Enki MARTIN
Antoine HEBERT

Projet Compilation

2025-2026

I. Introduction

Ce projet fait partie du développement d'un mini-compilateur pour un langage impératif simplifié, proche du C. Il couvre toutes les étapes classiques d'un compilateur : l'analyse lexicale, l'analyse syntaxique, la vérification sémantique et enfin la génération de code cible en MIPS.

Le but de ce compte-rendu n'est pas de refaire une description complète du compilateur ni de rappeler en détail les règles du langage, qui sont déjà bien expliquées dans l'énoncé et les supports de cours. L'idée est plutôt de présenter les choix techniques faits pendant le projet, la manière dont les outils fournis ont été compris et utilisés, ainsi que les raisons qui ont conduit à certaines décisions d'implémentation plutôt qu'à d'autres.

Une attention particulière a été donnée à la compréhension de l'architecture globale du compilateur, en particulier grâce à l'étude des structures de données et des fonctions utilitaires mises à disposition. Les différentes parties du projet seront donc abordées en mettant l'accent sur la méthode de travail adoptée, les difficultés rencontrées et les solutions mises en place pour les résoudre.

II. Lexico.l

L’implémentation de l’analyse lexicale a été la première vraie étape concrète du projet. Contrairement à d’autres parties du compilateur, elle ne démarrait pas totalement de zéro : un squelette de fichier ainsi que quelques exemples étaient déjà fournis dans les supports de cours. Le but n’était donc pas d’inventer un analyseur lexical complet, mais surtout de bien comprendre ce qui était attendu et de compléter ce qui manquait de façon cohérente, ce qui n’étais pas si évident au début.

Approche générale

La démarche suivie a été principalement déductive. Les diapositives du cours donnaient certaines règles isolées, notamment pour les identificateurs (IDF), mais sans fournir une liste complète de tous les tokens du langage. Plutôt que de faire une implémentation approximative, j’ai préféré partir de ces exemples pour en déduire des règles plus générales, puis reconstruire progressivement l’ensemble du lexique, ce qui a pris un peut de temps.

Les identificateurs ont servi de point de départ. Même si leur définition dans les slides était partielle, elle donnait assez d’indices pour comprendre la logique globale (lettre initiale, chiffres possibles, underscore). À partir de cette base, les autres catégories lexicales comme les entiers, les chaînes de caractères, les opérateurs et les mots-clés ont été définies en appliquant les mêmes principes.

Reconstruction du lexique

Une fois les premières règles comprises, j’ai entamé une phase de « reverse engineering » du langage attendu. Pour cela, j’ai listé tous les tokens utilisés dans la grammaire (grammar.y) ainsi que dans les définitions (defs.h), puis j’ai construit une table reliant :

- les symboles du langage (par exemple +, ==, &&, print, etc.)
- les tokens retournés par l’analyse lexicale (TOK_PLUS, TOK_EQ, TOK_AND, …)
- leur rôle syntaxique et sémantique

Cette table, d’abord faite sur papier puis retravaillée sur ordinateur, m’a permis de vérifier la cohérence globale du lexique et d’éviter les oublis, même si au début elle n’était pas très clair.

Cas particuliers et choix techniques

Certains points ont demandé une attention particulière :

- **Les entiers** : le langage accepte des entiers décimaux et hexadécimaux. La reconnaissance lexicale a été séparée de la conversion, celle-ci étant réalisée avec strtol, ce qui permet de s’appuyer sur la bibliothèque standard sans refaire ce travail.
- **Les chaînes de caractères** : seules les chaînes composées de caractères imprimables et de séquences d’échappement classiques (\", \\, \n) sont acceptées. Leur décodage réel est volontairement différé, ce qui simplifie l’analyse lexicale et évite de la rendre trop complexe, même si ça fait une étape en plus après.
- **Les commentaires** : les commentaires de type // sont ignorés directement lors de l’analyse lexicale, ce qui simplifie beaucoup les étapes suivantes.
- **Les erreurs lexicales** : toute séquence invalide provoque immédiatement une erreur avec le numéro de ligne, ce qui correspond exactement aux attentes du sujet, et évite des bugs plus loin.

