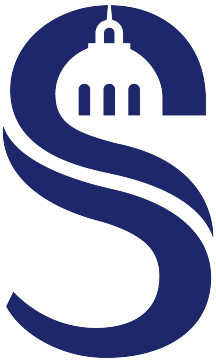

SORBONNE UNIVERSITÉ

Master Informatique

Rapport de projet
Compilation d'un langage impératif vers la Mini-ZAM

Adan Bougherara
Vivien Demeulenaere

Remis le 11 Mars 2022

Table des matières

1 Introduction **2**

2 Architecture du compilateur **2**

2.1 Parsing . . . 2

2.2 Arbre syntaxique . . . 2

2.3 Compilateur . . . 3

A Compilation du code . . . 3

B Normalisation du code . . . 3

2.4 Utilisation du compilateur . . . 3

3 Grammaire reconnue par le compilateur **3**

4 Schémas de compilation **5**

4.1 Programme . . . 5

4.2 Instructions . . . 5

A Affichage d'un entier . . . 5

B Bloc . . . 5

C Liaison locale . . . 6

D Modification physique d'une référence . . . 6

E Modification physique du contenu d'un tableau . . . 6

F Instruction conditionnelle . . . 7

G Boucle . . . 7

H Ruptures de calcul . . . 7

I Sortie de fonction . . . 8

4.3 Expressions . . . 8

A Invocation . . . 8

B Constantes . . . 9

C Variable . . . 9

D Accès au contenu d'un tableau . . . 9

E Application d'opérateur unaire . . . 10

F Application d'opérateur binaire . . . 10

G Allocation et initialisation d'une référence . . . 10

H Tableau . . . 11

I Déréférencement d'une référence . . . 11

J Taille d'un tableau . . . 11

4.4 Fonction globale . . . 11

4.5 Cas spéciaux . . . 11

5 Jeux de tests **12**

1 Introduction

Ce projet vise à compiler un langage impératif assez réduit vers des instructions exécutables par la Mini-ZAM, qui n'est autre qu'une implantation simplifiée de la machine virtuelle de OCaml. Pour se faire, notre compilateur codé en Java a été couplé à l'outil de Parsing ANTLR4. Il est également à noter que l'analyse syntaxique du programme source n'est pas l'objet de ce projet et qu'il est possible que certains programmes soient compilés mais que leur exécution ne se déroule pas comme prévu puisque le compilateur aura été trop permissif sur certains types par exemple.

2 Architecture du compilateur

Le compilateur implanté en Java est découpé en plusieurs packages dont l'architecture est détaillée ci-dessous.

2.1 Parsing

La première étape pour compiler un programme vers du bytecode de la Mini-ZAM consiste à lire le fichier d'entrée selon une grammaire spécifiée au préalable. Celle-ci se trouve dans le fichier `grammarCA.g4` contenu dans le package `grammar` du code source. A partir de cette grammaire, ANTLR4 permet de générer différentes interfaces dont `grammarCAListener` que nous avons du implanter afin de définir des règles à chaque fois que du texte répondant à la grammaire est reconnu.
Ceci nous a ainsi permit de générer un arbre syntaxique de type `IASTprogram` qui est ensuite traité par le compilateur.

2.2 Arbre syntaxique

L'approche adoptée pour mettre en oeuvre le compilateur s'appuie sur une organisation en arbre syntaxique. En effet, chaque règle définie dans la grammaire décrite plus loin engendre la création d'un noeud de cet arbre. Chacun de ces types de noeuds est décrit par une interface dans le package `interfaces` et son implantation concrète se trouve dans le package `ast`. Toutes ces interfaces étendent l'interface suivante :

```
package interfaces;

public interface IASTvisitable {
        <Result, Data, Anomaly extends Throwable>
        Result accept(IASTvisitor<Result, Data, Anomaly> visitor, Data data) throws Anomaly;
}
```

Cette interface a pour but de mettre en place le patron de conception *visiteur*.
Ainsi, tout noeud devra implanté la méthode `accept`. Cette dernière se contentera d'appeler la méthode `visit` du visiteur passé en argument avec l'ast lui même en paramètre afin d'utiliser la bonne méthode du visiteur. En voici l'implantation concrète :

```
@Override
public <Result, Data, Anomaly extends Throwable> Result
    accept(IASTvisitor<Result, Data, Anomaly> visitor,
                Data data) throws Anomaly {
        return visitor.visit(this, data);
}
```

2.3 Compilateur

Le compilateur situé le package `compiler` implante l'interface `IASTvisitor` qui contient une méthode `visit` pour chacun des types d'ast existant. La signature de cette classe est la suivante :

```
public class Compiler implements IASTvisitor<Void, LocalEnvironment, CompilationException>
```

Ainsi, les méthodes `visit` du compilateur ne renverront pas de résultats puisqu'elle se contenteront d'ajouter du texte à un objet `Writer`. De plus, elles prendront en argument en plus d'un ast, un environnement local qui pourra différer selon les schémas de compilation détaillés dans la suite du rapport. Enfin, ces dites méthodes pourront éventuellement lever une erreur de type `CompilationException` défini dans le package `tools`.

