Rapport du projet de compilation:

Transformation d'un code source en un programme assembleur MIPS

Par Papa Talla Dioum

Table des matières

1. Introduction

Ce projet a pour objectif la réalisation complète d'un compilateur pour le langage MiniC, un sous-ensemble volontairement restreint du langage C. MiniC impose un typage fort, ne supporte ni pointeurs ni tableaux, et limite les constructions du langage afin de se concentrer sur les mécanismes fondamentaux de compilation.

Le compilateur développé traduit un programme MiniC en assembleur MIPS, en suivant une architecture classique en plusieurs phases : analyse lexicale, analyse syntaxique avec construction d'un arbre abstrait, vérification contextuelle, puis génération de code.
Le projet a été mené de manière progressive et incrémentale, chaque étape étant validée par des tests avant de passer à la suivante.

2. Organisation générale du compilateur

Le compilateur est structuré en plusieurs modules indépendants, chacun correspondant à une étape précise du processus de compilation. Cette séparation permet de raisonner clairement sur chaque phase, de faciliter le débogage et de tester les composants isolément.

L'analyse lexicale est réalisée à l'aide de Lex, l'analyse syntaxique avec Yacc, tandis que les passes de vérification et de génération de code sont écrites manuellement en C. Une bibliothèque utilitaire fournie (**miniccutils**) est utilisée pour la gestion de l'environnement, des symboles et de l'émission du code assembleur.

Le flux général est le suivant : le fichier source est d'abord transformé en tokens, puis analysé syntaxiquement afin de produire un arbre abstrait. Cet arbre est ensuite enrichi lors de la passe de vérification (types, portées, offsets mémoire), avant d'être parcouru une dernière fois pour produire le code MIPS final.

Le compilateur est organisé en plusieurs modules clairement séparés, mais les principales parties sur lesquelles ont a travaillées sont les suivantes:

- **lexico.l** : analyse lexicale (Lex)
- **grammar.y** : analyse syntaxique et construction de l'AST (Yacc)
- **defs.h** : définition des structures de nœuds de l'arbre
- **passe_1.c** : vérification contextuelle et typage
- **passe_2.c** : génération de code MIPS
- **scripts/** : scripts de tests automatisés
- **Tests/** : jeux de tests Syntaxe / Vérification / Génération de code

3. Analyse lexicale

La première étape du projet a consisté à compléter le fichier **lexico.l** afin de reconnaître l'ensemble des lexèmes du langage MiniC. Le rôle de cette étape est de transformer un flux de caractères en une suite structurée de tokens compréhensibles par l'analyse syntaxique.

Le fichier lexico.l a été complété pour :

- reconnaître les mots-clés (int, bool, void, if, while, for, do, print, etc.)
- reconnaître les identificateurs (TOK_IDENT)
- reconnaître les littéraux :
    - entiers (TOK_INTVAL)
    - booléens (TOK_TRUE, TOK_FALSE)
    - chaînes (TOK_STRING)

Une attention particulière a été portée à la gestion correcte des chaînes, car leur valeur devait être conservée jusqu'à la génération de code.

- l'initialisation correcte de yylval (notamment strval pour les chaînes)
- la gestion correcte de yylineno

L'analyse lexicale gère également les commentaires et les espaces, qui sont ignorés, et signale toute erreur lexicale (caractère invalide) avec un message explicite incluant le numéro de ligne. Cette précision est essentielle pour répondre aux exigences des tests automatiques.

4. Analyse syntaxique et construction de l'arbre abstrait

4.1 Grammaire MiniC

Le fichier grammar.y implémente la grammaire MiniC fournie dans le sujet, incluant :

- déclarations globales et locales
- blocs imbriqués
- instructions (if, while, do-while, for, print, expressions)
- expressions arithmétiques, logiques et bitwise
- affectations comme expressions

4.2 Construction de l'arbre abstrait

Chaque règle de grammaire construit un nœud de l'arbre syntaxique abstrait (AST) à l'aide de la fonction générique :

node_t make_node(node_nature nature, int nops, ...);

Tous les nœuds possèdent :

- nature : type du nœud
- lineno : numéro de ligne
- opr[] : tableau des fils
- nops : nombre de fils

Les champs spécifiques (ident, type, value, str, offset, etc.) sont remplis progressivement lors des passes suivantes.