Organisation et lisibilité

Un effort particulier a été fait pour rendre le fichier lexico.l lisible. Les règles sont regroupées par catégories (mots-clés, opérateurs, identificateurs, constantes), ce qui facilite la relecture et la maintenance du fichier. Cette organisation correspond aussi à la table de correspondance utilisée lors de la phase de conception, ce qui rend l'ensemble plus cohérent, même si ce n'est pas obligatoire.

De plus, le fichier utilise des macros comme LETTRE, CHIFFRE, etc., afin de factoriser les règles et de rendre la structure des tokens plus explicite. Ce choix améliore à la fois la clarté du code et la robustesse de l'analyse lexicale, bien que ça rajoute un peu de verbosité.

Bilan

L'analyse lexicale a surtout été un travail de compréhension et d'organisation, plutôt qu'un simple exercice d'implémentation. Le fait de ne pas avoir une spécification totalement exhaustive a obligé à réfléchir en profondeur sur le langage cible. Cette réflexion a ensuite servi de base solide pour les autres phases du compilateur, ce qui a clairement facilité la suite du projet même si ça n'a pas toujours été simple.

III. Grammar.y

Dans grammar.y, chaque règle de grammaire sert à construire un morceau de l’arbre du programme.

À cahque fois qu’une règle est reconnue, j’utilise des fonctions comme make_node, make_node_2, make_node_3, make_node_4ou encore des fonctions utilitaires comme noed_LIST, noed_AFFECT, noed_WHILE, etc.

Ces fonctions servent à créer des nœuds qui représentent les différentes structures du langage.

Chaque nœud correspond à une élément précis du programme et possède un type, par exemple : NODE_IF pour une condition, NODE_PLUS pour une addition, NODE_IDENT pour une variable ou NODE_BLOCK pour un bloc d’instructions.

Dans grammar.y, les tokens (TOK_INT, TOK_BOOL, TOK_IDENT, TOK_INTVAL…) et les déclarations %union et %type servent à transporter les informations pendant l’analyse du programme.

Grâce à ça, chaque règle peut récupérer ce qui a déjà été reconnu, créer un nouveau nœud, et construire l’arbre syntaxique.

IV. Visualisation de l’arbre

Lors de la création de l’arbre, un fichier apres_syntaxe.dot est automatiquement généré.

Ce fichier contient la description complète de l’arbre syntaxique au format Graphviz.

Par exemple, on peut y trouver des lignes comme :

N1 [shape=record, label="{{NODE PROGRAM|Nb. ops: 2}}"];

N2 [shape=record, label="{{NULL}}"];

edge[tailclip=true];

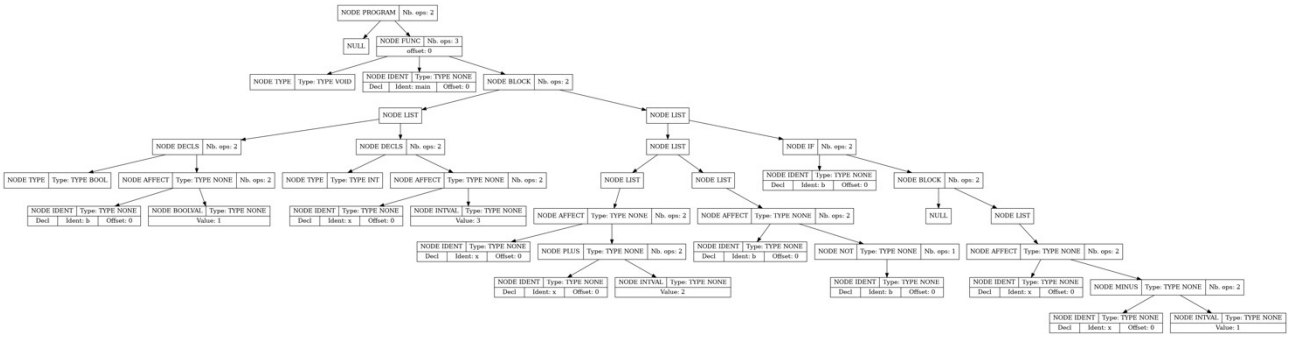
N1 -> N2

Ce fichier ne sert pas directement à l’exécution, mais à visualiser la structure du programme.

