Projet MicroGo

Rapport des parties 1 et 2

Reina Al Masri Sophie Sarr

8 décembre 2025

Table des matières

1 Introduction **1**

2 Architecture générale **2**

3 Partie 1 – Analyse statique de MicroGo **2**
3.1 Analyse lexicale : mgolexer.mll . . . 2
3.2 Bonus : insertion automatique des points-virgules . . . 2
3.3 Analyse syntaxique : `mgoparser.mly` . . . 3
3.4 Vérification de types : `typechecker.ml` . . . 6
3.5 Bonus : analyse statique avancée dans `typechecker.ml` . . . 8

4 Partie 2 – Génération de code MIPS **9**
4.1 Schéma de compilation retenu . . . 9
4.2 Modèle de pile et tableaux d'activation . . . 10
4.3 Structures et allocation sur le tas . . . 11
4.4 Retours multiples et affectations multiples . . . 11
4.5 Runtime et impression . . . 13
4.6 Compilation d'un programme complet . . . 13

5 Adaptation de `mips.ml` **14**

6 Tests, résultats et difficultés rencontrées **14**
6.1 Campagne de tests . . . 14
6.2 Exemple de génération de code . . . 15
6.3 Difficultés rencontrées . . . 16

7 Conclusion **16**

1 Introduction

L'objectif du projet MicroGo est de construire, à partir d'un squelette fourni, un analyseur et un compilateur pour un sous-ensemble du langage Go (MicroGo) vers MIPS 32 bits. La partie 1 du sujet demande de compléter le lexeur, le parseur et le vérificateur de types. La partie 2 demande de générer du code MIPS à partir de l'AST et du typage, en suivant le modèle de pile proposé dans le cours et dans l'énoncé.

Ce rapport se limite aux éléments que nous avons complétés ou étendus par rapport au squelette :

— **Partie 1** : complétion de `mgolexer.mll`, `mgoparser.mly` et `typechecker.ml`.

— **Partie 2** : implémentation de `compile.ml` et petites extensions de `mips.ml` nécessaires à la génération de code.

Les autres fichiers (`mgoc.ml`, configuration Dune, etc.) sont utilisés tels quels ou presque tels quels et ne sont pas détaillés ici.

2 Architecture générale

L'architecture proposée dans le sujet est conservée. Nous résumons ici uniquement les fichiers dans lesquels nous avons apporté des modifications substantielles ou nécessaires pour les parties 1 et 2 :

— `mgolexer.mll` : lexeur complet pour MicroGo, avec gestion des entiers, des chaînes et des mots-clés, ainsi que le bonus d'insertion automatique des points-virgules.
— `mgoparser.mly` : parseur Menhir couvrant la grammaire de MicroGo de la figure 1 de l'énoncé.
— `typechecker.ml` : vérification statique complète selon les règles de typage du sujet, incluant les fonctions à retours multiples, les structures et le bonus sur l'utilisation de `fmt.Print`.
— `compile.ml` : génération de code MIPS pour toutes les constructions de l'AST, gestion de la pile, des variables locales, des structures allouées sur le tas et des retours multiples.
— `mips.ml` : nous réutilisons l'infrastructure du squelette (type `asm`, impression de programme) et du cours, et nous l'augmentons légèrement (quelques instructions supplémentaires, un petit *builder* et la gestion des constantes chaînes).

3 Partie 1 – Analyse statique de MicroGo

3.1 Analyse lexicale : mgolexer.mll

Le lexeur construit dans `mgolexer.mll` reconnaît :

— les mots-clés de MicroGo (`package`, `import`, `type`, `struct`, `func`, `var`, `new`, `return`, `for`, `if`, `else`, `int`, `bool`, `string`, `true`, `false`, `nil`) via une table `keyword_or_ident` ;
— les identificateurs, entiers (avec contrôle de dépassement `int64`) et chaînes de caractères ;
— les opérateurs et délimiteurs (arithmétiques, logiques, `:=`, `++`, `{}`, parenthèses, etc.) ;
— les commentaires multi-lignes `/* ... */` et sur une ligne `// ...`.

Erreurs lexicales. Nous produisons des erreurs explicites pour les cas suivants :

— chaîne non terminée : `"...` suivi d'une fin de ligne ou de fichier ⇒ `"unterminated string literal"` ;
— entier trop grand : exception lors de la conversion en `int64` ⇒ `"literal constant too large"` ;
— caractère inconnu : symbole ne correspondant à aucun token ⇒ `"unknown character : X"` ;
— commentaire `/* ...` non refermé avant `EOF` ⇒ `"unterminated comment"`.

Ces erreurs sont déclenchées au moment où le lexeur lit le lexème fautif, ce qui permet à `mgoc` d'afficher une localisation précise (ligne, colonne).

3.2 Bonus : insertion automatique des points-virgules

Le sujet demande l'insertion automatique d'un `;` en fin de ligne dans certains cas (identifiant, entier, chaîne, `true`, `false`, `nil`, `return`, `++`, `--`, `)` ou `}`).

