

Rapport pour le projet LIEC: Compilateur Python vers MIPS en Racket

Panchalingamoorthy Gajenthiran

8 Janvier 2019

Table des matières

1	Introduction	1
2	Analyses	2
2.1	Analyse lexicale	2
2.2	Analyse syntaxique	4
2.3	Analyse sémantique	6
3	Compilation	8
3.1	mips-data	9
3.2	mips-loc	9
3.3	comp	9
3.3.1	Condition	10
3.3.2	Boucle	11
3.3.3	Fonctions	11
3.3.4	Appel de fonctions	12
3.3.5	<i>liec.rkt</i>	12
4	Améliorations à apporter	13
5	Conclusion	14

1 Introduction

Le projet consiste à réaliser un compilateur Python vers MIPS en Racket. Cependant, j'ai plutôt opté pour créer un compilateur C. Pour réaliser ce projet, je suis parti des fichiers de la séance 7 ("Écrivons

ensemble un interpréteur") et du fichier *minicomp.rkt* de la séance f ("La mémoire").

Pour exécuter notre programme, il suffit de se diriger vers le dossier *compiler* et d'entrer la commande suivante :

```
racket liec.rkt <fichier.liec>
```

Remarque : Les programmes passés en paramètre ont tous l'extension *.liec*. Pour pouvoir tester mon compilateur vous pouvez utiliser les fichiers *.liec* qui se trouve dans le dossier *test*. Cette commande vous affichera seulement le code en MIPS. Pour pouvoir le compiler, il suffit de le rediriger vers un fichier MIPS (*.s*) à l'aide de :

```
racket liec.rkt <fichier.liec> > test.s et de le compiler avec : spim -f test.s.
```

Avant de compiler le programme passé en paramètre, je vais d'abord faire plusieurs analyses du code.

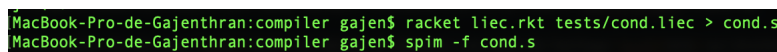


FIGURE 1 – Commande pour exécuter le compilateur

2 Analyses

Afin de vérifier la validité du code, le programme va être analysé lexicalement, syntaxiquement et sémantiquement avant d'être compilé.

2.1 Analyse lexicale

L'analyse lexicale découpe le texte du code source en une suite de lexèmes. Il permet de vérifier les différents "mots" de notre langage. Pour définir les différents mots de notre langage, nous allons utiliser l'outil **lexer** (**lex**) en Racket. Tout ce qui concerne l'analyse lexicale se déroulera dans le fichier *lexer.rkt*. Nous allons donc définir des tokens fixes (des lexèmes qui ont des valeurs fixes, comme les symboles) à l'aide de **define-empty-tokens** et des tokens variables (dont on connaît pas la valeur, comme les nombres, les chaînes de caractères, les booléens) à l'aide de **define-tokens**.

Le code de la séance 7 fournit une grande panoplie de lexèmes. Je vais donc pas vous détailler tous les lexèmes afin de ne pas vous ennuyer.

Tout d'abord nous allons définir des **empty-tokens** pour les opérateurs : des opérateurs arithmétiques/logiques ("**+**", "**-**", "*****", "**/**", "**AND**", "**OR**"...) ou des opérateurs de comparaison ("**>**", "**<**", "**==**", "**!=**" ...). Comme **empty-tokens**, il y a également les mots-clefs du programme ("**if**", "**else**",

"while", "return", "rec"...), les signes de ponctuations (";", "(", ")", ":", "..."). Une fois les symboles de notre programme fini, on va devoir définir les expressions variables grâce à des regexp, nous avons :

- les chaîne de caractères (`[a-zA-Z]+` avec possibilité de mettre des tirets pour suivre la convention Snake case)
- les nombres (`[0-9]+` avec possibilité de mettre un tiret pour représenter les nombres négatives)
- les valeurs booléennes (soit la valeur `true`, soit la valeur `false`)
- les types (soit `int`, `str`, `bool` ou `nil`)

Cependant, il y a de nombreux lexèmes à rajouter et à modifier par rapport à la séance 7 vu qu'il s'agit d'un programme en C. Voici une liste des différentes modifications :

