Projet

Compilation

Karim **Qorar**
Mohamed **Ghaibouche**

Sommaire

Introduction

Objectif et Réalisation

Dans le cadre de ce projet, nous avons eu pour objectif de développer un compilateur pour le langage MiniC. Nous avons abordé les étapes essentielles à la compilation : **l'analyse lexicale**, **l'analyse syntaxique**, **les vérifications sémantiques** et **la génération de code assembleur**. Les différentes étapes ont pour but de transformer les suites de caractères d'un fichier en langage MiniC en un fichier équivalent en langage assembleur MIPS, tout en s'assurant que le code respecte bien la syntaxe MiniC. Nous avons pu implémenter l'intégralité du compilateur, y compris les différents arguments tels que la limitation du nombre de registres. Il n'existe pas de **bugs** connus de notre part à ce jour. Cependant, nous avons potentiellement trouvé un problème avec *'minicc_ref'*.

Utilisation

Liste des commandes utiles au projet:

- ***./minicc file.c*** : Compile le fichier *'file.c'* en un fichier *'out.s'*. La **liste des arguments** est disponible en utilisant la commande *'./minicc -h'*
- ***python3 Tests/Tests.py*** : Lance les fichiers de test. La liste des arguments est détaillée dans la section **implémentation** du compte-rendu
- ***java -jar Tests/mars_4_2.jar nc fichier.s*** : Lance le fichier assembleur *'fichier.s'* à l'aide de l'outil **MARS**, développé par l'université du Missouri

D'autres précisions sur les commandes seront donnés tout au long du compte rendu.

Crédits et Remerciements

Ce projet a été développé sous la direction de Monsieur Quentin Meunier, à qui reviennent les crédits pour la conception originale et la mise en place des spécifications du projet de compilateur. Nous tenons à le remercier, lui ainsi que Monsieur Olivier Marchetti, pour leur soutien et leur encadrement tout au long du projet.
Pour la rédaction du rapport, nous avons utilisé **chatGPT** afin de corriger les éventuelles fautes d'orthographes.

Analyse Lexicale page suivante

Implémentations principales

Analyse lexicale

L'analyse lexicale constitue la première étape du processus de compilation. Cette étape vise à transformer un **fichier** contenant un algorithme codé en langage MiniC en une suite de **lexèmes (tokens)**, qui représentent les blocs de base du langage.

Pour réaliser l'implémentation de l'analyse lexicale, nous avons utilisé l'outil **Lex**, qui permet de développer des analyseurs lexicaux efficacement et facilement. Afin de mener à bien l'analyse lexicale, il a été nécessaire d'établir des règles permettant de reconnaître les différentes parties du langage MiniC.

Les éléments du langage sont identifiés par des tokens spécifiques à MiniC, tels que *'int'*, *'bool'*, *'if'*, *'else'*. Les identificateurs, qui correspondent aux noms attribués aux différentes variables, sont détectés grâce à des séquences valides de lettres et de chiffres. Les opérateurs sont quant à eux immédiatement reconnus grâce à des règles spécifiques.

```
int start = 0;
int end = 100;

void main() {
    int i, s = start, e = end;
    int sum = 0;
    for (i = s; i < e; i = i + 1) {
        sum = sum + i;
    }
    print("sum: ", sum, "\n");
}
```

Figure 1: Programme MiniC sous forme de code, avant l'analyse lexicographique

TOK_INT 2	TOK_IDENT 2 'start'	TOK_AFFECT 2	TOK_INTVAL 2 '0'	TOK_SEMICOL 2	TOK_INT 3
TOK_IDENT 3 'end'	TOK_AFFECT 3	TOK_INTVAL 3 '100'	TOK_SEMICOL 3	TOK_VOID 5	TOK_IDENT 5 'main'
TOK_LPAR 5	TOK_RPAR 5	TOK_LACC 5	TOK_INT 6	TOK_IDENT 6 'i'	TOK_COMMA 6	TOK_IDENT 6 's'
TOK_AFFECT 6	TOK_IDENT 6 'start'	TOK_COMMA 6	TOK_IDENT 6 'e'	TOK_AFFECT 6	TOK_IDENT 6 'end'
TOK_SEMICOL 6	TOK_INT 7	TOK_IDENT 7 'sum'	TOK_AFFECT 7	TOK_INTVAL 7 '0'	TOK_SEMICOL 7
TOK_FOR 8	TOK_LPAR 8	TOK_IDENT 8 'i'	TOK_AFFECT 8	TOK_IDENT 8 's'	TOK_SEMICOL 8
TOK_IDENT 8 'i'	TOK_LT 8	TOK_IDENT 8 'e'	TOK_SEMICOL 8	TOK_IDENT 8 'i'	TOK_AFFECT 8
TOK_IDENT 8 'i'	TOK_PLUS 8	TOK_INTVAL 8 '1'	TOK_RPAR 8	TOK_LACC 8	