Ensuite, on transforme ce fichier en image avec la commande :

dot -Tpng apres_syntaxe.dot -o arbre.png

Cela permet d’obtenir une image de l’arbre syntaxique, ce qui aide beaucoup à mieu comprendre comment le programme est analysé et organisé par le compilateur.



Cette arbre est la représentation du code de l’exercice 4 du TD qui est le suivant :

```
1  void main() {
2    int a = 120, b = 80;
3    if (a > b) {
4      a = a - b;
5    }
6    print("a = ", a, " - b = ", b);
7  }
```

V. Première passe

La première passe du compilateur correspond vraiment au cœur logique du projet. Contrairement à l'analyse lexicale ou syntaxique, qui reposent surtout sur des outils automatiques comme flex et bison, la passe sémantique demande une compréhension globale du langage, de l'arbre abstrait généré et des règles de cohérence à respecter. C'est à ce moment-là que le compilateur commence réellement à « comprendre » ce que fait le programme analysé, et pas seulement à le lire.

Les premiers choix

Le choix principal qui a guidé l'implémentation de passe_1.c a été de suivre de très près la structure de la grammaire. Plutôt que de gérer les erreurs au fur et à mesure qu'elles apparaissaient, la démarche a consisté à relire entièrement le fichier grammar.y afin de lister tous les types de nœuds pouvant apparaître dans l'arbre syntaxique. Cette approche a permis de s'assurer qu'aucun cas n'était oublié et que chaque construction du langage était traitée correctement, ce qui évite beaucoup de problèmes plus tard.

Un point particulièrement important dans cette passe a été la compréhension du fichier miniccutils.h. Ce fichier regroupe toutes les fonctions utilitaires prévues pour la gestion des contextes, des symboles, des types et des offsets mémoire. Même s'il peut être tentant d'implémenter la passe de manière un peu empirique, il est vite apparu que la logique attendue par le projet était déjà largement décrite dans cette interface. Une partie non négligeable du travail a donc consisté à lire ce fichier attentivement, ligne par ligne, pour comprendre non seulement ce que font les fonctions, mais surtout comment elles sont censées s'articuler entre elles.

La gestion des portées

La gestion des portées a été un aspect central de cette passe. Le choix a été fait d'utiliser une pile explicite de contextes afin de représenter correctement l'imbrication des blocs (NODE_BLOCK), du programme principal et des fonctions. À chaque entrée dans une nouvelle portée, un nouveau contexte est créé et empilé, puis détruit lors de la sortie de cette portée. Cette méthode permet une résolution correcte des identificateurs tout en autorisant la redéfinition légale de variables dans des blocs internes, ce qui est un comportement attendu dans un langage de type C.

La résolution des identificateurs repose sur une recherche ascendante dans la pile de contextes. Cette stratégie permet de respecter le masquage lexical tout en gardant une implémentation relativement simple. À chaque utilisation d'un identificateur, la passe vérifie systématiquement qu'il a bien été déclaré auparavant, ce qui permet de détecter rapidement les erreurs de type « variable non définie », même dans des cas un peu complexes.

Le détail des expressions

Le typage des expressions a été traité de manière méthodique. Plutôt que de gérer chaque opérateur individuellement dans le corps principal de la passe, des fonctions auxiliaires ont été utilisées pour déterminer le type résultant des opérateurs unaires et binaires. Cette factorisation permet de réduire la duplication de code et de rendre plus explicite les règles de compatibilité entre les types du langage. Dès qu'une incompatibilité est détectée, une erreur est levée immédiatement avec un message clair incluant le numéro de ligne.

Les structures de contrôle comme if, while, for et do-while ont également fait l'objet d'une attention particulière. Pour chacune d'elles, la passe vérifie systématiquement que la condition associée est bien de type booléen, quelle que soit la complexité de l'expression. Cela permet d'éviter des erreurs sémantiques subtiles qui pourraient sinon apparaître plus tard lors de la génération de code.

Le traitement des déclarations a aussi nécessité une logique spécifique. Les déclarations multiples, parfois combinées avec des initialisations, ont été analysées de manière itérative afin de garantir à la fois la cohérence des types et l'unicité des identificateurs dans une même portée. Cette approche se rapproche du comportement d'un compilateur réel, où les erreurs de redéfinition ou d'initialisation incorrecte doivent être signalées immédiatement, sans attendre les phases suivantes.

