Rapport de Compilation

Rapport en français.
Auteurs : Thomas D., Ethan H., Mathieu M. et Thomas B.
GitHub Repo : https://github.com/ThomasByr/SoS2MIPS

Introduction

Le but de ce projet est de réaliser un compilateur pour le langage de programmation SoS vers le langage assembleur MIPS. Le compilateur est écrit en C et utilise les bibliothèques flex et bison.
Avant de commencer, il faut tout d'abord installer les dépendances nécessaires au projet. Pour ce faire, il faut exécuter la commande suivante :

```
sudo apt install flex bison
```

Un paquet Debian pour qtspim (l'émulateur MIPS choisi) est également disponible dans le dossier `vm`. Pour l'installer, il faut exécuter les commandes suivantes :

```
cd vm && sudo dpkg -i qtspim.deb
sudo apt-get install libgl1 libxkbcommon-x11-0
```

Ensuite, il faut compiler une version de production du programme avec la commande suivante :

```
make release
```

Le binaire exécutable produit se trouve dans le dossier `bin`.

Répartition du travail

Le projet a été réalisé en groupe de 4 personnes. La première partie du projet, qui consiste à préparer une application fonctionnelle avec récupération des arguments en ligne de commande n'est pas parallélisable, et a donc été réalisée par une seule personne. Les structures de vecteur `vec_t`, de dictionnaire `dict_t`, de threadpool `threadpool_t`, et de pile d'instructions (quoique cette dernière fut revue en groupe) `astack_t` ont également été réalisées par une seule personne.
Les autres parties du projet ont été réalisées en groupe, notamment la génération de code intermédiaire (en groupe plus restreint) avec les quads, et la génération de code assembleur.

Structure du projet

Le projet est composé de plusieurs dossiers :

- `src` : contient le code source du projet
- `inc` : contient les fichiers d'en-tête du projet
- `gen` : contient les fichiers sources flex et bison
- `tests` : contient les fichiers de tests unitaires
- `vm` : contient les fichiers nécessaires à l'émulation du code assembleur
- `docs` : contient les fichiers de documentation du projet ainsi que ce rapport
- `styles` : contient les fichiers de style pour la génération du rapport

Les fichiers et dossiers suivants servent à la maintenance et à l'intégration continue du projet :

- `makefile` : fichier de configuration de la compilation
- `.gitignore` : fichier de configuration de git
- `setup.bash` : tests et compilation générique lors de *releases* sur github
- `.github/` : contient les fichiers de configuration github
- `.vscode/` : contient les fichiers de configuration de l'éditeur de texte Visual Studio Code
- `scripts/` : contient des scripts pour la génération de documentation

Quelques structures en détail

Pour faciliter la compréhension du code, nous avons défini plusieurs structures de données. Nous allons détailler celles qui ne découlent pas immédiatement de la lecture des documents fournis en cours, en td ou en tp. Nous allons spécifier à chaque fois le header dans lequel la structure est définie, ainsi que la structure elle-même.

Vecteur `vec_t`

Le vecteur est une structure de données dynamique qui permet de stocker des éléments de type `void*`. Il est possible d'ajouter des éléments à la fin du vecteur, de supprimer des éléments à la fin du vecteur, et de récupérer un élément à une position donnée. Son header se trouve dans le fichier `inc/vec.h`.

La structure `vec_s` est intentionnellement opaque à l'utilisateur. Elle est définie dans le fichier `src/vec.c` comme suit :

```c
/// inside vec.h
typedef struct vec_s *vec_t;

/// inside vec.c
struct vec_s {
  void **data;
  size_t size;
  size_t capacity;

  pthread_mutex_t *lock;
};
```

Le vecteur est donc composé d'un tableau de pointeurs `void*` (`data`), d'une taille (`size`), d'une capacité (`capacity`), et d'un mutex (`lock`). Le mutex est utilisé pour rendre le vecteur thread-safe. Le vecteur se redimensionne automatiquement à chaque fois que la capacité est atteinte (à un certain pourcentage) ; et dès que des éléments sont insérés ou supprimés, le vecteur vide l'espace mémoire non utilisé mais le conserve pour les prochaines insertions.