Pour cela, les éléments suivants sont ajoutés :

— une référence globale `insert_semi : bool ref` indiquant si le prochain saut de ligne doit produire un `SEMI` synthétique ;

— deux fonctions utilitaires `mark` et `unmark` appliquées à chaque token émis.

Idée générale :

— `mark token` met `insert_semi := true` si `token` appartient à l'ensemble du sujet (identifiant, littéral, `RETURN`, `INCR`, `DECR`, `RPAR`, `END`) ;

— `unmark token` remet le drapeau à `false` pour les autres tokens (dont `SEMI`, `BEGIN`, `LPAR`, etc.) ;

— lors de la lecture d'un `\n`, si `insert_semi` vaut `true`, le lexeur renvoie un `SEMI` (et remet le drapeau à `false`), sinon il ignore simplement le saut de ligne.

Listing 1 – Insertion automatique de `;` dans le lexeur

```
let insert_semi = ref false

let mark token =
  insert_semi :=
    match token with
    | IDENT _ | INT _ | STRING _
    | TRUE | FALSE | NIL
    | RETURN | INCR | DECR
    | RPAR | END -> true
    | _ -> false ;
  token

let unmark token =
  insert_semi := false ;
  token

| ['\n'] {
    new_line lexbuf ;
    if !insert_semi then unmark SEMI else token lexbuf }
```

Ainsi, un programme comme :

```
x := 3
y := 4
```

est vu par le parseur comme `x := 3;` puis `y := 4;`.

3.3 Analyse syntaxique : `mgoparser.mly`

La grammaire de MicroGo est donnée dans la figure 1 de l'énoncé (figure **??**). Nous avons suivi cette grammaire de près pour écrire le fichier `mgoparser.mly`.

Construction systématique des nœuds annotés. Tous les nœuds d'instructions et d'expressions sont annotés par une localisation `loc`, en utilisant des constructeurs auxiliaires :

Listing 2 – Constructeurs d'instructions et d'expressions

```
let mk_i loc idesc = { idesc; iloc = loc }
let mk_var id      = { edesc = Var id;  eloc = id.loc }
let mk_bool loc b  = { edesc = Bool b; eloc = loc }
```

Ces fonctions sont utilisées dans les règles Menhir importantes (`instr`, `instr_if`, `expr`, ...) pour construire un AST bien localisé à partir de `$startpos` et `$endpos`.

Programme et import optionnel de `fmt`. Le non-terminal `prog` renvoie un couple `(fmt_imported, decls)`. Nous distinguons explicitement :

— `PACKAGE main; decls EOF` ⇒ `(false, decls)` ;

— `PACKAGE main; IMPORT "fmt"; decls EOF` ⇒ `(true, decls)` (et seulement si la chaîne exacte est `"fmt"`).

En cas de nom de package différent de `"main"` ou d'import incorrect, le parseur lève une exception `Error`.

Déclarations de structures et de fonctions. Pour les structures, nous gérons les groupes de champs de même type (`quo, rem int`) via un non-terminal `varstyp` qui produit une liste `(ident * mgotype) list`, puis nous aplatissons ces listes.

Pour les fonctions, nous stockons :

— la liste des paramètres `params : (ident * mgotype) list` ;

— la liste des types de retour `return : mgotype list`, même s'il n'y a qu'un seul type.

Ces deux formes de déclarations sont capturées par la règle Menhir `decl` :

Listing 3 – Règles Menhir pour les déclarations

```
decl:
  (* Dclaration de structure *)
  TYPE id=ident STRUCT BEGIN groups=field_groups END SEMI
    { Struct { sname = id; fields = List.flatten groups } }

  (* Dclaration de fonction *)
| FUNC fname=ident LPAR pl=params_opt RPAR ret=return_opt b=bloc SEMI
    { Fun { fname = fname; params = pl; return = ret; body = b } }
;

(* Groupes d'identifiants partageant le mme type, ex. "quo, rem int" *)
varstyp:
  | ids=idents1 t=mgotype { List.map (fun x -> (x, t)) ids }
```

Ainsi, le couple `(⟨vars⟩, type)` du sujet est traduit de manière systématique en listes `(ident * mgotype) list` dans l'AST.

Traitement de `if / else if / else`. Nous encodons les chaînes `if ... else if ... else ...` en réutilisant le constructeur `If(cond, th, alt)` dans la branche `else`.
Par exemple, `if cond {th} else if cond2 {th2} else {el}` se représente par un `If` dont la troisième composante contient un autre `If` .

Boucles `for`. Nous traitons les trois formes de `for` imposées par le sujet :

— `for bloc` ⇒ `For(true, bloc)` ;

— `for cond bloc` ⇒ `For(cond, bloc)` ;

— `for init; cond; post bloc` est réécrit en un bloc contenant `init` suivi d'un `For(cond, body')` où `body'` est le bloc original prolongé par `post`.