Enfin, la passe 1 a été conçue comme une phase purement vérificatrice, sans produire directement de code. Elle n'a pas d'effet visible en sortie, mais enrichit l'arbre syntaxique avec des informations essentielles comme les types, les liens vers les déclarations et les offsets futurs. Cette séparation claire des rôles simplifie fortement la passe suivante et rend l'architecture globale du compilateur plus lisible et plus robuste, même si tout n'est pas parfait.

En résumé, l'implémentation de passe_1.c a été guidée par une volonté de rigueur et de couverture complète des cas possibles. En s'appuyant sur une lecture attentive de la grammaire et de miniccutils.h, et en traitant chaque nœud de l'arbre de manière systématique, cette passe fournit une base sémantique solide sur laquelle la génération de code peut ensuite s'appuyer sans trop de surprises.

VI. Deuxième passe

La seconde passe du compilateur correspond au moment où l'on passe d'un raisonnement abstrait à quelque chose de concret : la génération de code MIPS. Même si cette étape peut sembler très différente de la passe sémantique, un point important est vite apparu : la structure globale du programme reste quasiment la même que dans la passe 1. Ce parallèle n'est pas un hasard et a largement influencé la façon dont la passe 2 a été implémentée.

Un élément clé a été de comprendre que la génération de code reste avant tout une traversée de l'arbre syntaxique abstrait. Comme pour la passe 1, l'implémentation repose sur une grande structure en switch/case parcourant récursivement les mêmes types de nœuds. La vraie différence se situe dans l'objectif : là où la passe 1 vérifie et annote l'arbre, la passe 2 produit des instructions MIPS en s'appuyant sur les fonctions fournies par miniccutils.h.

Rôle de miniccutil.h

Comme pour la passe sémantique, le fichier miniccutils.h joue un rôle central dans cette étape. Il fournit toutes les primitives nécessaires à la génération du code MIPS, que ce soit pour la gestion des registres temporaires, de la pile, des labels ou encore des appels système. Une grande partie du travail a donc consisté à comprendre cette interface et son fonctionnement, plutôt qu'à écrire directement de l'assembleur MIPS à la main.

Ce choix impose un cadre assez strict, mais il permet aussi de garantir une génération de code homogène et cohérente sur l'ensemble du projet. Une fois cette abstraction comprise, l'écriture de la passe 2 devient beaucoup plus mécanique et prévisible.

Gestion de la pile

La gestion de la pile a été l'un des premiers points structurants de cette passe. Le choix a été fait de centraliser l'allocation et la libération du stack frame au niveau du nœud NODE_PROGRAM. Concrètement, toute la mémoire nécessaire aux variables et aux temporaires est réservée au début du programme, puis libérée proprement à la fin.

Cette approche simplifie fortement la logique globale, car il n'est pas nécessaire de gérer des allocations dynamiques à chaque bloc. La réinitialisation des offsets temporaires avant le début de l'analyse permet également de garantir une allocation correcte et prévisible tout au long de la génération du code.

Gestion des registres et expressions

La génération des expressions arithmétiques et logiques repose sur une discipline simple mais stricte concernant les registres. Chaque sous-expression réserve un registre temporaire, y place son résultat, puis libère ce registre une fois qu'il n'est plus utile. Cette convention permet d'éviter les conflits de registres sans avoir à mettre en place un véritable allocateur complexe, ce qui aurais largement dépassé le cadre du projet.

Le fait que cette logique soit quasiment identique pour la majorité des opérateurs binaires a grandement facilité l'implémentation. Cela rend aussi le code plus lisible et plus facile à vérifier, car le schéma général reste toujours le même.

Opérateurs complexes et comparaisons

Les opérateurs plus délicats, comme la division et le modulo, ont nécessité un traitement spécifique. Le choix a été fait d'intégrer explicitement une vérification de la division par zéro à l'aide de l'instruction teq. Cette vérification est placée juste avant l'instruction de division, ce qui permet de respecter les contraintes sémantiques du langage et d'éviter toute génération de code incorrect.