- rajout des commentaires longs (comme en C qui débute avec `/*` et se termine par `*/`)
- remplacement de `begin` et `end` par `{` et `}` pour les blocs d'instructions
- remplacement de `not`, `and` et `or` qui existe en Python par des symboles qui correspondent mieux en C à savoir `!`, `&&` et `||`
- représentation des listes à l'aide de `{` et `}` au lieu de `[` et `]` (cependant j'ai décidé de laisser les crochets pour de futures implémentations).
- rajout du signe `-` pour les nombres négatifs

```

(define-empty-tokens keywords
  (Lrec Lreturn
   Lif Lthen Lelse
   Lwhile
   Lnil
   Leof))

(define-empty-tokens operators
  (Leq Lneq Llt Lgt Llte Lgte Lassign
   Ladd Lsub Lmul Ldiv Lmod
   Land Lor Lnot Lxor))

(define-empty-tokens punctuations
  (Lsc Lopar Lcpar Lobra Lcbra
   Locbra Lccbra Lcol Lcom Llist))

(define-tokens atoms
  (Lident Lnum Lstr Lbool Ltype))

;; regexp abbreviations
(define-lex-abbrev latin_
  (:or (char-range "a" "z")
       (char-range "A" "Z")
       "_"))

(define-lex-abbrev latin_num
  (:or latin_ numeric))

```

FIGURE 2 – Déclarations des `tokens`

Une fois l'analyse lexicale terminée, nous pouvons utiliser les tokens générés lors de l'analyse lexicale pour l'analyse syntaxique.

2.2 Analyse syntaxique

L'analyse syntaxique permet de comprendre et de vérifier la structure du code. Cette partie va définir la structure de notre code (définir sa grammaire). Pour cela, l'outil `parser` (`yacc`) en Racket nous sera d'une grande utilité. En effet, cet outil va nous permettre, à l'aide nos tokens définis auparavant, de créer notre grammaire et dessiner la structure de notre code (on définira également les priorités opératoires, le symbole de départ et de fin).

Encore une fois, le `parser` de la séance 7 est plutôt fourni en terme de contenu. Il y a néanmoins certains ajustements à faire pour avoir une structure syntaxique proche du langage C. Notre programme s'agit d'un ensemble de définitions. Une définition représente soit une re/définition de variable, soit une déclaration de fonction (récursive ou non). Pour réaliser ces défini-

tions, on doit donc définir ce qu'est une expression (pour les variables), un bloc d'expression (pour les fonctions), les arguments (pour les fonctions).

Nous n'allons pas expliquer toute la grammaire du langage (pour ne s'éterniser sur ce sujet, surtout que ce dernier ressemble au votre) mais nous allons quand même lister l'ensemble des différences notables par rapport à votre code :

- Les fonctions ne prennent plus de **block** mais un **fblock** comme bloc d'instructions. Contrairement au **block**, le **fblock** retourne une valeur. Dans votre code, le **block** retournerait forcément une valeur que cela soit un bloc de fonction ou de condition. Ici, le bloc ne retourne rien, ce qui est parfois problématique mais traiterons cela dans une prochaine section.
- De plus la structure d'une fonction a été complètement remaniée, nous indiquerons le type en premier, et les arguments seront entre parenthèses. Cependant, je n'ai pas réussi à totalement dessiner la structure des arguments. En effet, je dois écrire `[type1], [type2] : [nom1], [nom2]` où en C on devrait écrire `[type1] [nom1], [type2] [nom2]`. Par exemple, pour la fonction `main` cela donnerait `int main(int, str list : argc, argv)` au lieu de `int main(int argc, str argv)`.
- Comme vous aurez remarqué dans l'exemple ci-dessus, un autre problème est le type **str** qui reviendrai à représenter un **char** et un **char*** (il n'y a pas de distinctions entre les deux avec **str**). J'ai décidé de laisser ce type là pour me faciliter la tâche avec **str** sans forcément créer de **char** (et de gérer la mémoire). De même pour le type booléen (**bool**) qui n'est pas en C, mais cela m'a semblé intéressant de le mettre (le **false** représentera la valeur 0, le **true** la valeur 1). De toute manière, ces modifications là auraient du être faites lors de l'analyse lexicale.
- l'ajout d'une boucle qui aura la forme suivante : `"while" "(" sexpr ")" expr`. Une **sexpr** est une expression singulière qui peut prendre la forme d'une opération, d'un appel de fonction, d'une opération ou d'une **sexpr** entre parenthèses tandis qu'une **expr** peut être une définition, un appel de fonction, une condition, une boucle ou encore un bloc d'expressions. Ainsi l'**expr** offre plus de possibilité qu'une expression singulière, c'est ça qui sera utilisé pour les conditions, pour les boucles et les blocs de fonctions (en réalité les blocs de fonction sont des **exprs** qui correspondent à un ou plusieurs **expr** à la suite).