Figure 2: Programme MiniC sous forme de lexèmes, après l'analyse lexicographique

Analyse syntaxique page suivante

Analyse syntaxique

À la fin de l'analyse lexicale, le fichier source est transformé en une suite de Tokens, il est alors nécessaire de vérifier que la suite de Tokens est arrangée de manière à respecter les règles grammaticales du langage miniC. Pour réaliser cette analyse syntaxique nous allons utiliser l'outil **Yacc** qui nous faciliterait grandement la tâche.

L'analyse syntaxique a pour objectif principal de réordonner la suite de Token récupéré lors de l'étape précédente afin de pouvoir obtenir une structure arborescente, cette structure représente la structure hiérarchique du programme contenu dans le fichier qui est en train d'être compilé.

Pour l'analyse syntaxique de notre compilateur miniC, nous utilisons Yacc pour créer des règles qui définissent comment organiser les tokens (éléments de base du code) en structures valides comme les commandes ou les boucles. Ces règles aident à construire un **arbre syntaxique**, qui représente visuellement la structure du code. Cet arbre montre non seulement comment les éléments sont reliés mais aussi leur ordre et leur hiérarchie, facilitant ainsi les étapes suivantes du processus de compilation.

À la fin de cette étape, nous obtenons structure arborescente représentant la hiérarchie du code source. Cette étape du processus de compilation est vital car elle permet de s'assurer que le programme respecte la grammaire du langage MiniC

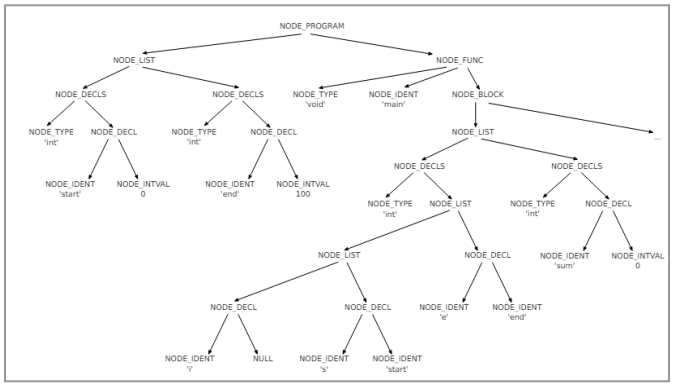

Figure 3: Programme MiniC après l'analyse syntaxique

Vérifications sémantiques page suivante

Vérifications sémantiques

La première passe, la passe de vérifications contextuelles (sémantique) intervient immédiatement après l'analyse syntaxique afin de détecter les erreurs qui ne sont pas de nature syntaxique mais qui concernent le contexte d'utilisation des variables et des types ainsi que de compléter l'arbre existant pour faciliter l'étape suivante, la génération de code.

Lors de cette passe, le compilateur vérifie et prépare plusieurs aspects, notamment:

- Les variables ne peuvent être déclarées qu'en temps que *boolean* ou *integer*. Chaque variable utilisée doit avoir été déclarée préalablement, cette vérification empêche l'accès à des segments mémoires non autorisés. Les variables globales ne peuvent pas être déclarées avec des expressions.
- La visibilité des variables est contrôlée en fonction de leur contexte de déclaration. Les variables locales dans un contexte ne sont pas accessibles en dehors de ce dernier, hors sous-contextes.
- Les opérations effectuées entre variables doivent être compatibles avec leurs types. Par exemple, une addition n'est possible qu'entre des *integer* ou des expressions de type compatible.
- Les expressions de condition d'instructions, telles que présentes dans *if* ou *for*, doivent être de type *boolean*.
- Les différents champs des nœuds de l'arbre sont remplis. Par exemple, le champ *offset* des variables, qui correspond à l'emplacement en mémoire, ainsi que le champ *offset* de la fonction *main()*, qui correspond à la taille initiale à allouer pour la pile, sont initialisés ici. Les fonctions *env_add_element()* pour les variables et *add_string()* pour les chaînes de caractères permettent d'obtenir cette valeur. Aussi, le champ *decl_node*, qui sert à relier chaque utilisation de variables à sa déclaration, est rempli lors de cette passe.