Pour initialiser un vecteur, il faut utiliser la macro `vec_new`. Cette macro permet l'utilisation d'un paramètre optionnel pour définir la capacité initiale du vecteur. Par défaut, la capacité initiale est de 0, et le vecteur est réalloué à chaque fois que la capacité est atteinte. Si la capacité initiale est définie, le vecteur est alloué d'un coup avec la capacité initiale donnée.

```c
vec_t vec = vec_new(); // vec_new(10) for an initial capacity of 10
vec_push(vec, (void*) 421);

printf("%ld\n", (long) vec_get(vec, 0)); // prints 42
vec_free(vec);
```

Dictionnaire `dict_t`

Le dictionnaire est une structure de données dynamique qui permet de stocker des paires clé-valeur de type `void*` et `void*` (par défaut, `void *` et `size_t` ont la même taille en mémoire, nous pouvons donc utiliser les deux de manière transparente). Il est possible d'ajouter des paires clé-valeur, de supprimer des paires clé-valeur, et de récupérer une valeur à partir d'une clé donnée. Son header se trouve dans le fichier `inc/dict.h`.

La structure `dict_s` est intentionnellement opaque à l'utilisateur. Elle est définie dans le fichier `src/dict.c` comme suit :

```c
/// inside dict.h
typedef struct dict_s *dict_t;

/// inside dict.c
struct dict_s {
  int hash_content;
  size_t nbits;
  size_t mask;

  size_t capacity;
  size_t *keys;
  size_t *values;
  size_t nitems;
  size_t n_deleted_items;
};
```

Le dictionnaire est donc composé d'un tableau de clés (`keys`), d'un tableau de valeurs (`values`), d'une capacité (`capacity`), d'un nombre d'éléments (`nitems`), et d'un nombre d'éléments supprimés (`n_deleted_items`). Le dictionnaire est également composé d'un masque (`mask`) et d'un nombre de bits (`nbits`) qui permettent de calculer l'index d'un élément dans le tableau de clés et de valeurs. Le dictionnaire est également composé d'un booléen (`hash_content`) qui permet de déterminer si les clés doivent être interprétées comme des chaines de caractères ou comme des pointeurs.

Pour initialiser un dictionnaire, il faut utiliser la macro `dict_new` . Cette macro permet l'utilisation d'un paramètre optionnel pour définir le mode de hashage des clés. Par défaut, les clés sont interprétées comme des pointeurs. Si le paramètre optionnel est défini à 1, les clés sont interprétées comme des chaines de caractères.

```c
dict_t dict = dict_new(1); // hash content

// following example also works with default mode
dict_push(dict, (void *)"foo1", (void *)"bar");
dict_push(dict, (void *)"foo2", (void *)"baz");

char *value = (char *)dict_get(dict, (void *)"foo1");
printf("%s\n", value); // prints bar
value = (char *)dict_get(dict, (void *)"foo2");
printf("%s\n", value); // prints baz
dict_free(dict);
```

ThreadPool `threadpool_t`

Le threadpool est une structure de données dynamique qui permet de créer un ensemble de threads qui peuvent exécuter des tâches. Il est possible d'ajouter des tâches au threadpool, et de récupérer le résultat d'une tâche. Son header se trouve dans le fichier `inc/threadpool.h`.