A Compilation du code

La méthode `compile` est le point d'entrée du compilateur. Elle prend en argument un `IASTprogram` et retourne le code compilé sous la forme d'une chaîne de caractères. Elle initialise également un environnement local vide de type `LocalEnvironment`. Celui-ci est passé en paramètre à la méthode `visit` avec le programme. On obtient ainsi une première version compilée du programme.

B Normalisation du code

Une fois le code compilé selon des schémas de compilation bien définis, il s'avère nécessaire d'effectuer une seconde passe sur le code afin d'effectuer une normalisation et ainsi rendre le bytecode interprétable par la Mini-ZAM. Cette passe sert notamment à éviter que deux étiquettes se suivent dans le code en retirant l'une d'elle et en renommant l'étiquette enlevée par celle que l'on garde dans les autres parties de code.

2.4 Utilisation du compilateur

Le compilateur prend obligatoirement en entrée un fichier contenant le code à compiler. L'extension de ce dernier est libre. De plus, il est possible de spécifier en second argument un fichier de sortie dans lequel sera écrit le code compilé. Il est à noter que si le fichier existe déjà, il est écrasé. Enfin, il est possible d'omettre cet argument, auquel cas le code obtenu sera affiché sur la sortie standard.

3 Grammaire reconnue par le compilateur

Le compilateur prend en entrée un programme contenant une instruction. Celle-ci peut elle même contenir plusieurs autres instructions. On parle alors de séquence d'instructions. Dans tous les cas, pour qu'un programme soit compilé, il faudra qu'il respecte la grammaire suivante :

$\pi ::=$	**Programme**
$\mid (f) * s$	
$s ::=$	**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
$\mid tab[e_1] := e_2$	Modification physique du contenu d'un tableau
$\mid$ **if** e **then** s (**else** s')?	Conditionnelle
$\mid$ **while** e **do** b **done**	Boucle
$\mid$ **continue**	Saut dans une boucle
$\mid$ **break**	Sortie de boucle
$\mid$ **return** e	Sortie de fonction
$\mid$ **skip**	Instruction ignorée

$b ::=$	**Bloc**
$\mid s$	Instruction
$\mid s\,;\, b$	Séquence

$e ::=$	**Expression**
$\mid$ **fun** f (e_1, ... e_n)	Invocation
$\mid c$	Constante
$\mid x$	Variable locale
$\mid x[e_2]$	Accès au contenu d'un tableau
$\mid \{e_1, e_2, ..., e_n\}$	Tableau
$\mid$ (**length** e)	Taille d'un tableau
$\mid$ ($\ominus$ e)	Application d'opérateur unaire
$\mid$ ($e1 \oplus e2$)	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**	Négation

$\oplus ::=$	**Opérateur binaire**
$\mid + \mid - \mid * \mid / \mid ==$	
$\mid < \mid > \mid <= \mid >=$	

$f ::=$	**Fonction globale**
$\mid$ **fun** f $(e_1 ... e_n) = b$	Définition

$c ::=$	**Commentaire**
$\mid$ //commentaire	Ligne de commentaire
$\mid$ /* commentaire */	Bloc de commentaire

4 Schémas de compilation

Cette partie vise à décrire le schéma de compilation de chacune des instructions reconnues par le compilateur. On notera ρ la pile, e l'environnement et k le pointeur de pile. La pile et l'environnement vide seront notés ϵ.