La règle Menhir correspondante construit directement cette structure :

Listing 4 – Réécriture de `for init; cond; post bloc`

```
| FOR init=instr_simple_opt SEMI cond=expr SEMI post=instr_simple_opt body=bloc
    {
      let loc = ($startpos, $endpos) in
      let loop_body =
        match post with
        | None -> body
        | Some p -> body @ [p]
```

```
      in
      let loop_instr = mk_i loc (For (cond, loop_body)) in
      let prefix =
        match init with
        | None -> []
        | Some i -> [i]
      in
      mk_i loc (Block (prefix @ [loop_instr]))
    }
```

Affectations simples et déclarations `:=`. La forme `x, y := ...` est traitée séparément de `x, y = ...`. Nous imposons, comme le sujet, que le membre gauche de `:=` ne contienne que des identifiants. La fonction `only_vars` vérifie ce point et lève une exception syntaxique en cas d'expression plus complexe :

Listing 5 – Restriction sur le membre gauche de `:=`

```
let only_vars el =
  List.map (fun e ->
      match e.edesc with
      | Var id -> id
      | _ -> raise Error) el

| lhs=expr_list1 DEFINE rhs=expr_list1
    {
      let ids = only_vars lhs in
      let lhs_exprs = List.map mk_var ids in
      let loc = ($startpos, $endpos) in
      let assign = mk_i loc (Set(lhs_exprs, rhs)) in
      mk_i loc (Vars(ids, None, [assign]))
    }
```

`fmt.Print` et expressions. Les expressions `fmt.Print(...)` sont reconnues par une règle spécifique qui transforme l'appel en constructeur `Print` de l'AST. Les autres appels utilisent le constructeur `Call`.

Précédence et associativité des opérateurs. Pour respecter la table des opérateurs de la figure **??** sans alourdir la grammaire, nous utilisons les déclarations de précédence Menhir suivantes :

Listing 6 – Précédence des opérateurs dans `mgoparser.mly`

```
%left OR
%left AND
%nonassoc EQ NEQ LT LE GT GE
%left PLUS MINUS
%left STAR DIV MOD
%right UMINUS UNOT
%left DOT
```

Dans Menhir, les lignes les plus bas ont la *précédence la plus forte*. L'ordre choisi réalise donc l'intuition suivante :

- `.` (accès champ) a la plus forte précédence ;
- les opérateurs unaires `-` et `!` (via `%prec UMINUS/UNOT`) viennent juste après ;
- puis les opérateurs multiplicatifs `* / %`, puis additifs `+ -` ;
- ensuite les comparaisons (`== != < <= > >=`), puis `&&`, puis `||`.

Par exemple, l'expression `a + b * c == d && e` est automatiquement interprétée comme

$$((a + (b * c)) == d)e$$

sans parenthèses supplémentaires dans la grammaire.

3.4 Vérification de types : `typechecker.ml`

Le fichier `typechecker.ml` implémente les règles de typage décrites dans la section 3 de l'énoncé. Nous séparons trois environnements :

- `senv` : types de structures (`S` $\mapsto$ liste de champs typés `(ident * typ) list`) ;
- `fenv` : fonctions globales (`f` $\mapsto$ déclaration complète, incluant paramètres et types de retour) ;
- `tenv` : environnement local de typage (`x` $\mapsto$ `typ`) pour les variables en portée.

L'identifiant spécial `_` n'est jamais inséré dans `tenv` (ni dans les ensembles de variables utilisés pour le bonus). Il ne peut donc pas apparaître dans une expression bien typée.

Construction des environnements globaux. La fonction `prog` commence par parcourir la liste des déclarations pour remplir `senv` et `fenv`, conformément au « trois temps » proposés dans le sujet :

1. collecte de toutes les structures dans `senv` (sans vérifier encore leurs champs) ;
2. collecte de toutes les fonctions dans `fenv`, en vérifiant l'unicité des noms et la bonne formation des types utilisés (y compris les `*S`) ;
3. vérification des champs de chaque structure (`check_fields`) : pas de doublons dans les noms de champs et types bien formés.

La fonction auxiliaire `check_typ` implémente le jugement

$$\Gamma \vdash \tau \text{ bien formé}$$

en autorisant uniquement `int`, `bool`, `string` et `*S` avec `S` déclarée dans `senv`.

Typage des expressions. La fonction

$$\texttt{type\_expr} : \mathit{expr} \to \mathit{tenv} \to \mathit{typ}$$

réalise le jugement $\Gamma \vdash e : \tau$. Elle est utilisée soit directement, soit via `check_expr` qui compare le type obtenu avec un type attendu.