La structure `threadpool_s` est intentionnellement opaque à l'utilisateur. Elle est définie dans le fichier `src/threadpool.c` comme suit :

```c
/// inside threadpool.h
typedef struct threadpool_s *threadpool_t;

/// inside threadpool.c
struct threadpool_s {
  pthread_mutex_t lock;
  pthread_cond_t notify;
  pthread_t *threads;
  threadpool_task_t *queue;
  int thread_count;
  int queue_size;
  int head;
  int tail;
  int count;
  int shutdown;
  int started;
};

typedef struct {
  void (*function)(void *);
  void *argument;
} threadpool_task_t;
```

Le threadpool est donc composé d'un tableau de threads (`threads`), d'un tableau de tâches (`queue`), d'un nombre de threads (`thread_count`), d'une taille de queue (`queue_size`), d'un index de tête de queue (`head`), d'un index de queue (`tail`), d'un nombre de tâches dans la queue (`count`), d'un booléen (`shutdown`) qui indique si le threadpool est en train de se terminer, et d'un booléen (`started`) qui indique si le threadpool a été démarré. Le threadpool est également composé d'un mutex (`lock`) et d'une condition (`notify`) qui permettent de synchroniser les threads.

Pour initialiser un threadpool, il faut utiliser la fonction `threadpool_create`, qui prend en paramètre le nombre de threads à créer, la taille de la queue, et des flags (qui ne sont pas utilisés pour le moment).

```c
threadpool_t pool = threadpool_create(4, 100, 0);

// add tasks to threadpool
for (int i = 0; i < 100; i++) {
  threadpool_add(pool, &some_function, NULL, 0);
}

// wait for all tasks to finish and destroy threadpool
threadpool_destroy(pool, 1)
```

Pile d'instructions assembleur `astack_t`

La pile d'instructions assembleur est une structure de données dynamique qui permet de stocker des instructions assembleur. Elle est utilisée par le compilateur pour stocker les instructions assembleur block par block générées par le compilateur. Il s'agit de la seule structure composite, dont son fonctionnement dépend entièrement du bon fonctionnement des structures précédentes. Son header se trouve dans le fichier `inc/assstack.h`.

La structure `astack_s` est intentionnellement opaque à l'utilisateur. Elle est définie dans le fichier `src/assstack.c` comme suit :

```
/// inside assstack.h
typedef struct astack_s *astack_t;

/// inside assstack.c
struct astack_s {
  vec_t data;
  vec_t text;
  vec_t name;
  dict_t text_blocks;
};
```

La pile d'instructions assembleur est composée d'un vecteur de données (`data`), d'un vecteur de vecteurs de textes (`text`), d'un vecteur de noms (`name`), et d'un dictionnaire de blocs de texte (`text_blocks`). Le vecteur de données (`data`) est utilisé pour stocker les directives assembleur de la section `.data`. Le vecteur de vecteurs de textes (`text`) est utilisé pour stocker les directives assembleur de la section `.text` ; chaque vecteur de textes correspond à un bloc de texte. Le vecteur de noms (`name`) est utilisé pour stocker les noms des blocs de texte, dans le même ordre que le vecteur de vecteurs de textes. Le dictionnaire de blocs de texte (`text_blocks`) est utilisé pour stocker les correspondances entre les noms des blocs de texte et leurs index dans le vecteur de vecteurs de textes.

Pour initialiser une pile d'instructions assembleur, il faut utiliser la macro `astack_new`, qui ne prend deux paramètres, la taille initiale du vecteur de données et la taille initiale du vecteur de vecteurs de textes.

```
// won't work when freeing astack but it's ok for this example
// normally, we would malloc bloc names
// directives are copied, so we can free them after pushing them
// which allows us to also use string literals
astack_t stack = astack_new(2, 10);
astack_push_data(stack, "msg: .asciiz \"Hello World!\"");
astack_push_text(stack, "main", "li $v0, 4");
astack_push_text(stack, "main", "la $a0, msg");
astack_push_text(stack, "main", "syscall");
```

Par défaut, le block `main` de la section `.text` est créé. Il est possible d'ajouter des directives assembleur à la pile d'instructions assembleur avec les fonctions `astack_push_data` et `astack_push_text`. La fonction `astack_push_data` prend en paramètre une directive assembleur de la section `.data` et l'ajoute au vecteur de données. La fonction `astack_push_text` prend en paramètre une directive assembleur de la section `.text` et le nom du bloc de texte auquel elle appartient. Si le bloc de texte n'existe pas, il est créé. La directive assembleur est ensuite ajoutée au vecteur de vecteurs de textes correspondant au bloc de texte.