Ainsi à la fin de cette analyse, nous obtiendrons l'ensemble des structures syntaxiques nécessaires pour reconnaître notre code. Cette structure va nous servir pour notre analyse sémantique. Les structures grammaticales (les dé-

finitions, les blocs d'instructions, les conditions, les boucles ou encore les appels de fonction) seront définies dans le fichier *ast.rkt* afin de pouvoir les réutiliser prochainement (et également d'avoir un aperçu de notre arbre).

Remarque : J'ai décidé de garder la même convention que votre code en mettant un "P" devant les structures syntaxiques (pour les structures parsées : Pvardef, Pfundef, Pcond...) et un "L" devant les tokens.

```
(definition
  ((type Lident Lassign sexpr Lsc) (Pvardef $2 $4 $1 (sp 2)))
  ((Lident Lassign sexpr Lsc) (Pvar $1 $3 (sp 2)))
  ((type Lident Lopar argtypes Lcol fargs Lcpar fblock) (Pfundef #f $2 $6 $8 (Fun $1 $4) (sp 2)))
  ((Lrec type Lident Lopar argtypes Lcol fargs Lcpar fblock) (Pfundef #t $3 $7 $9 (Fun $2 $5) (sp 3))))
```

FIGURE 3 – Création de notre grammaire

```
;;; Définition d'une variable
;; type id "=" expr
(struct Pvardef (id expr type pos) #:transparent)

;;; Rédefinition d'une variable
;; id "=" expr
(struct Pvar (id expr pos) #:transparent)

;;; Définition d'une fonction
;; ( "rec" )? type-ret id "(" args ":" type-args ")" body
(struct Pfundef (rec id args body type pos) #:transparent)
```

FIGURE 4 – Création d'un arbre syntaxique en relation avec la grammaire créée ci-dessus

2.3 Analyse sémantique

L'analyse sémantique va nous permettre de vérifier le sens de notre code à savoir le bon nombre d'arguments, le type ou encore la portée des variables, pour à la fin produire un arbre de syntaxe abstraite (AST) qui va être utilisé lors de la compilation. Mais avant d'analyser sémantiquement le code, on va d'abord définir, dans le fichier *stdlib.rkt*, une table de hachage pour vérifier les opérations et les fonctions natives. En effet, les opérations sont considérées comme des fonctions retournant un type et prenant des arguments. Par exemple, l'opérateur "+" prend en argument une liste d'entiers et un entier et retourne un entier, ou encore l'opérateur "==" prend une liste d'entiers et un entier en argument et retourne une valeur booléenne (ce qui veut donc dire que nous ne pouvons pas comparer deux chaînes de caractères).

Ainsi nous commencerons par la fonction `liec-check` dans le fichier *semantics.rkt* qui vérifiera la présence d'une fonction `main` (sinon quittera le programme et affichera un message d'erreur) et vérifiera l'ensemble du programme à l'aide de la fonction `check-exprs` qui prend en paramètre le programme (qui est en réalité la structure syntaxique que nous venons de créer), un environnement et un type attendu. L'environnement, ici correspond à la table de hachage que nous avons défini dans *stdlib.rkt*.