- **Constantes et `nil`.** Les constantes entières, booléennes et chaînes sont typées en `TInt`, `TBool`, `TString`. La constante `Nil` est représentée comme un pointeur vers une structure abstraite (`TStruct "_"`). Ce type spécial ne peut pas être utilisé comme type d'une variable sans annotation explicite (règle de `var`, voir plus bas).
- **Variables.** Dans le cas `Var x`, nous recherchons `x.id` dans `tenv`. En cas d'absence, une erreur `"unknown variable"` est levée. Le bonus « variables non utilisées » marque en même temps la variable comme lue (voir section bonus).
- **Structures : `new(S)` et accès champ `e.x`.** L'expression `New s` vérifie que `s` est bien une structure déclarée dans `senv` et retourne `TStruct s`, ce qui correspond à la règle de formation

$$\frac{S \in \Gamma}{\Gamma \vdash \texttt{new(S)} : *S}.$$

Pour `Dot(e1,f)`, nous appliquons la règle suivante :

```
| Dot (e1, f) ->
    let t1 = type_expr e1 tenv in
    let struct_name =
      match t1 with
      | TStruct s -> s
      | _ -> error e.eloc "dot applied to non
    -struct"
    in
    let fields =
      try Env.find struct_name senv
      with Not_found ->
        error e.eloc ("unknown struct type "
    ^ struct_name)
    in
    try
      let (_, t) =
        List.find (fun (id,_) -> id.id = f.id
    ) fields
      in t
    with Not_Found ->
      error e.eloc ("unknown field " ^ f.id)
```

$$\frac{\Gamma \vdash e : *S \qquad S\{x : \tau\} \qquad e \neq \texttt{nil}}{\Gamma \vdash e.x : \tau}$$

FIGURE 1 – Accès aux champs d'une structure.

Dans notre représentation interne, le type `TStruct s` correspond au type pointeur `*S` de l'énoncé. La règle d'inférence ci-dessus est donc appliquée en considérant `TStruct s` comme l'analogue de `*S`.

Si `e1` n'est pas de type pointeur vers une structure, ou si le champ `f` n'existe pas dans la définition de `S`, une erreur de typage est levée.

Opérateurs unaires et binaires. Les cas `Unop` et `Binop` suivent directement la table des opérateurs du sujet : `-e` n'est autorisé que sur `int`, `!e` que sur `bool`, les opérateurs arithmétiques exigent deux `int` et produisent un `int`, les comparaisons produisent un `bool` lorsque les deux opérandes ont le même type (dans notre implémentation nous nous limitons aux entiers, ce qui suffit pour les tests fournis), et `&&`/`||` exigent deux `bool`.

Appels de fonction. Pour `Call(f,args)`, nous récupérons la déclaration `fd` dans `fenv`, extrayons la liste des types paramètres `param_types` et vérifions chaque argument avec `check_expr`. Si `fd.return = [t]`, l'appel est lui-même typé `t`. Si la fonction a plusieurs types de retour, l'appel n'est jamais utilisé comme expression isolée : il n'apparaît que dans les contextes qui attendent un n-uplet de résultats (affectation multiple ou `return`, cf. ci-dessous).

`fmt.Print`. Le constructeur d'AST `Print args` est typé en vérifiant simplement chaque argument (cas général) ou, lorsque l'argument unique est un appel de fonction, en réutilisant la vérification de `Call`. Le type retourné est un `int` « factice », qui n'est jamais exploité ; `fmt.Print` ne sert qu'à déclencher des effets de sortie. Les contraintes globales sur l'import de `"fmt"` sont traitées dans le bonus (voir sous-section suivante).

Typage des instructions. Le jugement $\Gamma \vdash s$ est implémenté par `check_instr`, qui renvoie l'environnement mis à jour, et `check_seq` qui le plie sur une liste d'instructions.

— **Incrément / décrément.** Les cas `Inc e` et `Dec e` utilisent `check_expr e TInt tenv` et implémentent les règles

$$\frac{\Gamma \vdash_\ell e : int}{\Gamma \vdash e{+}{+}} \qquad \frac{\Gamma \vdash_\ell e : int}{\Gamma \vdash e{-}{-}}$$

où $\vdash_\ell$ signifie « valeur gauche » (ici contrôlé au niveau de l'AST).

— **Conditionnelles et boucles.** Les cas `If(e,s1,s2)` et `For(e,s)` imposent que `e` soit de type booléen et vérifient récursivement les blocs, conformément aux règles

$$\frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash \texttt{if}\ e\, b_1\ \texttt{else}\ b_2} \qquad \frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash b}{\Gamma \vdash \texttt{for}\ e\, b}.$$

— **Affectations simples.** Pour une affectation générale `x1, ..., xn = e1, ..., en`, nous vérifions que le nombre de valeurs coïncide et que chaque expression a bien le type attendu par la valeur gauche correspondante (variable ou champ de structure), en appliquant la règle de typage des valeurs gauches du sujet.

Affectations multiples depuis un appel de fonction. Lorsque le membre droit est réduit à un seul appel de fonction `f(...)`, nous utilisons explicitement la règle de retours multiples du sujet :

$$\frac{(\Gamma \vdash_\ell e_i : \tau_i)_{e_i \neq \_} \quad \Gamma \vdash f(\vec{e}\,') \Rightarrow \tau_1, \ldots, \tau_n}{\Gamma \vdash e_1, \ldots, e_n = f(\vec{e}\,')}$$

Le typechecker vérifie l'arité de `f`, la compatibilité des types des paramètres et la correspondance entre les types de retour de `f` et les lvalues `e_i` du membre gauche.

