Compilation

Rapport de projet

2019 - 2020

I. Objectif

L'objectif de ce projet est de réaliser un compilateur pour le langage MiniC. Le langage MiniC est, comme son nom peut le laisser penser, une version "réduite" du C. Ainsi, quelques limitations sont à noter par rapport au langage C:

- les variables et les expressions ne peuvent être que de type int ou bool,
- la conversion est impossible (int vers bool ou inversement), par conséquent il n'y a pas de cast,
- il n'y a pas de fonction excepté le *main*,
- il n'est pas question de pointeurs, ni de tableaux,
- certains opérateurs ne peuvent pas être supportés tels que ++, --, -=, +=, *=, /=, >>=, <<=, &=, |= (énumération non exhaustive),
- il en va de même pour certains mots-clés et fonctionnalités bien connues pour les utilisateurs du langage C comme *switch*, *case*, *break*, *continue*, *goto*, *typedef*, *struct*, *union*, *volatile*, *register*, *packed*, inline, static, *extern*, *unsigned*, *signed*, *long*, *long long*, *short*, *char*, *size_t*, *float*, *double*.

Ce compilateur devra remplir la fonction de retourner en sortie un programme assembleur dans le langage MIPS à partir d'un programme MiniC en suivant différentes étapes à savoir l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique (il s'agit de l'étape de vérifications

contextuelles) et enfin la génération du code dans le langage assembleur cible.

II. Analyse lexicale

L'analyse lexicale constitue la première étape pour la réalisation de notre compilateur. Il s'agit de la transformation d'une suite de caractères d'un fichier source en une suite de lexèmes. Par exemple, la suite de lettres while, quand elle est entourée de caractères autres que des chiffres, des lettres, et du caractère _, correspond à un mot réservé du langage : un lexème va représenter le while qui lui est alors associé. Fort heureusement pour nous, nous n'avons pas eu à écrire la machine à état responsable de cette transformation. Afin de faciliter la description des lexèmes, nous nous sommes servi de l'outil Lex.

Pour ce projet, nous avons utilisé les lexèmes représentatifs du MiniC suivants :