Le code est divisé en plusieurs fonctions, chacune d'entre elles ayant une utilité propre. Chaque fonction prend le nœud en argument et gère toutes les erreurs éventuelles qui pourraient se produire lors d'une analyse sémantique (mauvais typage, informations manquantes…).

Le détail des fonctions, leur utilité et leur implémentation est donné dans les pages suivantes.

const char* get_symbol(node_nature node)

Cette fonction sert à récupérer le symbole associé à la nature d'un nœud donné, elle est seulement utilisée pour l'affichage des erreurs.

void analyse_passe_1(node_t root)

Cette fonction lance l'analyse sémantique en commençant par la racine de l'arbre. Elle initialise le contexte global, commence par analyser les déclarations initiales, puis passe à l'analyse du nœud principal, la fonction *main()*, avant de refermer le contexte global. Si la structure de l'arbre a un problème, comme l'absence de la fonction main, elle arrête l'exécution et signale une erreur.

void analyse_base(node_t node)

Cette fonction récursive est la fonction principale de notre analyse sémantique. Elle parcourt l'arbre pour appliquer les vérifications sémantiques spécifiques à chaque type de nœud. Selon la nature du nœud, elle applique différentes instructions. Pour les fonctions et les blocs, elle pousse un nouveau contexte, analyse les fils et referme le contexte. Pour les listes, elle s'appelle pour chaque fils de la liste. Pour les autres, elle appelle la fonction correspondante à la nature du nœud (expression ou instruction).

void analyse_instruction(node_t node)

Cette fonction analyse et vérifie la conformité des différentes instructions, telles que les boucles, les conditions et les affectations. Pour les boucles et les instructions conditionnelles, elle s'assure que les conditions sont bien de type *boolean*. Elle s'assure aussi les affectations se font entre variables et expressions de types compatibles. Elle parcourt l'ensemble des fils du nœud avec la fonction appropriée selon le type. En cas d'erreurs, comme un mauvais typage, elle retourne une erreur en utilisant notamment la fonction *get_symbol()*.

void analyse_expression(node_t node)

Cette fonction analyse les expressions pour s'assurer que les types des opérandes sont corrects et que les opérations sont effectuées entre des types compatibles et retournent le bon type. Elle traite les diverses opérations arithmétiques, logiques et de comparaison, et s'assure que les expressions respectent les règles de typage du langage. Par exemple, elle vérifie que les opérateurs booléens ne sont utilisés qu'avec des expressions booléennes. Pour les identificateurs *ident*, elle donne aussi l'adresse de déclaration du nœud, *decl_node*. Pour les chaînes de caractères *str*, elle donne l'*offset*.

void analyse_declaration(bool global, node_t node, node_type type)

Cette fonction traite les déclarations de variables en vérifiant les types et en enregistrant les variables dans le contexte approprié (global ou local). Elle s'assure que les types de déclaration correspondent aux valeurs assignées et que les variables globales sont initialisées avec des valeurs constantes. En outre, elle assigne la valeur de l'offset aux variables locales et initialise les variables globales non initialisées. Elle parcourt également les listes et garde l'information du type donné par le fils gauche de *NODE_DECLS* ainsi que l'information sur si la variable est globale ou non.

Génération du code assembleur page suivante

Génération du code assembleur

La deuxième passe du compilateur, la passe de génération de code, est la phase où le compilateur transforme l'arbre produit par l'analyse syntaxique et les vérifications sémantiques, en code assembleur exécutable. Cette étape suit immédiatement les vérifications sémantiques et utilise les informations qui ont été déterminées durant cette passe. Elle utilise par exemple l'offset pour déterminer la position dans la section *.data* ou dans la pile des variables.

Lors de cette passe, le compilateur s'occupe de traduire l'arbre en code assembleur:

- Toutes les variables globales et les chaînes de caractères utilisées dans le programme sont déclarées dans la section de données.
- Le code pour les fonctions, les boucles, les conditions et les expressions est généré et placé dans la section de texte du programme assembleur.