Déclarations de variables. Le cas `Vars(ids, topt, s)` regroupe toutes les variantes de `var` décrites dans l'énoncé. Nous distinguons :

— **Type explicite.** Si `topt = Some ty`, nous vérifions d'abord que `ty` est bien formé (`check_typ`). Puis nous ajoutons chaque identifiant `id` (sauf `_`) à l'environnement local `tenv` avec ce type et nous typageons la suite `s` dans l'environnement enrichi.

— **Sans type explicite, avec initialisation.** Lorsque `topt = None`, nous imposons une initialisation, soit par une liste d'expressions, soit par un appel de fonction. Nous refusons explicitement l'utilisation de `nil` sans type explicite, comme dans la règle du sujet interdisant ce cas. Les types des variables sont alors déduits des expressions (ou des types de retour de la fonction) avant d'être ajoutés à `tenv`.

— **Sans type et sans initialisation.** Ce cas est interdit et produit l'erreur `"variable declaration without type requires initialization"`.

Instruction `return`. Le cas `Return el` suit les deux règles de l'énoncé :

— soit `el` est une liste `[e1;...;en]` : nous vérifions que chaque expression `ei` a le type attendu par la signature de la fonction courante ;

— soit `el` est réduit à un unique appel `f(e1,...,ek)` : nous vérifions d'abord que l'appel est bien formé (même code que pour `Call`) et nous comparons la liste des types retournés par `f` à la liste des types de retour attendus pour la fonction courante.

Vérification de `main`. En fin de `prog`, nous vérifions qu'il existe bien une fonction `main` dans `fenv`, sans paramètre et sans type de retour, comme demandé dans le sujet. En l'absence de `main`, ou si sa signature est incorrecte, une erreur est levée.

3.5 Bonus : analyse statique avancée dans `typechecker.ml`

Variables locales déclarées mais non utilisées. Pour chaque fonction, nous maintenons deux ensembles (représentés par des `Env.t` unitaires) :

— `declared_vars` : variables déclarées avec leur position ;

— `used_vars` : variables effectivement lues dans des expressions.

Au début de `check_function`, ces deux ensembles sont remis à vide, puis les paramètres et les identifiants de chaque `Vars` sont ajoutés à `declared_vars`, sauf `_`. Chaque occurrence de `Var x` dans `type_expr` enregistre `x.id` dans `used_vars`.

À la fin de la fonction, nous parcourons `declared_vars` et affichons un avertissement pour toute variable `x` telle que `x` $\in$ `declared_vars` mais `x` $\notin$ `used_vars`. La position enregistrée permet d'imprimer un message de la forme : `Warning: File "...", line ..., variable 'x' declared but not used`.

Cohérence de `import "fmt"` et `fmt.Print`. Le parseur renvoie un booléen `fmt` indiquant si `"fmt"` a été importé. À la fin de `prog`, nous effectuons un parcours supplémentaire de tout l'AST pour compter les occurrences du constructeur `Print`. Si `fmt` vaut `true` mais qu'aucun appel à `fmt.Print` n'apparaît, nous levons l'erreur `"import fmt declared but fmt.Print is never used"`. Inversement, si `fmt` vaut `false` mais qu'au moins un `Print` est présent, nous levons l'erreur `"fmt.Print used but import fmt not declared"`.

4 Partie 2 – Génération de code MIPS

4.1 Schéma de compilation retenu

L'objectif de cette partie du projet est de compiler un sous-ensemble du langage MicroGo vers du code MIPS en suivant les principes suivants :

- le résultat d'une expression est toujours placé dans le registre `$t0` ;
- la pile sert à stocker les valeurs intermédiaires et les arguments des appels ;
- la gestion mémoire des structures s'effectue sur le tas via `syscall 9` ;
- les retours éventuels d'une fonction sont également ramenés dans `$t0` (adresse de tuple en cas de retours multiples).

Représentation des valeurs

Les principales décisions de représentation sont résumées dans le tableau 1.

Type MicroGo	Représentation MIPS
`int`	entier signé 32 bits (registre ou mémoire)
`bool`	entier 0 (faux) ou non nul (vrai)
`string`	pointeur vers une chaîne en section `.data`
`*S`	pointeur (adresse) vers un bloc sur le tas
`nil`	entier 0 (pointeur nul)
n-uplet de retours	bloc alloué sur le tas, pointé par `$t0`

TABLE 1 – Représentation des valeurs MicroGo dans la cible MIPS.

Protocole d'appel et pile

Le protocole d'appel adopté est le suivant :

- les arguments sont empilés de droite à gauche avant le `jal` ;
- le résultat principal est dans `$t0` à la fin de la fonction ;
- `$ra` et `$fp` sont sauvegardés dans le prologue, puis restaurés dans l'épilogue.

