# Projet Micro-Go

**LDD3 IM | Université Paris-Saclay**
**Alexis VO & Théo G[redacted]**
**Lundi 8 décembre 2025**

---

# Rapport de Projet

Dans ce rapport, nous décrivons ce qui a été implémenté, ce qui fonctionne et les difficultés rencontrées ainsi que les extensions ajoutées.

---

# Partie 1 — Lexer, Parser et Typechecker

## 1.1 Lexer

Le fichier `mgolexer.mll` a été complété pour couvrir toutes les règles de tokenisation du Micro-Go.

Travail réalisé :

- Tokenisation complète : mots-clés, opérateurs, identifiants, nombres, chaînes, commentaires.
- Gestion correcte des séquences d'échappement ( `\n` , `\t` , `\"` …).
- **Bonus implémenté : insertion automatique des points-virgules** (comme en Go).
  Cela repose sur un booléen `need_semi_EOL` qui décide si un retour à la ligne doit générer un `;` implicite.

---

## 1.2 Parser

Le parser ( `mgoparser.mly` ) implémente l'intégralité de la grammaire du sujet et plusieurs règles intermédiaires.

Travail réalisé :

- Définition complète des règles syntaxiques Micro-Go.
- Priorités explicites pour éliminer les ambiguïtés.
- Gestion des trois formes de `for` , des appels de fonction, de `fmt.Print` , des expressions unaires/binaires, etc.
- **Résolution de six conflits shift/reduce** observés initialement.

Cf. annexe (captures d'écran du fichier .conflicts).

La plupart ont été éliminés via des déclarations de précédence et une factorisation du cas `=` vs `:=`.

---

## 1.3 Typechecker

Le fichier `typechecker.ml` assure une vérification statique complète :

- Vérification des opérations arithmétiques, logiques et comparaisons.
- Prise en charge des appels de fonctions à un ou plusieurs retours (même si la compilation multi-retour n'est pas implémentée ensuite).
- Vérification des accès champ, de `new`, de `nil`, des affectations simples/multiples et des déclarations `var` / `:=`.
- Gestion rigoureuse des environnements :
  - `fenv` — signatures des fonctions,
  - `senv` — structures,
  - `map_tenv` — environnement local final pour chaque fonction.

Le typechecker fournit les environnements nécessaires au compilateur.

---

# Partie 2 — Compilation en Assembleur MIPS

## 2.1 Generation du code MIPS (compile.ml)

Le fichier `compile.ml` traduit l'AST typé en assembleur MIPS.

Travail réalisé :

- Convention : le résultat d'une expression est dans `$t0`, la pile sert aux évaluations intermédiaires.
- Gestion complète des expressions arithmétiques, logiques, comparaisons, `new`, `Dot`, appels fonctionnels, etc.
- Traduction des structures :
  - allocation dynamique (syscall 9),
  - offsets des champs,
  - environnement structuré pour accéder correctement à un champ via `lw` / `sw`.
- Gestion des variables locales et paramètres via un **offset environment** généré par `build_offset_env`.
- Compilation des instructions :
  - `Set` (hors multi-assignations),
  - `Inc` / `Dec`,
  - blocs,
  - `If`,

    - les trois formes du `For`,
    - `Return`.
- Gestion de la `.data` :
    - `new_string` ajoute dynamiquement les chaînes,
    - traitement spécial pour `"\n"` et `" "`,
    - labels uniques via `new_label`.

**Les multi-assignations (multi-Set) ne sont pas implémentées dans la compilation**, même si elles sont typées correctement.

---

## 2.2 Mips.ml

Le fichier `mips.ml` fournit toutes les primitives utilisées pour générer le code assembleur :

- Instructions arithmétiques et logiques ( `add`, `mul`, `div`, `slt`, `seq`, etc.).
- Accès mémoire ( `lw`, `sw` ), branchements ( `b`, `bnez`, `beqz` ).
- Appels système ( `syscall 1`, `syscall 4`, `syscall 9`, `syscall 10` ).
- Outils de construction du programme ( `label`, `asciiz`, concaténation via `@@`, etc.).

---

# Tests réalisés et script d'automatisation

Nous avons également ajouté **des tests supplémentaires** pour mieux vérifier les comportements limites. Ils sont dans le dossier `/tests/` où l'on distingue aussi les tests d'erreurs dans `/tests/err`.

**Un script bash `run_tests.sh`** nous a permis d'exécuter automatiquement l'ensemble de notre batterie de tests.

Un script bash `build_s.sh` nous a permis de générer tous les fichiers `.s` de chaque fichier `.go`.

---

# Résultats des tests

Voici la synthèse complète :

## Programmes produisant la sortie attendue

| Fichier | Statut |
|---|---|
| `arith.go` | correct |
| `bool.go` | correct |
| `forforms.go` | correct |
| `min.go` | correct |
| `nilok.go` | correct |

| Fichier | Statut |
|---|---|
| `var.go` | correct |

## Programmes compilés mais ne produisant pas la sortie attendue

| Fichier | Problème constaté |
|---|---|
| `instr.go` | Produit `2` au lieu de `512` → **problème d'offset sur la pile**, lié aux `push` dans les expressions. |
| `point.go` | Mauvais adressage des champs dans les structs → **offsets incorrects dans Dot / Set.Dot**. |
| `structfield.go` | Même cause que `point.go`. |

## Programmes ne produisant pas de code (attendu ou normal)

| Fichier | Explication |
|---|---|
| `div.go` | Contient un `fmt.Print` sur une structure → **pas implémenté**. |
| `multiretassign.go` | multi-assignation non implémentée en compilation. |
| `multiretshortdecl.go` | idem. |
| `test.go` | Produit du code, mais **ne devrait pas** car il n'y a pas de `main` → non bloquant mais améliorable. |

---

# Bilan général

- **Tous les fichiers passent la partie 1** (lexing, parsing, typechecking), sauf éventuellement `test.go` qui manque un `main`.
- **Certains fichiers ne passent pas la partie 2**, pour des raisons identifiées :
  - offsets incorrects dans certains accès mémoire,
  - absence des multi-assignations,
  - absence du support d'impression des structures.

---

# Difficultés rencontrées

- Gestion complexe des offsets, en particulier lorsqu'un `push` est effectué en plein milieu de l'évaluation d'une expression ; ceci affecte `instr.go` et `point.go`.
- Résolution des conflits du parser autour de `=` et `:=`.
- Mise au point de l'allocation et de l'accès mémoire aux structures.
- Débogage MIPS long et peu guidé.
- Synchronisation entre les environnements du typechecker et les environnements d'offsets du compilateur.

---

# Limitations actuelles

- **Multi-assignations non implémentées** en compilation ( `x, y = f()` ).
- Impression de structures non supportée.
- Offset handling parfois incorrect à cause des pushes intermédiaires.
- `test.go` non rejeté en absence de `main` .

---

# Conclusion

Nous avons un compilateur **fonctionnel**, capable de :

- lexing / parsing complet,
- typage strict et détaillé,
- génération d'un assembleur MIPS,
- exécution correcte d'un grand nombre de programmes tests.

Certaines limites restent présentes, principalement liées à la gestion des offsets et des multi-assignations, mais la structure globale du compilateur est stable.

---