Une représentation de la pile d'instruction assembleur est disponible ci-dessous.

```
// .data segment representation
vec_t data : { "msg: .asciiz \"Hello World!\"" }

// .text segment representation
vec_t text : {
  vec_t text_block_1 : { "li $v0, 4", "la $a0, msg", "syscall" },
}
vec_t name : { "main" }
dict_t text_blocks : { "main" => 1 /* index cannot be 0 (will cast to NULL) */ }
```

Le dictionnaire est utile et très efficace lorsqu'il s'avère nécessaire de chercher un bloc de texte dans le vecteur de vecteurs de textes. En effet, la recherche dans un dictionnaire est en moyenne plus rapide que la recherche dans un vecteur. A la fin de la compilation, la fonction `astack_fprintf` permet d'écrire la pile d'instructions assembleur dans un fichier. Le vecteur de données est écrit dans la section `.data` et le vecteur de vecteurs de textes est écrit dans la section `.text`, dans l'ordre des noms des blocs de texte.

Tests

Les tests unitaires sont disponibles dans le dossier `tests/`. Ils sont écrits en C et supposent que Valgrind est installé sur la machine. Pour compiler et exécuter les tests, il faut utiliser la commande `make check` ou `make check_quiet`. La seconde commande redirige toutes les sorties du programme vers `/dev/null` et ne garde que l'affichage de la progression des tests.

```
cd tests && make check
```

De manière alternative, `make check_quiet` permet d'exécuter les tests en redirigeant toutes les sorties du programme vers `/dev/null`. `make` n'utilise pas cette commande par défaut.

Documentation et Utilisation

La documentation intégrée est disponible avec la commande `--help` ou `-h`. Voici néanmoins un résumé des options disponibles. Nous indiquons si une option requiert un argument avec `<>`, et si une option courte n'est pas disponible, nous indiquons `..` à la place.

commande	aide	requis ?	défaut
`-h, --help`	affiche l'aide et **exit**	?	
`-v, --version`	affiche la version et **exit**	?	
`-l, --license`	affiche la licence **exit**	?	
`-i, --in <>`	chemin vers le fichier à compiler	✓	
`-o, --out <>`	fichier vers lequel sauver le code généré	✗	`a.s`
`.., --tos`	affiche la table des symboles	✗	
`.., --verbose`	quelques infos supplémentaires	✗	
`.., --no-exe`	n'exécute pas le fichier produit	✗	
`-O, --opt_lvl <>`	niveau d'optimisation (parmis `0` ou `1`)	✗	`0`

Notez que l'option `-O` n'est à ce jour pas implémentée. Par ailleurs, les fichiers d'entrée et de sortie peuvent être spécifiés sans utiliser les options `-i` et `-o`. Il suffit de les passer en argument de la commande, dans l'ordre si les deux venaient à être spécifiés de cette manière. Si un seul fichier est spécifié sans option, il est considéré comme le fichier manquant parmi les deux.
Un exemple d'utilisation est disponible ci-dessous.

```
./bin/sos -o out.s examples/hello_world.sos --tos --verbose
```

Remarques générales sur la grammaire

La remarque la plus générale sur ce projet porte sur les types de données qu'il est possible de traiter avec notre implémentation du compilateur SOS. Nous n'avons traité que le cas de variables stockant des entiers ou des tableaux d'entiers. Les chaînes de caractères peuvent être affichées mais doivent être statiques (entièrement définies au moment de la compilation). Les variables à virgule flottante à simple et double précision ne sont pas traitées.

Les opérateurs logiques ont aussi été revus. Étant donné qu'il est possible d'avoir des opérations arithmétiques à l'intérieur d'un bloc de test, les opérateurs commençant par `-` provoquent des erreurs dans notre implémentation. Nous avons donc enlevé ce préfixe superflu.