La fonction OCaml `call_expr` implémente ce protocole :

Listing 7 – Protocole d'appel (`call_expr`).

```
1 and call_expr fname args =
```

```
  let rec push_args = function
    | [] -> nop
    | a :: q -> push_args q @@ tr_expr a @@ push t0
  in
  let argc = List.length args in
  push_args args
  @@ jal fname
  @@ (if argc = 0 then nop else addi sp sp (4 * argc))
```

4.2 Modèle de pile et tableaux d'activation

Le modèle de pile suit celui du cours : chaque fonction dispose d'un *tableau d'activation* autour de `$fp` contenant, dans l'ordre, les variables locales, les registres sauvegardés et les arguments.

−16	−12	−8	−4	0	+4	+8
c	b	a	$ra		x	y	z
vars locales			sauvegarde		arguments		

FIGURE 2 – Exemple de tableau d'activation autour de `$fp` (inspiré de la figure 1 du sujet).

Pour représenter l'environnement local, le type suivant est utilisé :

Listing 8 – Infos de variable locale et environnement courant

```
type var_info = { offset : int; typ : typ option }

let current_env : (string, var_info) Hashtbl.t ref =
  ref (Hashtbl.create 17)

let find_binding x =
  let env = !current_env in
  match Hashtbl.find_opt env x with
  | Some b -> b
  | None -> { offset = 0; typ = None }

let offset x = (find_binding x).offset
```

Les offsets des variables locales sont calculés par un pré-parcours du corps de la fonction (`alloc_vars_seq`), qui avance de -4 en -4 par rapport à `$fp`.

Le prologue et l'épilogue de fonction sont ensuite factorisés :

Listing 9 – Prologue et épilogue

```
let begin_ locals_size =
  push ra
  @@ push fp
  @@ move fp sp
  @@ (if locals_size = 0 then nop else addi sp sp (-locals_size))

let end_ =
  move sp fp
  @@ pop fp
  @@ pop ra
  @@ jr ra
```

4.3 Structures et allocation sur le tas

Plusieurs tables globales sont introduites :

— `struct_fields : (string, (ident * typ) list) Hashtbl.t`
— `struct_sizes  : (string, int) Hashtbl.t`
— `struct_offsets: (string * string, int) Hashtbl.t`

Elles sont remplies dans `tr_prog` en parcourant les déclarations de structures. La taille d'une structure est le nombre de champs multiplié par 4 octets, et chaque champ obtient un offset multiple de 4.

Expression `new(S)`. L'expression `new(S)` est compilée en un appel système `9` (allocateur du tas) avec la taille récupérée dans `struct_sizes` :

Listing 10 – Traduction de `new S`

```
| New s ->
    let size =
      match Hashtbl.find_opt struct_sizes s with
      | Some sz -> sz | None -> 0
    in
    li a0 size @@ li v0 9 @@ syscall @@ move t0 v0
```

Accès champ `e.x`. Pour l'accès `e.x`, le type de `e` est déterminé via une fonction d'inférence simple (`infer_expr_type`). Une fois le nom de structure connu, l'offset du champ est lu dans `struct_offsets` et la valeur est chargée à partir de l'adresse de `e` :

Listing 11 – Accès à un champ de structure

```
| Dot (expr, field) ->
    (* accs champ via offset *)
    let struct_name =
      match infer_expr_type expr with
      | Some (TStruct name) -> name
      | _ -> ""
    in
    let off =
      match Hashtbl.find_opt struct_offsets (struct_name, field.id) with
      | Some o -> o
      | None -> 0
    in
    tr_expr expr @@ lw t0 off t0
```

4.4 Retours multiples et affectations multiples

Le squelette ne gérait que le cas d'une valeur de retour. Pour les retours multiples, la solution retenue est celle des n-uplets stockés sur le tas :

— le type de retour d'une fonction est une liste `typ list` ;
— si cette liste contient plusieurs éléments, l'instruction `return` alloue un bloc de taille `4 * n` sur le tas et y stocke consécutivement les valeurs ;
— `$t0` reçoit l'adresse du n-uplet.

Les types de retour connus pour chaque fonction sont stockés dans :

```
let fun_ret_types : (string, typ list) Hashtbl.t = Hashtbl.create 17
```

et la fonction auxiliaire `arity f` renvoie la longueur de cette liste.