La fonction `check-exprs` va vérifier au fur à mesure une expression à l'aide de `check-expr` (qui possède les mêmes arguments). `check-expr` va utiliser `match` pour vérifier la structure de l'expression passée en argument. Si la structure de l'argument ressemble à une des structures syntaxiques définies précédemment (`Pvardef`, `Pfundef`, `Pcond...`). Une fois l'expression matchée, nous vérifierons si il y a pas de définitions dupliquées, d'argument en trop d'arguments lors des appels de fonctions, des identifiants non définies... Si les expressions sont correctes et trouvent preneur alors il renvoie une liste de deux éléments avec d'un côté la structure pour l'arbre de syntaxe abstraite (AST) qu'on crée et d'un autre côté l'environnement qui nous permettra de stocker les variables et les fonctions déclarées. Cependant dans la fonction, nous enverrons seulement les structures de l'AST pour le compilateur (nous n'enverrons pas l'environnement, cela nous a seulement servi pour les vérifications).

Ici, je n'ai pas réalisé de grosses modifications. Vu que j'ai rajouté une boucle et le bloc d'une fonction, je dois donc introduire cela donc mon `check-exprs`. Pour les blocs, j'ai dû changer le comportement de mon bloc `Pblock` et mon bloc de fonction `Pfunblock`. Les deux vérifient un ensemble d'instructions donc avec `check-exprs` sauf que `Pfunblock` vérifie également la valeur de retour (qui doit être identique au type de la fonction). Par rapport à votre code, `Pblock` n'attend pas de type (j'ai mis `Any` pour dire qu'on peut mettre ce que l'on veut et qu'on a pas besoin de vérifier le type de la dernière instruction car en réalité dans `check-exprs` on vérifie l'ensemble des expressions sans se soucier du type mais on vérifie seulement la dernière expression. Ici en mettant `Any` on ne vérifie même pas la dernière expression).

Pour les boucles, je vérifie le "test" si la valeur de retour est une valeur booléenne, puis je vérifie l'expression (ou bloc d'expressions) du corps de la boucle (un peu de la même façon que le bloc d'instructions). Ensuite j'envoie les expressions de la boucle (test + corps de la boucle) comme premier élément et l'environnement comme second élément de la liste.

Il y a également d'autres fonctions dans le fichier *semantics.rkt* comme les messages d'erreur (`err`, `fail`, `errt`).

```

((Pvardef id expr type sp)
 (if (hash-has-key? env id)
      (err sp "~a: duplicate definition" id)
      (if (not (type-compat? expected-type Nil))
           (errt sp expected-type Nil)
           (let ((expr (check-expr expr env type)))
               (cons (Let id (car expr))
                     (hash-set env id type)))))))

```

FIGURE 5 – Vérifier la déclaration d’une variable et retourner une structure pour l’AST

```

;;; Re/Definition
;; Lier l'identifiant <n> à <v>.
(struct Let (n v) #:transparent)

```

FIGURE 6 – l’AST avec la structure Let

```

int main(int, str list : argc, argv) {
  int b = 10;
  return 0;
}

```

Exemple de notre programme

```

(#(struct:Func main
  #(struct:Closure #f (argc argv)
    #(struct:Funblock
      (#(struct:Let b #(struct:Const 10))
        #(struct:Let is #(struct:Const #t))
        #(struct:Const 0)) #f)))

```

AST renvoyé pour ce programme

FIGURE 7 – L’AST renvoyé pour le programme

3 Compilation

Une fois l’AST créé, on va devoir compiler l’ensemble des expressions à l’aide de cette dernière. On va donc se diriger vers le fichier *compiler-mips.rkt* qui lui-même a besoin du fichier *mips.rkt* et *ast.rkt*. Le fichier *mips.rkt* va rassembler l’ensemble des structures représentant les instructions MIPS qui vont ensuite nous aider pour la fonction `mips-emit` du fichier *compiler-mips.rkt*. En effet, `mips-emit` va se charger de vérifier la correspondance entre une instruction (passée en paramètre) et les structures créées dans *mips.rkt* (avec

`match`). Dès qu'on `match` une expression, on affiche sur le terminal (`printf`) l'instruction MIPS correspondante. Par exemple, si on `match` (`Move rd rs`), on écrira `"move $RS $RD"` où `RS` correspond à la valeur dans `rs` et `RD` correspond à la valeur dans `rd`. Quant au fichier *ast.rkt*, en plus de contenir, les structures de l'analyse syntaxique, il contient également l'AST qui nous permettra d'afficher les instructions MIPS.