Par ailleurs, il est actuellement impossible d'utiliser `$*` pour une boucle `for` à l'intérieur d'une fonction.

Milestones

Nous allons maintenant détailler les différentes étapes de développement du projet. Les exemples de code ci-dessous sont pour la plupart issus de tests, et certains peuvent même être retrouvés dans le dossier examples/. Notez également que ces étapes reflètent l'avancement de notre projet uniquement à l'instant où nous avons écrit ce document. Vous pourrez remarquer que quelques tests assembleurs ont été implémentés, notamment lors d'accès mémoire ("Index out of array bounds" par exemple), pour entrer dans l'état erreur au moment de l'exécution du programme et pas avant.

opérations arithmétiques

La toute première étape du projet a été de faire fonctionner les opérations arithmétiques basiques (`+`, `-`, `*`, `/`, `%`) qui ne fonctionnent que sur les entiers (le code suivant n'est syntaxiquement pas valide, il faut stocker le résultat de l'opération, le fichier assembleur a donc été tronqué).

```
$( expr (4-1)+2);
exit
```

```
li $t0, 4
li $t1, 1
sub $t2, $t0, $t1
li $t0, 2
add $t1, $t2, $t0
sw $t1, i
j _exit

_exit:
li $v0, 10
syscall

_exit2:
li $v0, 17
syscall
```

stockage de variables entières

La seconde étape a été de faire fonctionner le stockage de variables. Nous avons choisi d'écrire les variables entières dans le segment `.data`. Il est possible de stocker des variables dans le segment `.text` mais cela nécessite de faire des manipulations supplémentaires pour accéder à la mémoire. Nous avons donc choisi de stocker les variables dans le segment `.data` pour simplifier le code généré.

```
i = 1;
i = $( expr ${i}+1 );
exit
```

```
.data
error_msg0: .asciiz ""
msg_space: .asciiz " "
i: .word 0

.text
.globl main

main:
li $t0, 1
sw $t0, i
lw $t0, i
li $t1, 1
add $t2, $t0, $t1
sw $t2, i
j _exit

_exit:
li $v0, 10
syscall

_exit2:
li $v0, 17
syscall

_error:
li $v0, 4
la $a0, error_msg0
syscall
li $v0, 10
syscall
```

stockage de variables de type tableau d'entiers

L'étape naturelle suivante a été de faire fonctionner le stockage de variables de type tableau d'entiers. Nous avons choisi de stocker les tableaux dans le segment `.data` à l'aide de la directive `.word`.

```
declare tab[10];
tab[0] = 1;
exit
```

```
.data
error_msg0: .asciiz ""
msg_space: .asciiz " "
tab: .word 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
tab_size: .word 10
error_msg1: .asciiz "Index out of array bounds (sos:2)\n"

.text
.globl main

main:
li $t0, 0
li $t1, 1
lw $t2, tab_size
bge $t0, $t2, _error
mul $t0, $t0, 4
sw $t1, tab($t0)
j _exit

_exit:
li $v0, 10
syscall

_exit2:
li $v0, 17
syscall

_error:
li $v0, 4
la $a0, error_msg1
syscall
li $v0, 10
syscall
```

affichage en console avec `echo`

Les primitives système MIPS `1` et `4` (car nous ne nous occupons pas des chaînes de caractère et des nombres à virgule flottante à simple ou double précision) nous servent à l'affichage. La principale difficulté de l'utilisation du `echo` en SOS est le nombre d'argument très variable de cette directive. Il a donc fallu (comme pour toutes les règles dérivants de `ops`) allouer de la mémoire dynamiquement avec `sbrk`. Nous avons choisi de stocker les chaînes de caractères statiques dans le segment `.data` à l'aide de la directive `.asciiz`. Il est possible également d'afficher un tableau d'entier `tab` avec `echo ${tab[*]}`.

```
echo "Hello, world!\n";
exit
```

```
.data
error_msg0: .asciiz ""
msg_space: .asciiz " "
msg0: .asciiz "Hello, world!\n"

.text
.globl main

main:
li $v0, 9
li $a0, 256
syscall
move $t0, $v0
la $a0, msg0
li $v0, 4
syscall
j _exit

_exit:
li $v0, 10
syscall

_exit2:
li $v0, 17
syscall

_error:
li $v0, 4
la $a0, error_msg0
syscall
li $v0, 10
syscall
```