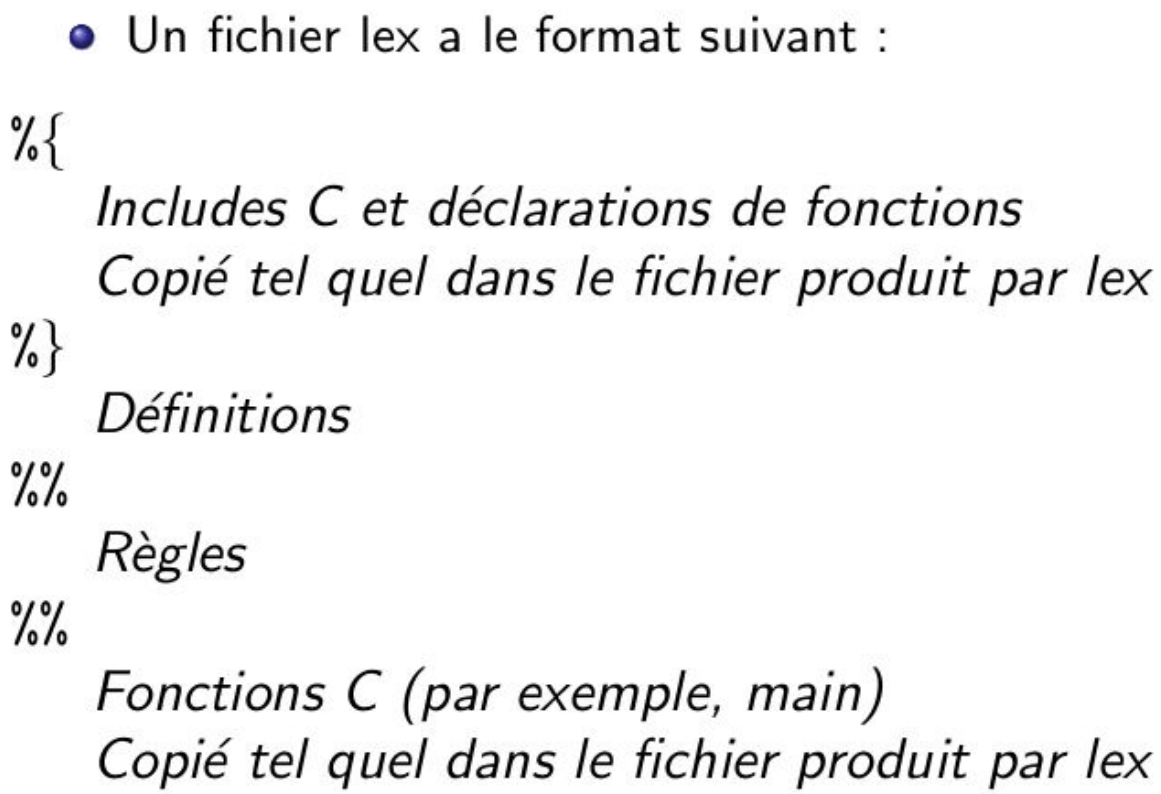
```
"void"      return TOK_VOID;
"int"       return TOK_INT;
"bool"      return TOK_BOOL;
"true"      return TOK_TRUE;
"false"     return TOK_FALSE;
"if"        return TOK_IF;
"else"      return TOK_ELSE;
"while"     return TOK_WHILE;
"do"        return TOK_DO;
"for"       return TOK_FOR;
"print"     return TOK_PRINT;
"+"         return TOK_PLUS;
"-"         return TOK_MINUS;
"/"         return TOK_DIV;
"*"         return TOK_MUL;
"%"         return TOK_MOD;
"="         return TOK_AFFECT;
"=="        return TOK_EQ;
"!="        return TOK_NE;
"<"         return TOK_LT;
"<="        return TOK_LE;
"<<"        return TOK_SLL;
">"         return TOK_GT;
">="        return TOK_GE;
">>"        return TOK_SRA;
">>>"       return TOK_SRL;
"!"         return TOK_NOT;
"~"         return TOK_BNOT;
"&"         return TOK_BAND;
"&&"        return TOK_AND;
"|"         return TOK_BOR;
"||"        return TOK_OR;
"^"         return TOK_BXOR;
";"         return TOK_SEMICOL;
","         return TOK_COMMA;
"("         return TOK_LPAR;
")"         return TOK_RPAR;
"{"         return TOK_LACC;
"}"         return TOK_RACC;
```

Extrait du fichier lexico.l.

Le fichier *lexico.l* est le fichier qui va utiliser l'outil Lex, précédemment introduit. Nous pouvons partager en quatre parties sa description. Il est tout d'abord constitué de directives de préprocesseur et de déclarations de fonction qui vont être copiées tels quels dans le fichier produit par l'outil lex. Cette première partie est encadrée par les séquences de caractères "%{" et "%}". Ensuite vient la partie contenant les définitions des lexèmes. Cette partie doit précéder la partie décrivant les règles lexicales. On va ainsi retrouver dans ces lexèmes, les types (void, int et bool), les mots réservés conservés du C (de *true* jusqu'à

print) ainsi que tous les symboles nécessaires pour définir la structure d'un programme ou les opérations sur la mémoire. Tous sont issus de la lexicographie du MiniC donnée dans les spécifications du projet. Nous avons conclu l'élaboration de ce fichier avec la fonction main qui sera copiée dans le fichier produit par lex.

Le fichier lex.yy.c est le fichier produit par lex à partir du fichier *lexico.l*. Voici une image extrait du cours de M. Meunier récapitulant la description du fichier *lexico.l*.



Description du fichiers lexico.l extraite du cours de M. Meunier

Lorsque le compilateur reçoit en entrée un programme écrit en MiniC, il commence par le découper en lexèmes en remplaçant caractère après caractère ou mot après mot chaque ligne du programme par un lexème correspondant. Le programme est alors semblable à une succession de lexèmes. C’est alors que survient l’analyse syntaxique.