3.1 mips-data

La fonction `mips-data` va afficher la section de données et la déclaration des variables dans un premier temps puis afficher la section de code, la déclaration de `main` comme global et le point d'entrée (qui est `main:`). Elle prend en paramètre une table de hachage et va parcourir l'ensemble des éléments de la table et l'afficher dans la section de données. Ainsi l'idée est de créer une table de hachage vide qui s'agrandira au fur et à mesure de la lecture du programme (ou plutôt de l'AST du programme), on rajoutera à chaque fois que l'on voit une chaîne de caractère, créer une variable dans la section de données.

3.2 mips-loc

`mips-loc` joue le même rôle que `mips-emit` et se charge d'afficher des instructions MIPS concernant les adresses.

3.3 comp

C'est dans cette fonction là qu'on compile les expressions. En effet, `comp` prend en paramètre l'expression à compiler, l'environnement et le décalage entre `sp` et `fp`. Nous allons agir de la même manière que le fichier `semantics.rkt` ou encore `eval.rkt` de la séance 7 en créant une liste de deux éléments avec d'un côté l'ensemble des instructions à compiler et l'environnement du programme. Au départ, je comptais faire avec `cons` pour concaténer l'instruction MIPS obtenue et l'environnement mais finalement un camarade de classe (Nabil Boutemour) m'a plutôt conseillé d'utiliser les `list` vu que c'est plus simple au lieu de `cons`. Ainsi, nous aurons toujours des `list` de deux éléments avec comme première élément l'instructions MIPS et comme second élément l'environnement (et du coup c'est plus en adéquation avec ce que vous utiliser dans votre *minicomp.rkt* vu que vous utiliser des `list` et des `append`).

Pour chaque expression, nous allons d'abord compiler les valeurs des structures et placer le contenu dans une variable `N` (on s'attend bien évidemment à recevoir une liste de deux éléments vu la manière dont se présente notre fonction), puis on retourne une liste où le premier élément comportera toutes les instructions MIPS dont le premier élément de la variable `N` (avec `first`) et le second élément comportera l'environnement donc le second élément de la variable `N` (avec `second`).

Par exemple pour compiler une opération qui prend en paramètre un symbole et une première valeur `v1` et une deuxième `v2`. On compile déjà `v1` et on le met dans `cv1` puis on compile `v2` et on le met dans `cv2`. Ainsi on se retrouve avec deux listes de deux éléments. Puis on retourne une `list` avec comme première élément l'ensemble des instructions MIPS dont (`first cv1`) et (`first cv2`) mais également d'autres instructions que l'on va implémenté "à la main" et pour pouvoir assimiler les instructions, nous utiliserons `append`.

Remarque : Le nom de chaque variable qui aura comme valeur une expression compilée commencera "c" (pour "compiled"). Je pense que c'est quand même une mauvaise idée d'utiliser `define` pour déclarer des variables, j'aurai pu utiliser `let` comme vous l'avez fait dans *eval.rkt*.