lecture en console avec `read`

La primitive système MIPS 5 nous sert à la lecture. Nous ne lisons que des entiers en console (que ça soit des variables entière, ou des accès lecture à des tableaux d'entiers). (Nous ne mettons pas le code généré en assembleur ici car il n'aide pas vraiment à la compréhension et est très volumineux ; vous pouvez néanmoins le retrouver avec la commande `./bin/sos examples/read.sos -o test.s`.)

```
echo "Entrez un entier : ";
read entier;
echo "Vous avez rentré : ";
echo ${entier};
declare tab[10] ;
i = 1 ;
echo "\nModifiez la valeur du tableau à l'indice 1 : " ;
read tab[${i}] ;
echo "La valeur du tableau est : " ;
echo ${tab[*]};
exit
```

opérations logiques

Avant de pouvoir implémenter les structures de contrôle, il faut implémenter les opérations (portes) logiques. Les opérations implémentées sont les suivantes : `eq` (pour `==`), `ne` (pour `!=`), `lt` (pour `<`), `le` (pour `<=`), `gt` (pour `>`), `ge` (pour `>=`), `a` (pour `&&`), `o` (pour `||`), et `!`.

Attention, le code SOS suivant n'est pas syntaxiquement correcte, il faut utiliser ces opérateurs dans des structures de contrôle et non seules.

```
test ( 1 eq 1 ) ; /* true  */
test ( 1 eq 2 ) ; /* false */
test ( 1 ne 1 ) ; /* false */
test ( 1 ne 2 ) ; /* true  */
test ( 1 lt 2 ) ; /* true  */
test ( 1 lt 1 ) ; /* false */
test ( 1 le 1 ) ; /* true  */
test ( 1 le 2 ) ; /* true  */
test ( 1 gt 2 ) ; /* false */
test ( 1 gt 1 ) ; /* false */
test ( 1 ge 1 ) ; /* true  */
test ( 1 ge 2 ) ; /* false */
```

structures de contrôle

Nous avons implémenté les structures de contrôle logiques suivantes : `if` et `if-else`. La structure de contrôle de flot `for` est aussi implémentée mais présente quelques imperfections. Grâce à la pile d'instructions, les structures de contrôles logiques reviennent à empiler successivement et à dépiler progressivement les noms des blocs sur la pile des blocs. La boucle `while` se programme facilement en combinant les structures de contrôle déjà présentes. La boucle `until` n'a pas été implémentée.

```
echo "Enter a number : ";
read i;
echo "\n";
if test ${i} ge 12 then
echo "true\n"
else
echo "false\n"
fi;
exit 3
```

```
.data                                   | _instr0:
error_msg0: .asciiz ""                  | li $v0, 9
msg_space: .asciiz " "                  | li $a0, 256
msg0: .asciiz "Enter a number : "       | syscall
i: .word 0                              | move $t0, $v0
msg1: .asciiz "\n"                      | la $a0, msg2
msg2: .asciiz "true\n"                  | li $v0, 4
msg3: .asciiz "false\n"                 | syscall
                                        | j _instr2
.text                                   |
.globl main                             | _instr1:
                                        | li $v0, 9
main:                                   | li $a0, 256
li $v0, 9                               | syscall
li $a0, 256                             | move $t0, $v0
syscall                                 | la $a0, msg3
move $t0, $v0                           | li $v0, 4
la $a0, msg0                            | syscall
li $v0, 4                               |
syscall                                 | _instr2:
li $v0, 5                               | li $t0, 3
syscall                                 | move $a0, $t0
sw $v0, i                               | j _exit2
li $v0, 9                               |
li $a0, 256                             | _exit:
syscall                                 | li $v0, 10
move $t0, $v0                           | syscall
la $a0, msg1                            |
li $v0, 4                               | _exit2:
syscall                                 | li $v0, 17
lw $t0, i                               | syscall
li $t1, 12                              |
sge $t2, $t0, $t1                       | _error:
beq $t2, 1, _instr0                     | li $v0, 4
j _instr1                               | la $a0, error_msg0
                                        | syscall
                                        | li $v0, 10
                                        | syscall
```