5. Visualisation et validation de l'arbre

Afin de valider la construction correcte de l'arbre abstrait, des **dumps de l'AST** ont été générés après l'analyse syntaxique. Ces dumps sont produits au format `.dot` et convertis en PDF à l'aide de Graphviz.

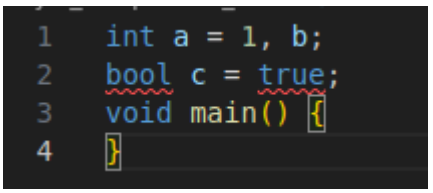

Fig 1: Code source utilisé pour valider la construction de l'arbre (Exo 3 TD)

Cette étape a permis de vérifier visuellement la structure des programmes analysés, notamment l'imbrication des blocs, la hiérarchie des expressions et la représentation des instructions de contrôle. Elle s'est révélée particulièrement utile pour corriger certaines erreurs de grammaire et de construction des listes (`NODE_LIST`).

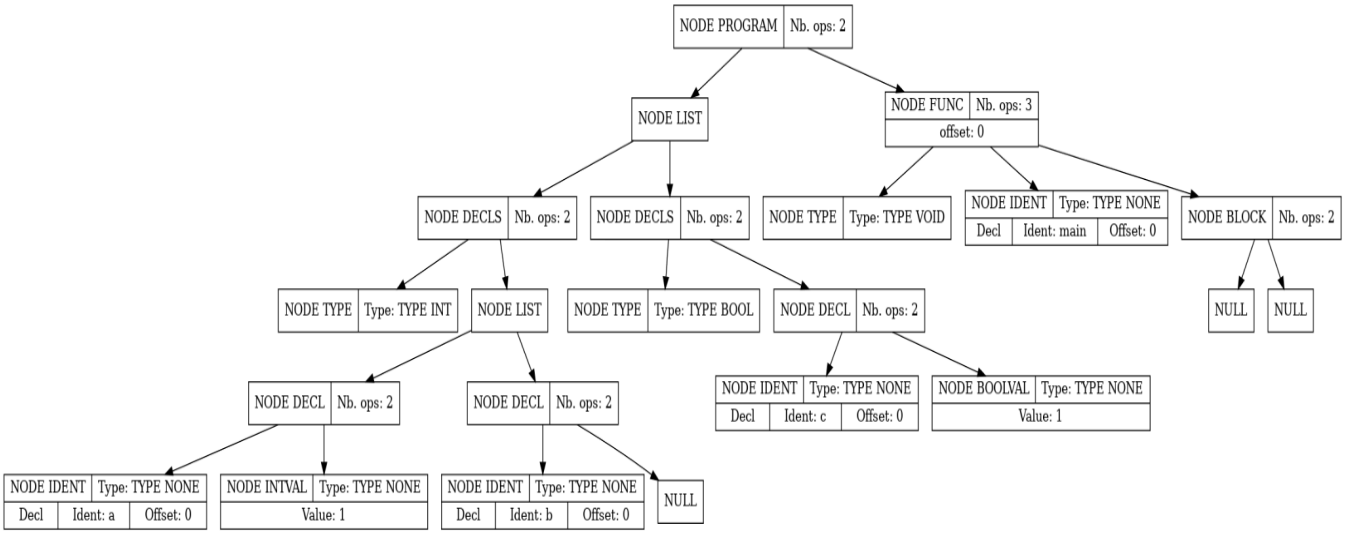

Fig 2: Arbre généré après syntaxe

L'arbre produit après l'analyse syntaxique représente uniquement la structure grammaticale du programme MiniC.
Il encode :

- l'imbrication des blocs,
- la hiérarchie des déclarations,
- la forme des expressions,
- l'ordre des instructions.

À ce stade :

- les types ne sont pas encore connus,
- les identificateurs ne sont pas reliés à leur déclaration,
- les offsets mémoire ne sont pas calculés,
- aucune règle sémantique n'est vérifiée.

Cet arbre est donc une vue purement syntaxique du programme.