III. Analyse syntaxique

Le but de l'analyse syntaxique est de vérifier que la succession de lexèmes produits à l'étape précédente est cohérente. Pour faire cette analyse de manière efficace, on décrit les associations valides avec l'aide de règles de grammaire (généralement ces règles sont de type hors-contexte). C'est normalement au cours de cette analyse qu'est construit l'arbre du programme. En effet, à chaque fois, qu'une règle valide de grammaire est reconnue (une suite de tokens qui a du sens), la partie correspondante de l'arbre du programme est construite.

Tout comme pour l'analyse lexicale, nous utilisons ici un outil simple avec lequel nous avons simplement écrit les règles de grammaire dans un fichier. L'outil utilisé pour ce projet s'appelle Yacc et le fichier où sont écrites toutes les règles de grammaire est grammar.y. En continuant l'analogie avec l'analyse lexicale, l'outil Yacc se sert du fichier où sont décrites les règles de grammaire pour générer les fichiers y.tab.c et y.tab.h. Enfin, les analyses lexicales et syntaxiques sont effectuées conjointement. L'image ci-dessous issue du cours de M. Meunier résume parfaitement le processus de compilation du compilateur.

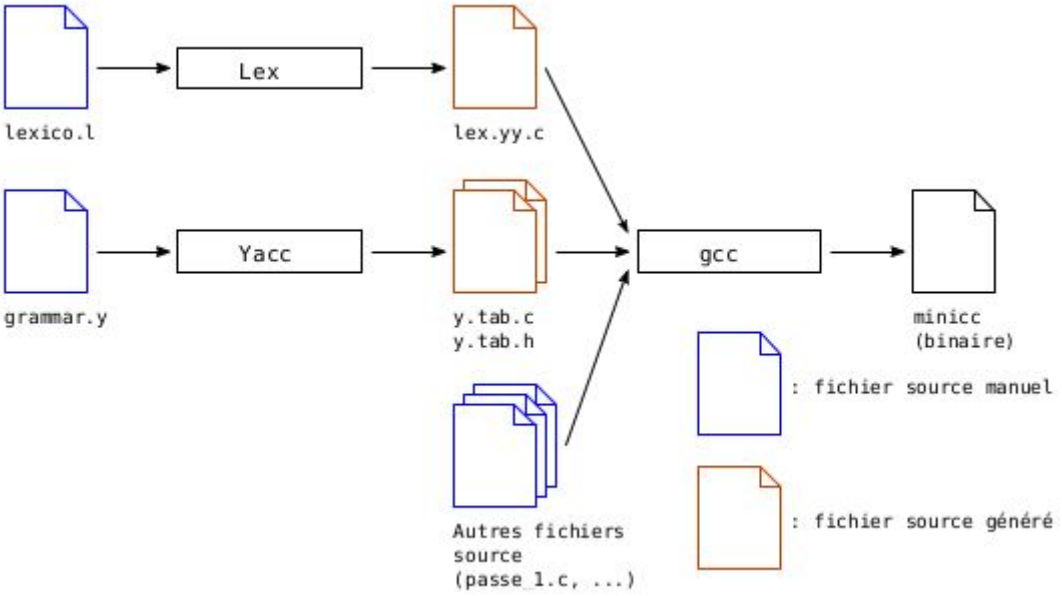
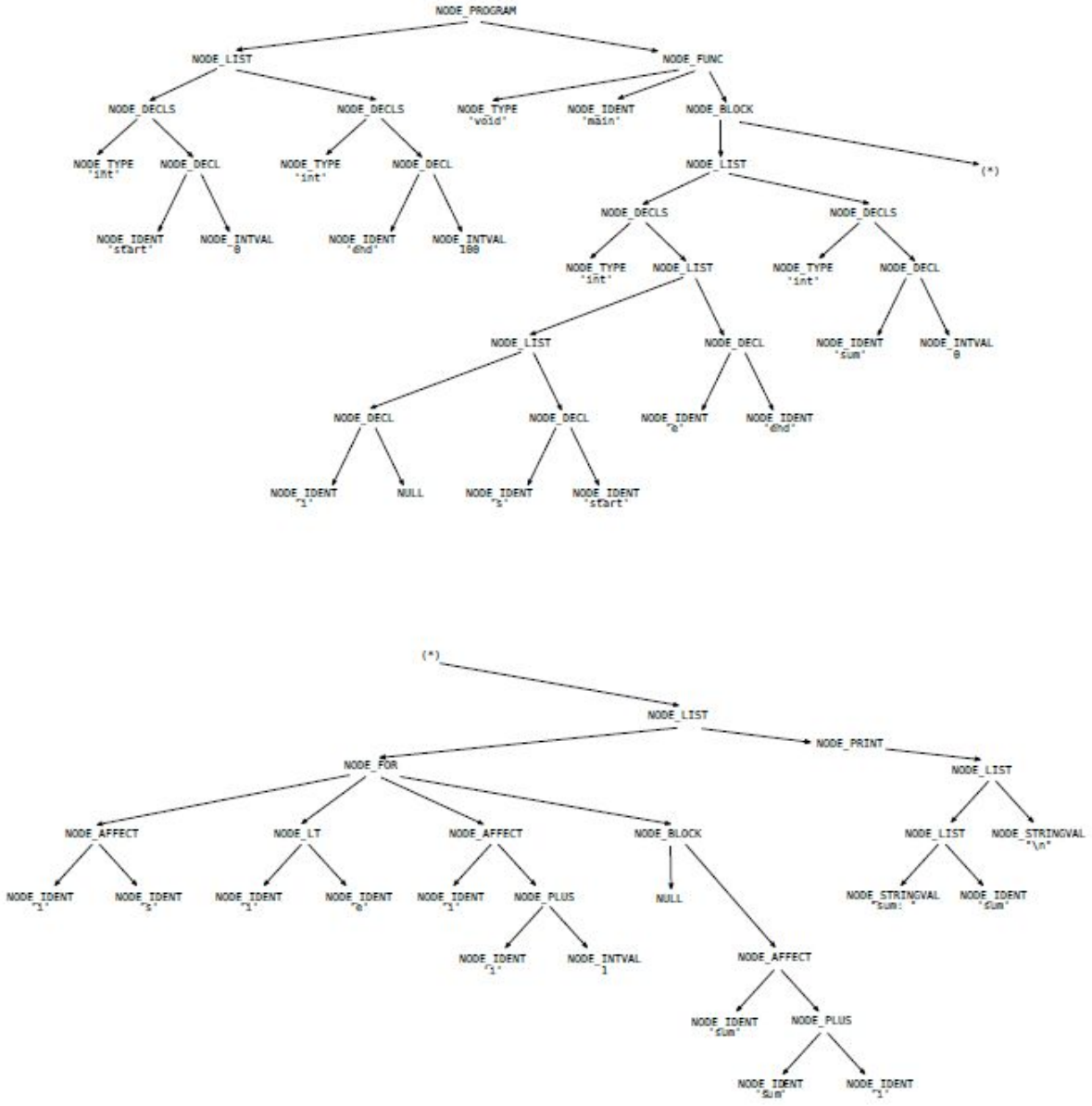