3.3.1 Condition

Concernant les conditions, c'est un peu différent car il faut créer des labels pour faire des branchements afin d'ignorer et/ou de passer à un bloc d'instructions différents. Pour les conditions, il faut commencer par compiler notre test (par convention, nos tests renvoient soit 0 lorsque c'est le test n'est pas rempli soit 1 sinon. Et nous stockerons cette valeur sur `$t9`), puis il faut compiler le `bloc-then` (le bloc d'instructions affecté si la condition est validée) et le `bloc-else` (le bloc d'instructions affecté lorsque la condition n'est pas validée). Cela ressemble plus au moins à ce que l'on a fait précédemment. Cependant, il faudra aussi créer des labels pour chaque condition. Donc j'ai décidé que pour les labels qui nous redirige vers le `bloc-then`, je les nommerai `then_[N]` et les labels qui nous redirigent vers le `bloc-else`, je les nommerai `else_[N]`. De plus, il ne faut pas oublier de créer un label `endif_[N]` où `N` est un nombre que nous incrémenterons à chaque condition (pour différencier chaque label) après chaque `bloc-then` pour lire la suite du code sinon on lira également le `bloc-else` vu qu'il est à la suite du `bloc-then`. Donc si le test est faux, c'est-à-dire `$t9` vaut 0 alors nous irons dans le `bloc-else`, en revanche, si `$t9` vaut 1 alors nous irons dans le `bloc-then` puis `bloc-endif` à la fin. Pour réaliser des branchements, nous utiliserons les instructions `Bnez` (qui prend un registre et un label et qui réalise un branchement si le registre

donnée ne vaut pas 0) et **Beqz** (pareil que **Bnez** sauf qu'il fait un branchement lorsque le registre vaut 0). Ensuite nous avons plus qu'à créer les labels (avec la structure **Label** sur *mips.rkt* et mettre les instructions en question).

3.3.2 Boucle

Les boucles agissent plus ou moins de la même manière. Tout d'abord, je compile le test et le corps de la boucle. Puis, j'écris les instructions du test obtenus lors de la compilation, je crée un label **loop_[N]**, je vérifie si le test est valide (toujours avec l'instruction **Beqz**) et si il n'est pas valide je vais dans le label **endloop_[N]**, j'écris les instructions correspondant au corps de la boucle que nous avons compilées au départ et je fais un branchement inconditionnel (avec l'instruction **B**) vers le même label. Et on oublie pas de retourner également l'environnement du corps de la boucle pour que les valeurs des variables soient modifiées.

3.3.3 Fonctions

Les fonctions possèdent un id (identifiant) et une clôture. On va donc compiler la clôture qui possèdent le corps de la fonction et les arguments, puis écrire le label qui portera le nom de l'id. Attention si l'id se nomme **main**, nous le nommerons **_main** pour ne pas avoir deux labels identiques. J'aurai également pu remplacer le **main** de MIPS par le **main** de mon programme (vu qu'il s'agit d'un programme en C, il faut obligatoirement une fonction **main**) mais j'ai préféré ne pas me casser la tête concernant ce point là car je pense que cela donnerait le même résultat. La clôture correspond à une des structures que je n'ai pas assez développé : pour l'instant il se contente seulement de compiler le corps de la fonction. Le corps de la fonction correspond à la structure **Funblock** qui contient l'ensemble des expressions et la valeur de retour. On a juste à compiler le corps de la fonction puis la valeur de retour et enfin écrire l'instruction **jr \$ra**.

Ce qui m'a le plus posé soucis dans la création d'une fonction, c'est la création d'un bloc d'expressions. En effet, au départ, en utilisant **append** et **map** pour compiler chaque expression de mon bloc, je n'enregistrai pas l'environnement. Et c'est Nabil Boutemour qui m'a fait comprendre que **map** n'était pas forcément une solution optimale pour la "mise à jour" de mon environnement. Du coup, j'ai décidé de prendre plus au moins le code que vous avez réalisé pour **eval-exprs** pour l'interpréteur en utilisant **foldl**.

3.3.4 Appel de fonctions

Cette partie est également à améliorer. Pour l'instant nous pouvons seulement appelé les "fonctions natives" à savoir les opérations arithmétiques et de comparaisons mais également les appels systèmes de MIPS (`print_num` et `print_str`). L'appel système `Systcall` compile la valeur passée en argument et vérifie l'appel système à appliquer selon l'identifiant. Quant aux opérations, on compile la première valeur et on compile la deuxième valeur et on effectue l'instruction MIPS avec la bonne opération selon le symbole donné (le travail réalisé lors des analyses va nous empêcher d'avoir des problèmes de priorités opératoires) . Nous traiterons des améliorations dans la section suivante.