Le détail des fonctions est donné ci-dessous.

void gen_code_passe_2(node_t root)

Cette fonction initie la seconde passe de génération de code pour un programme. Elle commence par créer des instructions pour la section de données, puis génère le code pour les déclarations globales et les chaînes utilisées dans le programme. Ensuite, elle crée la section de texte où le code pour les fonctions est généré. La fonction se termine en s'assurant que la pile est correctement nettoyée et en appelant l'instruction système pour terminer le programme.

void gen_code_base(node_t node)

Cette fonction récursive est la fonction principale de la génération de code. Elle parcourt l'arbre et pour chaque nœud, elle applique les instructions appropriées. Pour les fonctions, elle crée une étiquette et alloue de l'espace pour les variables locales. Pour le reste, elle est très similaire à la fonction *analyse_base()* de la première passe.

void gen_code_print(node_t node)

Cette fonction génère le code MIPS pour l'affichage de variables et de chaînes de caractères. Elle gère les identifiants et les valeurs littérales en générant les instructions nécessaires pour charger les valeurs et appeler le *syscall* approprié.

void gen_code_instruction(node_t node)

Cette fonction génère le code pour les différentes instructions. Les boucles et les instructions conditionnelles sont converties en branchement avec des étiquettes. Les affectations dépendent du nombre de registres disponibles et de si la variable à laquelle on affecte une expression est globale ou non. À chaque fois, on parcourt les expressions présentes pour générer leurs codes.

void gen_code_expression(node_t node)

Cette fonction s'occupe de traduire les expressions arithmétiques et logiques en séquences d'instructions MIPS. Elle s'occupe de charger les identificateurs ident dans le registre approprié. Elle prend également en compte les types d'opérations comme les additions, soustractions, et les opérations booléennes, pour produire les instructions MIPS correspondantes. Enfin, elle charge les *boolean* et *integer* dans des registres en s'assurant qu'il n'y a aucun dépassement de la fonction MIPS *ori*, qui est sur 16 bits. Durant toutes ces étapes, elle s'occupe également de vérifier si des registres sont disponibles et, le cas échéant, libère de la place en plaçant la valeur du registre dans la pile temporaire.

void gen_code_declaration(node_t node)

Cette fonction traite les déclarations en générant le code pour initialiser les variables dans la section de données ou au début de la fonction pour les variables locales. Les déclarations globales sont directement mises dans la section *.data* avec des valeurs initiales tandis que les locales sont gérées au niveau de la pile. Encore une fois, elle fonctionne de manière similaire à *analyse_declaration()*.

Arguments page suivante

Implémentations annexes

Arguments

Comme demandé dans le polycopié, le compilateur minicc prend en compte l'ajout de plusieurs arguments. Pour les implémenter, nous avons simplement utilisé une boucle while qui prend fin lorsqu'il n'y a plus d'arguments. On compare les différents arguments aux arguments attendus à l'aide de la fonction **strcmp()** de la librairie **string.h** et on effectue les changements nécessaires selon l'argument.

Le compilateur s'utilise donc de la manière suivante:

./minicc [options] <source_file_to_compile>

Avec la liste des options suivantes:

- ***-b*** : Affiche le banner du compilateur et est incompatible avec les autres options
- ***-s*** : Arrête la compilation après l'analyse syntaxique
- ***-v*** : Arrête la compilation après la passe de vérifications
- ***-h*** : Affiche l'aide
- ***-o filename*** : Définit le nom du fichier de sortie
- ***-t level*** : Définit le niveau de trace entre 0 et 5
- ***-r count*** : Définit le nombre maximum de registres à utiliser entre 4 et 8

Les différentes règles relatives à l'utilisation de ces options sont disponibles avec la commande *-h*.

Tests page suivante

Tests

Comme précisé dans le polycopié, les tests sont organisés dans les différents dossiers du répertoire *Tests/* entre les différentes étapes de la compilation. Les tests sont aussi divisés en deux sous-dossiers *OK/* et *KO/* à chaque étape.

Concernant les tests OK, les tests qui ne retournent aucune erreur, nous avons choisi de faire les mêmes tests pour les 3 différentes parties, les tests numérotés de 0 à 15, ainsi que 5 tests exclusifs à chaque partie. Par exemple, les tests OK numérotés de 16 à 20 de l'analyse syntaxique ne compilent plus lors de la passe de vérification.
Pour les tests KO, les tests sont tous exclusifs à chaque partie et testent des spécificités introduites par leur partie respective. Par exemple, les tests KO effectués lors de la passe de vérification syntaxique ne seront pas réutilisés lors l'analyse de la génération de code.