Image extraite du cours de M. Meunier

Le fichier grammar.y comprend les règles utilisées dans le langage MiniC. Ce fichier reprend les lexèmes définis dans le fichier *lexico.l* et indique les associations possibles entre ces derniers. De plus, grammar.y contient les fonctions nécessaires à la création de l'arbre du programme. Lorsqu'un lexème est repéré, un noeud est créé. Ces noeuds sont les éléments constitutifs de l'arbre du programme. Ils sont associés en suivant les règles de grammaire du MiniC définies hors-contexte et données dans les spécifications du sujet.



Exemple d'arbre créé après une analyse syntaxique

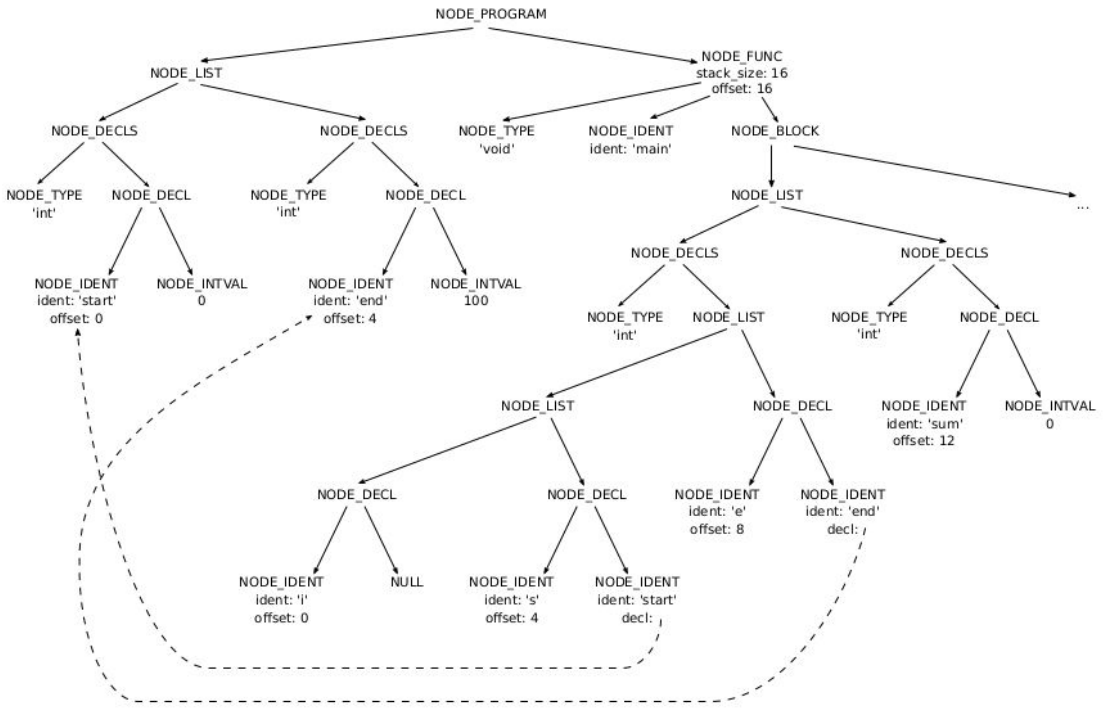
- Un fichier yacc a le format suivant :

```
%{
    Includes C et déclarations de fonctions
    Copié tel quel dans le fichier produit par lex
%}
    Définitions
%%
    Règles
%%
    Fonctions C
    Copié tel quel dans le fichier produit par yacc
```

Description du format de grammar.y extraite du cours de M. Meunier

IV. Analyse sémantique

Cependant, ajouter des mots les uns après les autres en suivant une grammaire ne suffit pas à donner du sens à un texte. En plus de l'analyse syntaxique, l'analyse sémantique s'assure que l'enchaînement des lexèmes est cohérent dans le contexte du programme. C'est à cette étape que notre compilateur vérifie les initialisations des variables, la fin du programme, l'ordre d'évaluation des opérations, la taille des différentes variables pour éviter les débordements de mémoires ainsi que les procédures d'affichage. Vous l'aurez bien compris, l'analyse sémantique correspond à la vérification de la concordance de l'arbre.



Exemple d'arbre créé après une analyse sémantique

V. Génération du code assembleur

Nous arrivons à la dernière étape de notre compilateur. Après l'analyse sémantique (vérification contextuelle), il s'agit maintenant de traduire le code source en langage assembleur (MIPS). Pour cela, nous nous sommes permis d'utiliser l'arbre précédent et de générer des instructions MIPS pour chaque noeud de l'arbre. Dans le fichier de sortie, par défaut *out.s*, nous obtenons le résultat de la compilation. Nous nous sommes familiarisés avec les notions de gestion de pile mais aussi d'allocation de registre. Pour cette dernière nous avons pris la liberté de créer une fonction get_load qui permet le chargement d'un registre. Cela était nécessaire notamment pour les différents calculs arithmétiques comme l'addition, la soustraction, la multiplication ou même la division.

VI. Déroulement du projet

Pour mener ce projet à terme nous avons d'abord suivi un plan sur plusieurs semaines. Les bases du langage C ainsi que certaines bases d'algorithmie élémentaire se doivent d'être parfaitement maîtrisée afin de réaliser ce compilateur. Ainsi, nous avons d'abord cherché à comprendre la fonction des fichiers et comment les utiliser. Il n'est sans nul doute besoin de rappeler qu'apprendre à les utiliser constitue le fondement de la réalisation de notre projet. Ainsi, nous nous sommes concentrés sur quelques exemples pratiques pour ensuite générer un lex et un yacc qui fonctionne avec les instructions indiquées dans le sujet. Nous avons pu tester ces outils avec un exemple de code assez complet qui a été fourni avec le cours. Par la suite, nous avons utilisé dump_tree pour vérifier le fonctionnement de lex et de yacc.