Un second arbre est également généré après la passe de vérification, ce qui permet de comparer l'arbre "brut" et l'arbre enrichi (types, offsets, liens de déclaration).

6. Passe 1 - Vérification contextuelle

6.1 Objectifs

La passe 1 constitue le cœur sémantique du compilateur. Elle a pour rôle de vérifier que le programme est valide du point de vue contextuel et de préparer toutes les informations nécessaires à la génération de code:

- la résolution des identificateurs
- le respect des règles de portée (scopes)
- le typage fort des expressions
- la vérification des règles du langage MiniC

6.2 Environnements et scopes

Le compilateur utilise :

- un **contexte global**
- une pile de **contextes locaux** pour les blocs imbriqués

Fonctions clés :

- push_global_context
- push_context / pop_context
- env_add_element
- get_decl_node

Cette passe gère les portées lexicales à l'aide d'une pile de contextes. Un contexte global est créé pour les variables globales, puis des contextes locaux sont empilés et dépilés à l'entrée et à la sortie de chaque bloc. Cette organisation permet d'interdire les doubles déclarations dans un même bloc tout en autorisant le masquage (shadowing) dans des blocs imbriqués.

6.3 Résolution des identificateurs

Lors de l'analyse des déclarations, chaque identificateur est enregistré dans l'environnement avec son type, son caractère global ou local, et un offset mémoire. Les offsets sont calculés de manière incrémentale et seront utilisés directement lors de la génération de code.

La fonction resolve_idents_expr :

- associe chaque utilisation d'un identificateur à son nœud de déclaration
- remplit les champs :
    - decl_node
    - type
    - offset
    - global_decl

La résolution des identificateurs est effectuée en parcourant récursivement les expressions. Chaque occurrence d'un identificateur est reliée à son nœud de déclaration, ce qui permet de

récupérer son type et son emplacement mémoire. Toute utilisation d'une variable non déclarée est immédiatement signalée comme une erreur.

6.4 Typage des expressions

La fonction type_expr vérifie et assigne les types :

- opérations arithmétiques : int → int
- comparaisons : int → bool
- logique : bool → bool
- bitwise et shifts : int → int
- affectation : types strictement identiques

Le typage des expressions est également réalisé durant cette passe. Les règles de typage du langage MiniC sont strictement appliquées : les opérations arithmétiques s'appliquent uniquement aux entiers, les opérations logiques aux booléens, et aucune conversion implicite n notice. Toute incohérence de type provoque une erreur avec indication du numéro de ligne.

6.5 Cas particuliers gérés

- main doit s'appeler main et retourner void
- les conditions de if, while, for, do-while doivent être booléennes
- les chaînes (NODE_STRINGVAL) sont traitées sans typage, mais préparées pour la génération de code