4.1 Programme

Le programme est l'arbre syntaxique abstrait d'entrée du compilateur. Il contient l'ensemble des fonctions définies globalement. De plus, il contient une instruction qui sera donc compilée avec une pile ainsi qu'un environnement vides et le pointeur de pile valant 0, d'où le schéma suivant :

$$\begin{aligned}\text{Prog}[\![\mathit{fonctions\ corps}]\!]_{\epsilon,\epsilon,0} = \ & \textbf{BRANCH L0} \\ & \text{Fonct}[\![\mathit{fonctions}]\!]_{\epsilon,\epsilon,0} \\ & \textbf{L0 :} \\ & \text{Instr}[\![\mathit{corps}]\!]_{\epsilon,\epsilon,0} \\ & \textbf{STOP}\end{aligned}$$

4.2 Instructions

A Affichage d'un entier

Le compilateur prend en charge l'affichage des entiers en utilisant la primitive de la mini-ZAM de la manière suivante :

$$\begin{aligned}\text{Instr}[\![\textbf{print } e]\!]_{\rho,e,k} = \ & \text{Expr}[\![e]\!]_{\rho,e,k} \\ & \textbf{PRIM } \texttt{print}\end{aligned}$$

Il s'agit donc de compiler dans un premier temps l'expression avant d'effectuer un appel à la primitive d'affichage.

B Bloc

Pour qu'un programme puisse contenir plusieurs instructions, il est possible de créer des blocs d'instructions. Ces blocs peuvent contenir une instruction et un bloc ou bien une simple instruction. Il y a donc deux cas à différencier.

B.1 Bloc avec une unique instruction

$$\text{Bloc}[\![i]\!]_{\rho,e,k} = \text{Instr}[\![i]\!]_{\rho,e,k}$$

B.2 Bloc avec une instruction et une autre séquence

$$\begin{aligned}\text{Bloc}[\![i, b]\!]_{\rho,e,k} = \ & \text{Instr}[\![i]\!]_{\rho,e,k} \\ & \text{Bloc}[\![b]\!]_{\rho,e,k}\end{aligned}$$

C Liaison locale

Dans le cas d'une liaison locale, il est nécessaire de compiler l'expression que l'on souhaite lier à la variable et mettre cette dernière sur le sommet de la pile. Ainsi, l'instruction contenue dans le corps de la liaison sera compilée avec une pile enrichie de cette variable. On s'assurera que cette dernière ait un nom unique pour que la portée de celle-ci soit prioritaire sur celle d'un argument de fonction ayant le même nom. Enfin, il conviendra de dépiler la variable.

$$\begin{aligned}\text{Instr}[\![\textbf{let } x = e \textbf{ in } i]\!]_{\rho,e,k} = \ & \text{Expr}[\![e]\!]_{\rho,e,0} \\ & \textbf{PUSH} \\ & \text{Instr}[\![i]\!]_{x::\rho,e,k} \\ & \textbf{POP}\end{aligned}$$

D Modification physique d'une référence

Il peut s'avérer nécessaire de modifier la valeur physique d'une référence existante ou bien de tout simplement l'initialiser. On distinguera donc deux schémas de compilation.

D.1 Initialisation d'une variable

L'initialisation d'une variable est très similaire à une liaison locale. Cependant, il s'agit d'une affectation globale, la valeur n'est donc jamais dépilée. Notons également, qu'effectuer une initialisation avec une expression autre qu'une référence déclenchera une erreur à la compilation.

$$\begin{aligned}\text{Instr}[\![x := e]\!]_{\rho,e,k} = \ & \text{Expr}[\![e]\!]_{\rho,e,0} \\ & \textbf{PUSH}\end{aligned}$$

D.2 Modification de la valeur d'une variable

Nous noterons $indice_x$ l'indice de la variable x dans la pile. Nous obtenons alors le schéma suivant :

$$\begin{aligned}\text{Instr}[\![x := e]\!]_{\rho,e,k} = \ & \text{Expr}[\![e]\!]_{\rho,e,k} \\ & \textbf{PUSH} \\ & \textbf{ACC } indice_x + 1 \\ & \textbf{SETFIELD 0} \\ & \textbf{ASSIGN } indice_x\end{aligned}$$

E Modification physique du contenu d'un tableau

Les tableaux sont alloués statiquement par le compilateur mais il est possible d'en modifier le contenu à partir de l'indice de la case à modifier. On notera $indice_{tab}$ l'indice de tab dans ρ.

$$\begin{aligned}\text{Instr}[\![tab[indice] := e]\!]_{\rho,e,k} = \ & \text{Expr}[\![e]\!]_{\rho,e,k} \\ & \textbf{PUSH} \\ & \text{Expr}[\![indice]\!]_{\rho,e,k+1} \\ & \textbf{PUSH} \\ & \textbf{ACC } indice_{tab} + 2 \\ & \textbf{SETVECTITEM}\end{aligned}$$

F Instruction conditionnelle

L'instruction conditionnelle s'écrit obligatoirement avec une condition et une conséquence, mais il est possible d'omettre l'alternative, d'où les deux schémas de compilation ci-dessous.