Instruction `return`

Le cas général est géré dans `tr_instr` pour la forme `Return el` :

Listing 12 – Retour multiple via tuple sur le tas.

```
| Return el ->
    let returns = !current_return_types in
    let ret_count = List.length returns in
    let rec count_values = function
      | [] -> 0
      | { edesc = Call (fn, _) ; _ } :: rest ->
          arity fn.id + count_values rest
      | _ :: rest -> 1 + count_values rest
    in
    let code =
      match ret_count with
      | 0 -> nop
      | 1 ->
          (* un seul rsultat : valeur directe dans $t0 *)
          (match el with
           | [] -> nop
           | [e] -> tr_expr e
           | _ -> nop)
      | _ ->
          (* construction d'un tuple sur le tas *)
          let produced = count_values el in
          if produced <> ret_count then nop
          else
            let size = 4 * ret_count in
            li a0 size
            @@ li v0 9
            @@ syscall
            @@ move t0 v0
            @@ move t3 t0
            @@ (let rec store_vals offset = function
                | [] -> nop
                | e :: rest ->
                    (match e.edesc with
                     | Call (fn, args) ->
                         let rc = arity fn.id in
                         (* cas d'un appel retournant plusieurs valeurs *)
                         ...
                     | _ ->
                         tr_expr e
                         @@ sw t0 offset t3
                         @@ store_vals (offset + 4) rest)
               in
               store_vals 0 el)
            @@ move t0 t3
    in
    code
    @@ move sp fp
    @@ pop fp
    @@ pop ra
    @@ jr ra
```

Affectations multiples

L'affectation multiple `x,y = ...` est traitée dans le cas `Set (lhs, rhs)`. Deux situations sont distinguées :

— affectation à partir d'un unique appel `f(...)` dont l'arité est strictement supérieure à 1 : `f` renvoie un tuple, lu directement champ par champ ;

— affectation à partir d'une suite d'expressions indépendantes : les valeurs du membre droit sont empilées puis dépilées pour chaque variable du membre gauche.

Listing 13 – Affectations multiples.

```
| Set (lhs, rhs) ->
    (* assignation multiple via tuple *)
    (match rhs with
     | [ { edesc = Call (fn, args) ; _ } ] when arity fn.id > 1 ->
         let ret_count = arity fn.id in
         if ret_count <> List.length lhs then nop
         else
           call_expr fn.id args
           @@ move t3 t0
           @@ (let rec assign_vals index = function
                | [] -> nop
                | dest :: rest ->
                    lw t0 (index * 4) t3
                    @@ store_lhs dest
                    @@ assign_vals (index + 1) rest
               in
               assign_vals 0 lhs)
     | _ ->
         (* cas gnral : expressions indpendantes *)
         ...
    )
```

La fonction auxiliaire `store_lhs` gère enfin l'écriture dans une variable simple ou dans un champ de structure.

4.5 Runtime et impression

Pour les appels à `fmt.Print`, la génération de code délègue à trois fonctions d'aide en MIPS : `print_int`, `print_string` et `print_bool`. La traduction d'une expression `Print [e1; ...; en]` effectue successivement `tr_expr ei` puis `jal print_*` en fonction du type statique de l'expression.

Le point d'entrée `_start` appelle `main` puis termine le programme avec `syscall 10`.

4.6 Compilation d'un programme complet

La fonction `tr_prog` assemble toutes les pièces :

— création d'un *builder* MIPS ;

— réinitialisation des tables globales (structures, types de retour) ;

— génération des fonctions d'aide pour `fmt.Print` ;

— définition du point d'entrée `_start` et compilation de toutes les fonctions déclarées.

Le bloc `runtime` définit le point d'entrée `_start`, effectue un `jal main` puis termine le programme avec le `syscall 10`.

5 Adaptation de `mips.ml`

Le fichier `mips.ml` fourni dans le squelette ne contenait que les définitions minimales d'instructions MIPS et les fonctions `print_program`. Il est étendu de manière modérée pour faciliter la génération de code dans `compile.ml`.

Registres supplémentaires

Outre `$t0`, `$t1`, `$a0`, `$v0`, `$sp` et `$ra`, plusieurs alias sont ajoutés : `$t2`, `$t3`, `$fp` et `$zero` :

Jeu d'instructions enrichi

Le jeu d'instructions de base est complété avec les opérations nécessaires au sujet 2 : soustraction, division entière et récupération de `lo`/`hi`, opérateurs logiques supplémentaires et comparaisons d'égalité/inegalité :

Builder pour le programme MIPS

Pour construire progressivement le segment texte et le segment données, un type `builder` est introduit, ainsi qu'une fonction de conversion finale vers `program`.

Constantes chaînes

Les chaînes MicroGo sont représentées par des labels dans le segment `.data`. La fonction `string_const` assure l'allocation unique d'une chaîne et renvoie son label :

Toutes les autres définitions du fichier `mips.ml` restent identiques au squelette initial.

6 Tests, résultats et difficultés rencontrées

6.1 Campagne de tests

Nous avons validé notre compilateur sur les programmes fournis dans le répertoire `tests/` ainsi que sur quelques exemples supplémentaires.

Programmes valides

Plusieurs programmes de test vérifient le bon fonctionnement global du compilateur :

— **arith.go**, **div.go** : expressions arithmétiques et opérateurs binaires ;
— **instr.go**, **min.go** : instructions de contrôle (`if`, `for`), `++`/`-` ;
— **multi.go**, **multi_return_stmt.go** : affectations multiples et fonctions à retours multiples ;
— **point.go** : structures, `new(S)` et accès champ `e.x` ;
— **var.go** : déclarations de variables et affectations simples ;
— **autosemi.go**, **verifvarbonus.go** : tests dédiés aux deux bonus (insertion automatique de `;` et variables locales non utilisées).