Pour la passe de génération de code, les tests sont différents puisqu'ils ne retournent aucune erreur à la compilation. En effet, les tests KO et cette partie sont censés provoquer une erreur ou avoir un comportement inattendu à l'exécution du code (*Runtime error*).

Pour l'automatisation des tests, nous avons réalisé un script python qui s'occupe de compiler tous les fichiers .c présents dans le répertoire Tests, sous-dossiers inclus. Le parcours du répertoire est rendu possible grâce à la librairie **os** et l'exécution des commandes avec enregistrement des erreurs est rendu possible grâce à la librairie **subprocess**.
Pour la passe 2, la génération du code, une fonction spécifique est utilisée et compare le résultat attendu de chaque fichier de test à la sortie de l'exécution du fichier assembleur correspondant. La librairie python utilisée est **difflib** et la commande pour l'exécution du fichier assembleur est *java -jar Tests/mars_4_2.jar nc np ae1 file_name.s*.
Dans chaque fichier .c du répertoire *Gencode/*, un commentaire est placé au début pour indiquer quelle doit être le résultat attendu. On compare donc ces valeurs avec celles de la sortie du fichier assembleur.
Dans chaque fichier .c des répertoires *KO/* (sauf de *Gencode/*), le commentaire indique la ligne où l'erreur doit se produire. On compare cette valeur à celle de la ligne de l'erreur retournée.

Le script affiche un court résumé à la fin qui donne des informations sur le nombre total de tests, le nombre de tests réussies, le nombre de tests OK et KO et le nombre de tests qui renvoient une erreur dans chaque sous-partie.

Le script accepte des arguments qui lui sont propres ainsi que d'autres qui seront utilisés comme arguments pour minicc:

- ***--s*** : N'exécute pas les tests de syntaxe
- ***--v*** : N'exécute pas les tests de vérification syntaxique (passe 1)
- ***--g*** : N'exécute pas les tests de génération de code (passe 2)
- ***--e*** : N'affiche pas les détails des erreurs, seulement le mot *'ERROR'*
- ***--ko*** : N'exécute pas les tests dans les dossiers *KO/*
- ***--ok*** : N'exécute pas les tests dans les dossiers *OK/*
- ***--r*** : N'affiche pas le résumé des tests à la fin
- ***--help*** : Affiche l'aide du script
- ***--h*** : Affiche l'aide de minicc
- ***--b*** : Affiche la bannière de minicc

Pour le parsing des arguments, le script utilise la librairie **sys** de python. Tous les autres arguments seront passés à minicc, peu importe qu'ils soient corrects ou non.

Lorsqu'un test passe sans problème, c'est-à-dire que le test ne retourne pas d'erreurs lorsqu'il est situé dans *OK/* ou que le test retourne une erreur à la bonne ligne si il est situé dans *KO/*, le script affiche *'SUCCESS'*. Si le test était un test KO, les informations de l'erreur sont aussi affichées. Si l'option *--e* est sélectionnée, le script n'affiche pas les détails de l'erreur.
Lorsqu'un test ne passe pas, c'est-à-dire qu'il a un comportement inattendu (mauvaise ligne, mauvaise sortie à l'exécution…), le script affiche *'ERROR'*.

Erreurs connues page suivante

Erreurs connues

À notre connaissance, notre compilateur fonctionne sans problèmes. Le code assembleur produit semble produire de bons résultats.

Cependant, nous avons relevé un problème de *'Invalid read'* dans le compilateur de référence *minicc_ref*. En effet, en effectuant la commande *valgrind ./minicc_ref Tests/verif/OK/test0.c*, le terminal affiche *'6 errors from 2 context'*.

Conclusion

Ce projet de conception d'un compilateur pour le langage MiniC a été une expérience enrichissante et instructive, illustrant l'application directe des concepts de compilation que nous avons étudiés en cours. L'emploi d'outils comme Lex et Yacc a grandement simplifié les processus d'analyse lexicale et syntaxique, permettant de convertir le code source MiniC en structures de données.

Les étapes de vérifications sémantiques ont permis d'assurer l'intégrité du contexte d'utilisation des variables et des types, minimisant les erreurs à l'exécution.

La dernière étape, la génération de code assembleur MIPS s'est appuyé sur les résultats des analyses précédentes pour produire un code exécutable. Ce projet a non seulement consolidé nos compétences en programmation mais a aussi approfondi notre compréhension des étapes critiques du processus de compilation, soulignant l'importance de chaque phase pour aboutir à un produit final fonctionnel et optimisé.