F.1 Instruction conditionnelle avec alternative

$$\begin{aligned}\text{Instr}[\![\textbf{if } e \textbf{ then } s \textbf{ else } s']\!]_{\rho,e,k} = \ & \text{Expr}[\![e]\!]_{\rho,e,k} \\ & \textbf{BRANCHIFNOT } L_{else} \\ & \text{Instr}[\![s]\!]_{\rho,e,k} \\ & \textbf{BRANCH } L_{end} \\ & \textbf{LABEL } L_{else} \\ & \text{Instr}[\![s']\!]_{\rho,e,k} \\ & \textbf{LABEL } L_{end}\end{aligned}$$

F.2 Instruction conditionnelle sans alternative

La compilation de l'instruction conditionnelle sans alternative est exactement la même que celle avec à l'exception que l'alternative est inexistante et elle n'est donc pas compilée. Le programme sera rendu interprétable par la mini-ZAM après la passe de normalisation qui enlèvera les étiquettes qui se suivent.

$$\begin{aligned}\text{Instr}[\![\textbf{if } e \textbf{ then } s]\!]_{\rho,e,k} = \ & \text{Expr}[\![e]\!]_{\rho,e,k} \\ & \textbf{BRANCHIFNOT } L_{else} \\ & \text{Instr}[\![s]\!]_{\rho,e,k} \\ & \textbf{BRANCH } L_{end} \\ & \textbf{LABEL } L_{else} \\ & \textbf{LABEL } L_{end}\end{aligned}$$

Il est à noter que les étiquettes L_{else} et L_{end} sont généreés de manière unique par le compilateur.

G Boucle

Comme dans le cas de l'alternative, les étiquettes L_{while} et L_{end}, présentes dans le schéma de compilation, sont générées de manière unique par le compilateur.

$$\begin{aligned}\text{Instr}[\![\textbf{while } e \textbf{ do } b \textbf{ done}]\!]_{\rho,e,k} = \ & \textbf{LABEL } L_{while} \\ & \text{Expr}[\![e]\!]_{\rho,e,k} \\ & \textbf{BRANCHIFNOT } L_{end} \\ & \text{Bloc}[\![b]\!]_{\rho,e,k} \\ & \textbf{BRANCH } L_{while} \\ & \textbf{LABEL } L_{end}\end{aligned}$$

H Ruptures de calcul

Les ruptures de calcul dénotées par les instructions **break** et **continue** suivent un schéma de compilation très similaire. En effet, il s'agit uniquement d'effectuer un saut inconditionnel.
Il est à noter qu'un appel à ces instructions en dehors d'une boucle déclenchera une erreur.

H.1 Saut dans une boucle

Notons L_{while} l'étiquette de début de la dernière boucle en cours de traitement.

$$\text{Instr}[\![\textbf{continue }]\!]_{\rho,e,k} = \textbf{BRANCH } L_{while}$$

H.2 Sortie de boucle

Notons L_{end} l'étiquette de fin de la dernière boucle en cours de traitement.

$$\text{Instr}[\![\textbf{break }]\!]_{\rho,e,k} = \textbf{BRANCH } L_{end}$$

I Sortie de fonction

Le schéma de compilation des invocations de fonctions globales défini plus bas nous assure que l'instruction **APPLY** est toujours appelée avec 1 en paramètre. Ainsi, il nous suffira de compiler l'expression suivant le mot-clef **return** et d'utiliser la commande **RETURN** de la Mini-ZAM.

$$\begin{aligned}\text{Instr}[\![\textbf{return } x]\!]_{\rho,e,k} = \ & \text{Expr}[\![x]\!]_{\rho,e,k} \\ & \textbf{RETURN 1}\end{aligned}$$

4.3 Expressions

A Invocation

Pour invoquer une fonction définie globalement, il convient de créer une fermeture encapsulant les arguments de la fonction dans un environnement dédié. Il s'agira ensuite d'effectuer un simple saut inconditionnel à l'endroit où est écrit le code de la fonction à l'aide de l'instruction **APPLY**. Une petite subtilité consiste à ajouter une constante dans la pile, car **APPLY** ne fonctionne qu'avec un nombre strictement positif d'arguments.
On notera L_f dans le schéma de compilation l'étiquette précédant ce code et $indice_f$ l'indice de la fermeture liée à l'invocation de f.
Naturellement, si la f n'est pas définie, une erreur sera levée.

