation de `extra_args` de $n - 1$.

Ajoutez dans votre implémentation le code d'interprétation de la nouvelle instruction `APPTERM`, ainsi qu'une passe de transformation du bytecode du programme qui remplace toutes les occurences de `APPLY + RETURN` par le `APPTERM` correspondant.

Fichiers de test : Des fichiers de test se situent dans le sous-dossier `appterm`. Tous ne contiennent pas d'instruction `APPTERM` : il faudra bien d'abord utiliser la passe de transformation du bytecode pour générer des instructions `APPTERM`. Vous comparerez la taille maximale de la pile avec et sans cette optimisation.

9.2 Blocs de valeurs (potentiellement mutables)

Les programmes traîtés jusqu'ici manipulent uniquement des valeurs immédiates entières. Nous vous proposons d'ajouter désormais un nouveau type de valeurs, que nous nommerons *bloc*, et qui représente une valeur allouée sur le tas qui contient plusieurs valeurs. Nous représenterons un bloc entre parenthèses, avec ses différentes valeurs séparées par des virgules.

Par exemple, le bloc $(1, 2)$ représente une paire constituée de l'entier 1 et de l'entier 2.

La liste `1::2::3::4::[]` sera, quant à elle, représentée par le bloc `(1,(2,(3,(4,0))))` (la valeur 0, ici placée en seconde position d'un bloc, représente la liste vide).

Le tableau `[| 10 ; 20 ; 30 ; 40 |]` sera représenté par le bloc suivant : $(10, 20, 30, 40)$.

Pour manipuler de tels blocs, vous aurez à ajouter plusieurs instructions :

1. **MAKEBLOCK n** qui crée un bloc de taille n. Si $n > 0$ alors la valeur présente dans `accu` constitue le premier élément du bloc, tandis que les $n-1$ éléments suivants sont dépilés de `stack`. À la fin de l'instruction, l'accumulateur contiendra le nouveau bloc créé.
2. **GETFIELD n** qui met dans l'accumulateur la n-ième valeur du bloc contenu dans `accu`.
3. **VECTLENGTH** qui met dans l'accumulateur la taille du bloc situé dans `accu`.
4. **GETVECTITEM** qui dépile un élément n de `stack` puis met dans `accu` la n-ième valeur du bloc situé dans l'accumulateur.

5. On pourra alors traiter des valeurs mutables (comme les références ou les tableaux) avec les instructions suivantes :
   — **SETFIELD n** qui met dans la n-ième valeur du bloc situé dans `accu` la valeur dépilée de `stack`.
   — **SETVECTITEM** qui dépile deux éléments n et v de `stack`, puis met v dans la n-ième valeur du bloc situé dans `accu`. La valeur représentant `()` est ensuite mise dans l'accumulateur.
   — **ASSIGN n** qui remplace le n-ième élément à partir du sommet de la pile par la valeur située dans `accu`, puis met la valeur `()` dans l'accumulateur.

Fichiers de test : Des fichiers de test pour les blocs de valeurs sont situés dans le sous-dossier `block_values`.

9.3 Gestion des exceptions

Les exceptions représentent un mécanisme puissant pour gérer la présence d'anomalies ou autres conditions exceptionnelles au cours de l'exécution d'un programme. Les systèmes de gestion d'exception permettent alors de modifier le flot de contrôle d'un programme afin de traiter de manière spécifique de telles conditions.

Dans la *Mini-ZAM*, une exception sera représentée par un nombre entier, et l'ajout d'un système permettant de gérer des exceptions reposera sur les modifications suivantes :

1. Ajout d'un nouveau registre dans la machine virtuelle, nommé `trap_sp`. Ce registre représentera la position du rattrapeur d'exceptions situé au plus haut dans la pile.
2. Ajout de l'instruction **PUSHTRAP L** qui empile un nouveau récupérateur d'exception. Ses effets sur les registres de la machine virtuelle sont les suivants :
   — Empile `extra_args`, `env`, `trap_sp` et $position(L)$
   — Met dans `trap_sp` un pointeur vers l'élément en tête de pile
3. Ajout de l'instruction **POPTRAP** qui sort du récupérateur d'exception courant. Ses effets sur les registres de la machine virtuelle sont les suivants :
   — Dépile un élément, puis un second dont la valeur est mise dans `trap_sp`, puis deux autres éléments.
4. Ajout de l'instruction **RAISE** qui lève l'exception contenue dans l'accumulateur :
   — Si aucun récupérateur n'est défini, alors le programme s'arrête et le numéro de l'exception est affiché.
   — Sinon, on remet la pile dans l'état où elle était au dernier `PUSHTRAP` (i.e. on repositionne le sommet de pile à l'endroit pointé par `trap_sp`), et on récupère `pc`, `trap_sp`, `env` et `extra_args` à partir de quatre éléments dépilés de `stack`.

Fichiers de test : Des fichiers de test concernant la gestion des exceptions sont situés dans le sous-dossier `exceptions`.