3.3.5 *liec.rkt*

Le fichier *liec.rkt* appelle les fonctions pour les analyses syntaxiques (`define parsed`), sémantique (`define prog`) et la compilation (`comp (Block prog)`). Le programme est un fichier que l'on donne en argument sur la ligne de commande (`((define argv (current-command-line-arguments)))`). On compile le programme avec une table de hachage vide : l'idéal aurait été de créer une librairie standard comme pour l'interpréteur. Je tiens à dire qu'au départ, j'avais eu beaucoup de mal à rajouter mon compilateur à la fin de mes analyses. Le problème venait du fait que j'avais deux AST sur le même dossier (je comptais fusionner l'AST de l'interpréteur avec celui de l'AST du compilateur au fur et à mesure de mon avancement) mais je ne savais pas que les structures déclarées sur une AST sont uniques donc non reconnaissable par une autre structure du même nom (par exemple une structure `Let` dans *ast1.rkt* n'est pas la même que le `Let` dans *ast2.rkt*). J'ai donc bloqué sur ça pendant plusieurs jours.

```
int main(int, str list : argc, argv) {
    int b = 10;
    return 0;
}
```

Exemple de programme

```
move $fp, $sp
_main:
li $v0, 10
addi $sp, $sp, -4
sw $v0, 0($sp)
li $v0, 0
jr $ra
move $t5, $v0
li $v0, 1
move $a0, $t5
syscall
li $v0, 0
jr $ra
```

Code MIPS obtenu

FIGURE 8 – Code MIPS renvoyé pour le programme

4 Améliorations à apporter

Concernant les améliorations à apporter, je pense qu'il y a plusieurs éléments à améliorer. Malheureusement, nous avons eu deux projets en parallèle à gérer donc c'était un peu plus compliqué de gérer les deux. Néanmoins, il y a des éléments plus importants et plus urgents à améliorer. Voici une liste non-exhaustive des améliorations possible (du plus important au moins important) :

- La référence à une variable qui fonctionne à moitié dans le sens où elle récupère seulement la première valeur déclaré. Nous pouvons vérifier cela en décrémentant l'indice de `'fp` pour pouvoir accéder aux autres variables.
- La représentation des chaînes de caractères en MIPS. Pour l'instant, j'ai la section de données `.data`, j'ai décidé de déclarer une table de hachage `data` qui stocker comme clé `str_` (de la même manière que les boucles et les conditions). La fonction `mips-data` place toutes les variables dans la section de données donc tout le contenu de ma table `data` sur la section de données. Cependant, juste après l'exécution de cette instruction, la table de hachage se vide et ne conserve pas l'élément.

- Les appels de fonctions, qui appellent seulement les fonctions natives et pas encore les fonctions déclarées par notre programme. Et nous pouvons mettre dans le même panier la gestion des fonctions avec l'environnement lexical (clôture).
- Créer une librairie standard pour les opérations, et les appels systèmes pour ne pas "envahir" la fonction `comp` et pour que le code soit plus lisible.
- Problème avec le bloc d'expression `Block` et bloc de fonction `Pblock` car le `Block` ne retourne pas de valeur de retour comme je vous l'ai dit précédemment donc les conditions, les boucles ne peuvent pas retourner de valeur, ce qui est problématique (pour les fonctions récursives par exemple). L'idéal aurait été de créer un seul `Block` et d'avoir une option pour la valeur de retour, dans la même idée que l'option `rec` pour les fonctions.
- La structure d'une fonction de notre programme ne ressemble pas forcément à une fonction en C (dû à la mauvaise organisation des arguments) mais je vous ai déjà parlé de cela auparavant
- Eventuellement ajouter plus d'opérations, les nombres flottants, les mots-clés (`const`, `static`...) en C
- Rajout d'une boucle `for`, d'une option pour conditions (pas forcément de `else` et rajout de `else if`)
- L'ajout de `printf`
- Les fonctions de type `void` mais du coup cela revient au problème `Pblock` que j'ai évoqué.
- Phases d'optimisations pour les instructions MIPS dans certaines situations

5 Conclusion

Pour conclure, ce projet m'a permis d'acquérir de nouvelles connaissances et m'a aidé à comprendre la manière dont on doit gérer les analyses et le passage vers un interpréteur ou un compilateur.