$$\begin{aligned}\text{Expr}[\![\text{f (} arg_1, arg_2 \text{ ..., } arg_n\text{)}]\!]_{\rho,e,k} = \ & \text{Expr}[\![arg_1]\!]_{\rho,e,k} \\ & \textbf{PUSH} \\ & \text{Expr}[\![arg_2]\!]_{arg_1::\rho,e,k+1} \\ & \textbf{PUSH} \\ & \text{...} \\ & \text{Expr}[\![arg_n]\!]_{[arg_1 \ldots arg_{n-1}]@\rho,e,k+n-1} \\ & \textbf{CLOSURE } L_f \ n \\ & \textbf{PUSH} \\ & \text{Const}[\![1664]\!]_{\{L_f,<arg_1,\ldots,arg_n>\}::\rho,e,k} \\ & \textbf{PUSH} \\ & \textbf{ACC } indice_f \\ & \textbf{APPLY 1}\end{aligned}$$

B Constantes

Les constantes présentent dans le langage peuvent être des entiers ou bien des booléens. Dans tous les cas, elles seront interprétées comme des entiers par la Mini-ZAM.

$$\begin{aligned}
\text{Const}[\![n]\!]_{\rho,e,k} &= \textbf{CONST } n \\
\text{Const}[\![\textbf{true}]\!]_{\rho,e,k} &= \textbf{CONST } 1 \\
\text{Const}[\![\textbf{false}]\!]_{\rho,e,k} &= \textbf{CONST } 0
\end{aligned}$$

C Variable

Il existe deux cas de figure pour accéder à une variable. Soit elle est définie de manière locale dans un bloc, dans ce cas, on ira la chercher dans la pile, soit elle est définie comme une variable de fonction et elle se trouvera alors dans l'environnement associé à la fermeture de cette dernière.
Le compilateur cherchera dans un premier temps si elle se trouve dans l'environnement local avant de vérifier la pile. Enfin, si la variable n'est présente nulle part, une erreur sera levée.

C.1 Variable d'environnement

Supposons que $x \in e$ Notons $indice_x$ l'emplacement de x dans e.

$$\text{Expr}[\![x]\!]_{\rho,e,k} = \textbf{ENVACC } indice_x$$

C.2 Variable locale

Supposons que $x \notin e$ et $x \in \rho$. Notons $indice_x$ l'emplacement de x dans ρ.

$$\text{Expr}[\![x]\!]_{\rho,e,k} = \textbf{ACC } indice_x$$

D Accès au contenu d'un tableau

L'accès au contenu d'un tableau est similaire à l'accès à une variable simple. On devra donc là encore chercher la variable dans l'environnement e puis dans la pile ρ. Si le tableau n'est pas trouvé, une erreur sera déclenchée.

D.1 Tableau dans l'environnement

Supposons que $tab \in e$ Notons $indice_{tab}$ l'emplacement de tab dans e.

$$\begin{aligned}
\text{Expr}[\![tab[indice]]\!]_{\rho,e,k} = \ & \text{Expr}[\![indice]\!]_{\rho,e,k} \\
& \textbf{PUSH} \\
& \textbf{ENVACC } indice_{tab} + 1 \\
& \textbf{GETVECTITEM}
\end{aligned}$$

D.2 Tableau dans l'environnement

Supposons que $tab \notin e$ et que $tab \in \rho$ Notons $indice_{tab}$ l'emplacement de tab dans ρ.

$$\begin{aligned}\text{Expr}[\![tab[indice]]\!]_{\rho,e,k} = \ & \text{Expr}[\![indice]\!]_{\rho,e,k} \\ & \textbf{PUSH} \\ & \textbf{ACC } indice_{tab} + 1 \\ & \textbf{GETVECTITEM}\end{aligned}$$

E Application d'opérateur unaire

Le seul opérateur unaire disponible dans ce langage est l'opérateur booléen de négation. Cependant, ce dernier peut être utilisé avec n'importe quel entier sans déclencher d'erreur de compilation. Il faut tout de même noter que la valeur **0** sera interprétée comme **false** et tout autre valeur comme **true**.
D'autres opérateurs unaires pourront être implantés pour peu qu'ils possèdent une primitive dans la Mini-ZAM.

$$\begin{aligned}\text{Expr}[\![\ominus x]\!]_{\rho,e,k} = \ & \text{Expr}[\![x]\!]_{\rho,e,k} \\ & \textbf{PRIM } \ominus\end{aligned}$$