Programmes d'erreur

Nous avons également vérifié que les erreurs lexicales, syntaxiques et de typage sont bien détectées et signalées avec un message explicite :

— **interror.go** : entier trop grand pour `int64` ⇒ `literal constant too large` ;
— **lexerror.go** : caractère inconnu ⇒ `unknown character : @` ;
— **stringerror.go** : chaîne non terminée ⇒ `unterminated string literal` ;

— **multi_assign.go** : nombre incorrect de valeurs dans une affectation multiple ⇒ `wrong number of values in assignment` ;
— **test.go**, **testmain.go** : problèmes autour de `main` (absence ou mauvaise signature) ;
— **testfmt.go** : incohérence entre `import "fmt"` et l'utilisation de `fmt.Print`, testant le bonus de cohérence.

Ces tests montrent que les différents étages du compilateur (lexeur, parseur, typechecker) coopèrent bien pour détecter les erreurs prévues par le sujet.

6.2 Exemple de génération de code

Considérons le programme `var.go` :

```
package main;
import "fmt";

func main() {
     var x, y int;
     x = 1;
     y = 6;
     x = x+2;
     y = y*(x+4);
     fmt.Print(y)
};
```

Le compilateur génère (entre autres) le code MIPS suivant pour la fonction `main`, en respectant le protocole d'appel et la gestion de la pile :

```
_start:
  jal  main
  li   $v0, 10
  syscall

main:
  addi $sp, $sp, -4
  sw   $ra, 0($sp)
  addi $sp, $sp, -4
  sw   $fp, 0($sp)
  move $fp, $sp
  addi $sp, $sp, -8

  li   $t0, 1
  addi $sp, $sp, -4
  sw   $t0, 0($sp)
  lw   $t0, 0($sp)
  addi $sp, $sp, 4
  sw   $t0, -4($fp)

  li   $t0, 6
  addi $sp, $sp, -4
  sw   $t0, 0($sp)
  lw   $t0, 0($sp)
  addi $sp, $sp, 4
  sw   $t0, -8($fp)
  ...
```

On retrouve ici le schéma du tableau d'activation décrit dans le cours : sauvegarde de `$ra` et `$fp`, déplacement de `$fp`, réservation de l'espace pour les variables locales, puis accès à ces variables via des offsets fixes relatifs à `$fp`.

6.3 Difficultés rencontrées

Typechecker : gestion des cas avancés. La principale difficulté a été de couvrir systématiquement tous les cas du sujet dans `typechecker.ml`, en particulier :

- la gestion uniforme des fonctions à retours multiples, que ce soit dans les `Set` ou les `Return` ;
- l'inférence de type dans les déclarations `var x, y := ...` avec des appels de fonctions ;
- la prise en compte du cas particulier de `nil`, qui ne doit pas être accepté comme initialisation sans type explicite.

Compilation : tableau d'activation et retours multiples. Du côté de la génération de code, les points les plus délicats ont été :

- l'allocation correcte des offsets des variables locales, en respectant le schéma du tableau d'activation autour de `$fp` ;
- la mise en place des retours multiples via un n-uplet sur le tas, de manière cohérente avec les affectations multiples.

Cohérence entre fichiers et travail collaboratif. Enfin, une difficulté plus « ingénierie logicielle » a été de garder l'ensemble des fichiers cohérent entre eux (`mgolexer.mll`, `mgoparser.mly`, `typechecker.ml`, `compile.ml`, `mips.ml`, ...).

Pour gérer cela, nous avons travaillé avec un dépôt `git` commun et nous avons dû communiquer régulièrement pour nous assurer que nous avions la même vision des conventions (forme de l'AST, modèle de pile, choix de représentation des structures, etc.). Ce travail de synchronisation a été essentiel pour éviter les incohérences entre les différentes parties du compilateur.

7 Conclusion

Ce projet nous a permis de mettre en pratique l'ensemble de la chaîne classique d'un compilateur vue en cours : analyse lexicale, analyse syntaxique, typage statique puis génération de code MIPS. En implémentant nous-mêmes ces étapes à partir d'un squelette partiel, nous avons pu approfondir concrètement :

- les liens entre la grammaire formelle, l'AST et les règles de typage ;
- la manière dont les environnements (variables, fonctions, structures) sont construits et utilisés dans un typechecker ;
- le passage d'un langage de haut niveau vers un modèle machine explicite (pile, registres, tableau d'activation, appels de fonctions) ;

Plusieurs prolongements seraient possibles : petites optimisations (propagation de constantes, élimination de code mort), vérifications dynamiques (pointeur nul, dépassement de structures) ou extension du langage (tableaux, `range`, etc.). Ces points sortent du cadre des parties 1 et 2 mais pourraient être envisagés dans la suite du projet.