La syntaxe d'une boucle `for` est la suivante :

```
for i in 1 2 3 4
do
  echo "i = " ${i} "\n"
done;
exit
```

```
declare tab[10];
for i in ${tab[*]}
do
  echo "top\n"
done;
exit
```

Code assembleur de la deuxième boucle `for` :

```
.data
error_msg0: .asciiz ""
msg_space: .asciiz " "
i: .word 0
msg0: .asciiz "i = "
msg1: .asciiz "\n"

.text
.globl main

main:
li $s0, 0
li $v0, 9
li $a0, 256
syscall
move $t0, $v0
li $t1, 1
sw $t1, 0($t0)
li $t1, 2
sw $t1, 4($t0)
li $t1, 3
sw $t1, 8($t0)
li $t1, 4
sw $t1, 12($t0)

_for0:
lw $t1, 0($t0)
addi $t0, $t0, 4
sw $t1, i
addi $s0, $s0, 1
li $v0, 9
li $a0, 256
syscall
move $t1, $v0
lw $t2, i
sw $t2, 0($t1)
la $a0, msg0
li $v0, 4
syscall
la $a0, msg_space
li $v0, 4
syscall
lw $a0, 0($t1)
li $v0, 1
syscall
la $a0, msg_space
li $v0, 4
syscall
la $a0, msg1
li $v0, 4
syscall
ble $s0, 3, _for0

_instr1:
j _exit

_exit:
li $v0, 10
syscall

_exit2:
li $v0, 17
syscall

_error:
li $v0, 4
la $a0, error_msg0
syscall
li $v0, 10
syscall
```

La syntaxe d'une boucle `while` est la suivante :

```
i = 0;
while test ${i} le 12 do
  echo ${i} "\n";
  i= $(expr ${i} + 1)
done;
exit
```

```
.data
error_msg0: .asciiz ""
msg_space: .asciiz " "
i: .word 0
msg0: .asciiz "\n"

.text
.globl main

main:
li $t0, 0
sw $t0, i

_instr0:
lw $t0, i
li $t1, 12
sle $t2, $t0, $t1
beq $t2, 1, _instr1
j _instr2

_instr1:
li $v0, 9
li $a0, 256
syscall
move $t0, $v0
lw $t1, i
sw $t1, 0($t0)
lw $a0, 0($t0)
li $v0, 1
syscall
la $a0, msg_space
li $v0, 4
syscall
la $a0, msg0
li $v0, 4
syscall
lw $t0, i
li $t1, 1
add $t3, $t0, $t1
sw $t3, i
j _instr0

_instr2:
j _exit

_exit:
li $v0, 10
syscall

_exit2:
li $v0, 17
syscall

_error:
li $v0, 4
la $a0, error_msg0
syscall
li $v0, 10
syscall
```

ce qu'il reste à faire (et qu'on sait comment faire)

- nombres négatifs
- nombres à double précision
- `concat` (les id sont que des entiers, on n'a pas encore fait l'assignation de tableaux)
- until ... do
- elif

ce qu'il reste à faire (et qui est pour l'instant obscure)

- typage (s'occuper des chaînes de caractères, stocker l'adresse de retour de sbrk dans la table des symboles)
- case esac

Licence