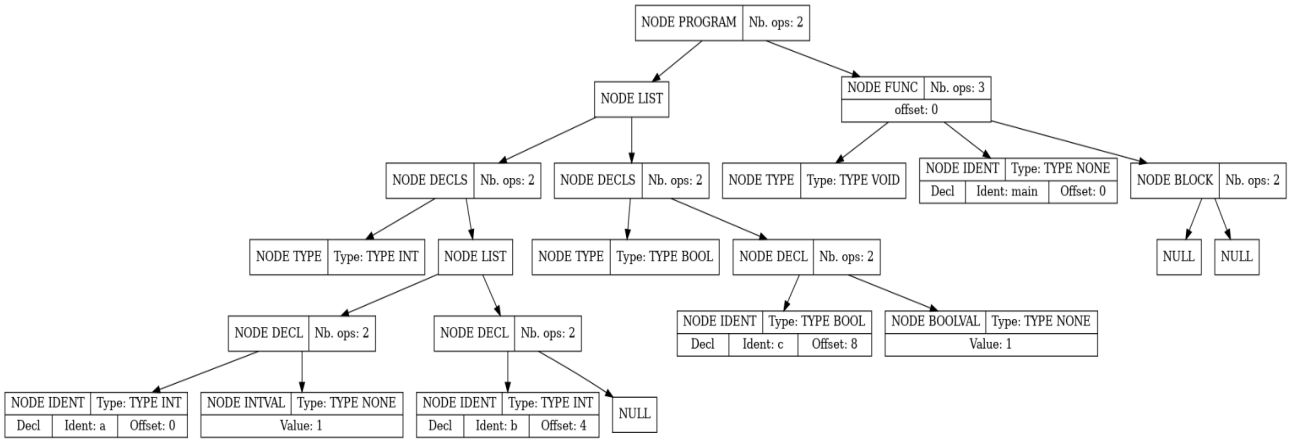

Fig 3: Arbre généré après Passe 1

La passe 1 ne modifie pas la structure, mais enrichit chaque nœud avec des informations sémantiques essentielles : types, résolution des identificateurs, offsets mémoire et gestion des portées.

Ces annotations permettent de détecter les erreurs contextuelles (déclarations multiples, variables non déclarées, incohérences de types) et constituent les données nécessaires à la génération correcte du code assembleur MIPS lors de la passe 2.

7. Passe 2 - Génération de code MIPS

La passe 2 est responsable de la traduction de l'arbre enrichi en code assembleur MIPS. Cette génération est réalisée par un parcours récursif de l'arbre, en s'appuyant sur les fonctions utilitaires fournies pour émettre les instructions MIPS.

7.1 Organisation générale

La passe 2 parcourt l'AST et génère du code MIPS en s'appuyant sur les primitives fournies :

- gestion des registres temporaires
- génération de labels
- appels système (print_int, print_string, exit)

7.2 Génération des expressions

Les expressions sont évaluées de manière récursive, en utilisant des registres temporaires pour stocker les résultats intermédiaires. Les affectations produisent les instructions de chargement et de stockage appropriées (lw, sw), en tenant compte des offsets calculés lors de la passe 1.

Les structures de contrôle (if, while, for, do-while) sont traduites à l'aide de labels générés dynamiquement et d'instructions de branchement conditionnel. La gestion des labels garantit l'unicité et la bonne lisibilité du code produit.

7.3 Instruction print

L'instruction print a fait l'objet d'un traitement particulier. Elle accepte une liste de paramètres pouvant être des expressions ou des chaînes. Les chaînes sont imprimées à l'aide du syscall MIPS dédié, tandis que les expressions sont évaluées puis affichées comme des entiers.

La fonction gen_print :

- parcourt une liste de paramètres
- distingue :
    - chaînes → syscall print_string
    - expressions → syscall print_int

Cette distinction repose sur les informations ajoutées lors de la passe 1.

À l'issue de la passe 2, le compilateur génère un fichier assembleur MIPS nommé **`out.s`**. Ce fichier constitue le résultat final du processus de compilation et correspond à une traduction directe du programme MiniC analysé, après vérification sémantique complète.

La passe 2 s'appuie sur l'arbre syntaxique enrichi par la passe 1 (types, offsets, portées, résolutions des identificateurs) pour produire un code assembleur correct et exécutable. Contrairement aux étapes précédentes, il ne s'agit plus d'analyse, mais de traduction.