Les comparaisons et opérateurs logiques sont traduits de manière assez directe en instructions MIPS (slt, xor, xori, etc.). L'objectif ici n'était pas d'optimiser le code produit, mais de générer un code correct, clair et

systématique. Les différentes comparaisons (<, <=, ==, !=, etc.) sont obtenues par combinaison d'instructions simples, ce qui rend leur fonctionnement facile à comprendre.

Structures de contrôle

Les structures de contrôle comme if et while sont implémentées à l'aide de labels générés dynamiquement. Chaque structure crée ses propres labels de branchement, ce qui évite toute collision, même dans le cas de structures imbriquées. Cette méthode reflète directement la logique du programme source et permet de suivre assez facilement le flot d'exécution dans le code MIPS généré.

Instruction print

Le traitement de l'instruction print illustre bien la philosophie générale adoptée pour cette passe. Chaque paramètre est traité séparément, ce qui correspond exactement à la sémantique du langage. Les chaînes de caractères sont placées dans la section .data et référencées via des labels globaux, tandis que les expressions numériques sont évaluées puis passées dans les registres attendus par les syscalls MIPS.

Cette séparation claire entre chaînes et valeurs numériques permet de garder une implémentation simple tout en restant fidèle au comportement attendu du langage.

Robustesse

Enfin, comme pour la passe 1, la génération de code a été pensée de manière défensive. Les nœuds non pertinents pour cette passe sont simplement ignorés, ce qui permet de parcourir l'arbre sans imposer une correspondance parfaite entre chaque construction syntaxique et une génération d'instructions. Cette tolérance rend la passe plus robuste et plus facile à faire évoluer.

En conclusion, la passe 2 s'est révélée moins complexe qu'elle ne le paraissait au départ, une fois le parallèle avec la passe 1 bien compris. Malgré la différence apparente entre vérification sémantique et génération de code MIPS, les deux phases reposent sur une structure conceptuelle très proche. Cette similarité a permis de réutiliser les raisonnements développés précédemment et de produire une génération de code cohérente, systématique et conforme aux attentes du projet.

VII. Conclusion

Ce projet de compilation a été un exercice très formateur, aussi bien sur le plan technique que sur la manière de travailler. Il a permis d'appliquer concrètement des notions vues en cours qui peuvent parfois sembler assez théoriques, comme l'analyse lexicale, syntaxique et sémantique, ainsi que la génération de code bas niveau. Le fait d'avancer étape par étape a obligé à construire une compréhension progressive et cohérente du langage et de son fonctionnement interne.

L'un des principaux apprentissages de ce projet a été l'importance de bien lire et comprendre les interfaces fournies, en particulier le fichier miniccutils.h. Même s'il était peu documenté, ce fichier contenait en réalité la majorité des mécanismes attendus et imposait un cadre précis pour réfléchir aux différentes passes du compilateur. Une fois cette contrainte acceptée, il est devenu un véritable support pour structurer correctement le travail.

Le choix d'implémenter les passes 1 et 2 de manière très similaire, en s'appuyant sur un parcours systématique de l'arbre syntaxique abstrait, s'est avéré efficace. Cette cohérence a facilité la transition entre la vérification sémantique et la génération de code, tout en rendant le projet plus lisible et plus simple à maintenir. Elle montre aussi qu'un compilateur n'est pas une suite de blocs indépendants, mais plutôt une série de transformations successives appliquées à une même structure.

Au final, l'ensemble des fonctionnalités demandées a été implémenté et testé, et le compilateur génère un code MIPS fonctionnel et conforme aux spécifications. Au-delà du résultat obtenu, ce projet a surtout permis de mieux comprendre ce qui se passe réellement à l'intérieur d'un compilateur, et de réaliser que des mécanismes complexes peuvent être construits de manière progressive, logique et organisée.

Enfin, il serait difficile de conclure sans évoquer les éléments extérieurs ayant aidé à mener ce projet jusqu'au bout : beaucoup de persévérance, quelques nuits un peu trop courtes, et un soutien moral inattendu mais efficace — avec une mention spéciale aux glaçons qui fondent, véritables sources de motivation lors des moments les plus intenses du développement.