F Application d'opérateur binaire

A l'instar de l'opérateur unaire, les applications d'opérateurs binaires sont compilées à l'aide de primitives de la Mini-ZAM. Le seul opérateur faisant exception est `==` qui sera transformé en `=` au moment de la compilation.
Les opérations peuvent être parenthésées mais suivent les règles de priorités usuelles.

$$\begin{aligned}\text{Expr}[\![e_1 \oplus e_2]\!]_{\rho,e,k} = \ & \text{Expr}[\![e_2]\!]_{\rho,e,k} \\ & \textbf{PUSH} \\ & \text{Expr}[\![e_1]\!]_{\rho,e,k+1} \\ & \textbf{PRIM } \oplus\end{aligned}$$

G Allocation et initialisation d'une référence

L'allocation d'une référence consiste simplement à compiler l'expression associée à cette dernière et à allouer un bloc.

$$\begin{aligned}\text{Expr}[\![\textbf{(ref x)}]\!]_{\rho,e,k} = \ & \text{Expr}[\![x]\!]_{\rho,e,k} \\ & \textbf{MAKEBLOCK 1}\end{aligned}$$

H Tableau

Créer un tableau est très similaire à l'allocation d'une référence. Cependant, le compilateur doit cette fois-ci allouer n cases pour un tableau de taille n. On prendra également soin d'empiler le contenu du tableau dans l'ordre inverse.

$$\begin{aligned}\text{Expr}[\![\{e_1, e_2, ..., e_n\}]\!]_{\rho,e,k} = \ & \text{Expr}[\![e_n]\!]_{\rho,e,k} \\ & \textbf{PUSH} \\ & \text{Expr}[\![e_{n-1}]\!]_{\rho,e,k+1} \\ & \textbf{PUSH} \\ & ... \\ & \text{Expr}[\![e_1]\!]_{\rho,e,k+n-1} \\ & \textbf{MAKEBLOCK } n\end{aligned}$$

I Déréférencement d'une référence

Si un programme contient une instruction de déréférencement d'une variable non initialisée, le compilateur lèvera une erreur. Sinon, notons $indice_x$ l'indice de la variable x dans la pile.

$$\begin{aligned}\text{Expr}[\![\textbf{(! x)}]\!]_{\rho,e,k} = \ & \textbf{ACC } indice_x \\ & \textbf{GETFIELD 0}\end{aligned}$$

J Taille d'un tableau

Afin d'itérer sur les tableaux, il parait essentiel d'avoir une primitive sur ces derniers permettant de donner leur taille. En voici son schéma de compilation :

$$\begin{aligned}\text{Expr}[\![\textbf{(length } tab\textbf{)}]\!]_{\rho,e,k} = \ & \text{Expr}[\![tab]\!]_{\rho,e,k} \\ & \textbf{VECTLENGTH}\end{aligned}$$

Remarquons, que si *tab* n'est pas un tableau, aucune erreur de compilation ne sera déclenchée. En revanche, des erreurs à l'exécution peuvent avoir lieu.

4.4 Fonction globale

Compiler une fonction globale revient à écrire une étiquette unique à chaque fonction notée L_f et à compiler son corps. Ses arguments seront simplement ajoutés à l'environnement du corps.

$$\begin{aligned}\text{Fonct}[\![\textbf{fun } f\ (arg_1...arg_n)]\!]_{\rho,e,k} = \ & \textbf{LABEL } L_f \\ & \text{Bloc}[\![b]\!]_{\rho,<arg_n...arg_1>@e,k}\end{aligned}$$

4.5 Cas spéciaux

Certaines chaînes de caractères sont ignorées lors de l'étape de parsing. On peut notamment citer l'instruction spéciale **skip** qui ne possède donc pas de schéma de compilation. Enfin les espaces et les commentaires sont ignorés. Ces derniers sont délimités avec la suite de caractères `//` s'ils sont sur une ligne ou bien avec `/*` puis `*/` s'ils s'étendent sur plusieurs lignes.

5 Jeux de tests

Afin de tester les différents schémas de compilation énumérés plus haut, nous mettons à votre disposition différents jeux de tests.
Ces derniers ont pour but de donner quelques exemples d'utilisation du petit langage impératif mais aussi de tester son bon fonctionnement. Ils se trouvent dans le dossier `test` fourni avec le projet. Sous le code est écrit l'affichage qui doit se produire après exécution du code compilé.