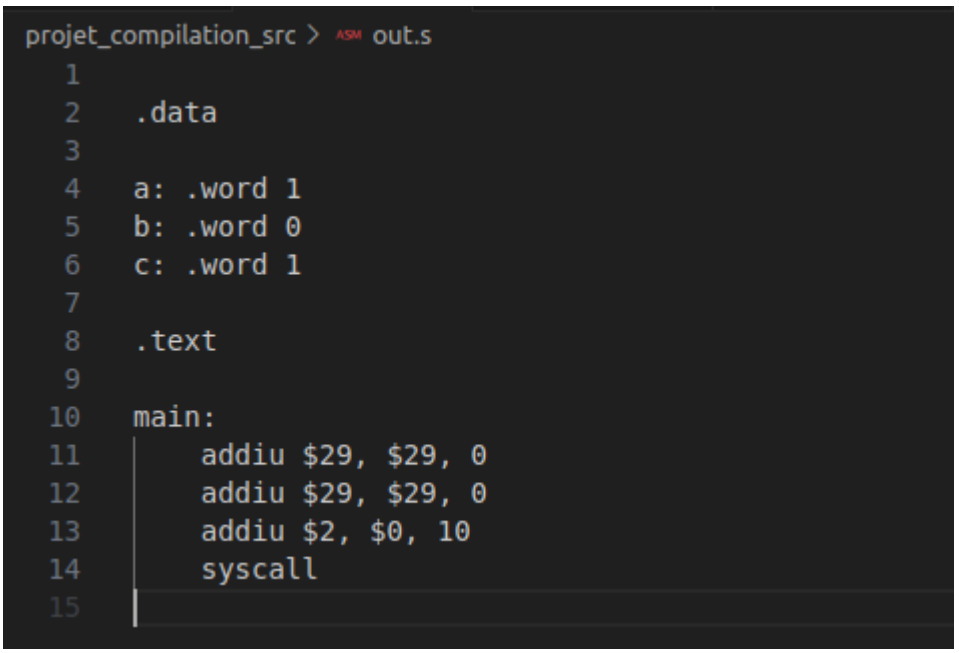

Fig 4: Code assembleur généré après passe 2 (Out.s Exo 3 TD)

Structure générale du fichier out.s

Le fichier généré est divisé en deux grandes sections, conformément aux conventions MIPS :

Section .data

.data
a: .word 1
b: .word 0
c: .word 1

Cette section contient les variables globales du programme.
Chaque variable est associée à :

- un label (son identificateur),
- une directive .word,
- une valeur initiale calculée ou fournie dans le programme source.

Les informations nécessaires à cette génération (nom, type, caractère global, offset) ont été déterminées lors de la passe 1.

Section .text

.text

main:
  addiu $29, $29, 0
  addiu $29, $29, 0
  addiu $2, $0, 10
  syscall

Cette section contient le code exécutable, correspondant ici à la fonction main, seule fonction autorisée dans MiniC.

On y retrouve :

- le label main, point d'entrée du programme,
- les instructions d'initialisation de la pile (même si l'offset est nul dans cet exemple),
- l'appel système de terminaison du programme (syscall avec $v0 = 10).

Chaque instruction est directement issue de la traduction des nœuds de l'arbre :

- déclarations → allocation mémoire,
- affectations → chargements et stockages (lw, sw),
- expressions → calculs dans les registres,
- instructions de contrôle → branchements conditionnels,
- appels à print → appels système MIPS appropriés.

Le fichier out.s est donc la matérialisation finale de toutes les décisions prises lors des étapes précédentes.

Sa structure bas niveau permet de vérifier visuellement la correspondance entre le programme source MiniC et son exécution au niveau machine.

8. Tests et validation

Les tests ont été développés progressivement tout au long du projet. Ils sont organisés conformément aux consignes dans une arborescence séparant les tests de syntaxe, de vérification et de génération de code, chacun décliné en cas valides (OK) et invalides (KO).

Un script run_tests.sh permet d'exécuter l'ensemble des tests avec les options appropriées (`-s`, `-v`, ou sans option) et de vérifier le comportement attendu du compilateur. Les tests de génération de code s'appuient sur l'exécution du code MIPS produit et sur l'observation de la sortie affichée.

9. Fonctionnalités implémentées et limites connues

Le compilateur implémente l'ensemble des fonctionnalités requises par le sujet : analyse lexicale et syntaxique complète, gestion des portées, typage fort, génération de code MIPS fonctionnelle et support des principales structures de contrôle.

Les principales limitations concernent l'absence d'optimisations, la non-détection statique de certaines erreurs à l'exécution, et la dépendance à l'émulateur MIPS pour certains comportements. Ces choix sont assumés et cohérents avec les objectifs pédagogiques du projet.

10. Conclusion

Ce projet a permis de mettre en pratique l'ensemble des concepts fondamentaux de la compilation, depuis l'analyse lexicale jusqu'à la génération de code assembleur. L'approche incrémentale adoptée, combinée à une stratégie de tests systématique, a permis d'aboutir à un compilateur fonctionnel, robuste et conforme aux spécifications du langage MiniC.

Le travail réalisé constitue une base solide, à la fois pour une évaluation académique et pour d'éventuelles extensions futures du langage ou du compilateur.