Pour l'étape suivante, nous avons créé un fichier passe.c qui avait pour but d'effectuer l'analyse sémantique qui est l'analyse contextuelle du code source (cf. partie IV de ce rapport pour plus d'explication). À travers cette première passe, nous avons cherché à obtenir un arbre similaire à celui associé au code dans le cours. Nous avons continué sur le même fichier la seconde passe et dernière étape de la compilation à savoir la génération du code assembleur. Le code obtenu est proche sans être exactement semblable à celui fourni dans le cours mais fonctionne néanmoins. En parallèle, nous avons écrit différents tests ainsi qu'un script python. Nous avons par ailleurs achevé l'écriture de la section décrivant les lignes de commandes qui doivent être supportées par minicc.

VII. Détails sur common.c

Le fichier common.c permet de définir les options supportées par le compilateur minicc. Les options disponibles sont les suivantes :

- -b : Affiche une bannière MiniC en ascii art ainsi que nos noms,
- -o <Nom_de_fichier> : Change le nom du fichier de sortie (par défaut out.s),
- -t <int> : Modifie le niveau de trace de 0 à 5 (0 pour aucunes traces, 5 pour toutes les traces, par défaut 0),
- -r <int> : Change le nombre maximum de registres utilisables entre 4 et 8 (par défaut 8),
- -s : Arrête la compilation à l'étape de l'analyse syntaxique,
- -v : Arrête la compilation à l'étape de la passe de vérification,
- -h : Renvoie la liste des options présentées ci-dessus et leur fonction.

Common.c contient également le format des messages d'erreurs et ce quelle que soit leur nature.

VIII. Fonctionnement de passe.c

Pour le passe.c, nous avons choisi d'implanter l'analyse sémantique ainsi que la génération de code en un seul et même fichier. Cela se traduit dans le code par la présence de deux passes : passe_uno() qui correspond à l'analyse sémantique et passe_dos() pour la génération de code.

Pour l'analyse sémantique, passe_uno() est composé d'un switch qui va venir vérifier la bonne utilisation du lexème. Cela signifie que si un lexème est utilisé dans le mauvais contexte, un message d'erreur sera retourné.

La génération du code se fait dans passe_dos(). De manière analogue au fonctionnant de passe_uno(), un switch vient créer le code assembleur à partir des noeuds de l'arbre du programme (et donc des lexèmes correspondants). La création du code s'effectue avec les fonctions fournies par la bibliothèque de création des programmes MIPS. Ainsi, chaque noeud est traduit en MIPS par une série de commandes. Ci-après, les fonctions permettant de créer du code en MIPS :

- `void create_data_sec_inst();`
- `void create_text_sec_inst();`
- `void create_word_inst(char * label, int32_t init_value);`
- `void create_asciiz_inst(char * label_str, char * str);`
- `void create_label_inst(int32_t label);`
- `void create_label_str_inst(char * label);`
- `void create_comment_inst(char * comment);`
- `void create_lui_inst(int32_t r_dest, int32_t imm);`
- `void create_addu_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_subu_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_slt_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_sltu_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_and_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_or_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_xor_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_nor_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_mult_inst(int32_t r_src_1, int32_t r_src_2);`
- `void create_div_inst(int32_t r_src_1, int32_t r_src_2);`
- `void create_sllv_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_srlv_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_srav_inst(int32_t r_dest, int32_t r_src_1, int32_t r_src_2);`
- `void create_addiu_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_andi_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_ori_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_xori_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_slti_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_sltiu_inst(int32_t r_dest, int32_t r_src_1, int32_t imm);`
- `void create_lw_inst(int32_t r_dest, int32_t imm, int32_t r_src_1);`
- `void create_sw_inst(int32_t r_src_1, int32_t imm, int32_t r_src_2);`
- `void create_beq_inst(int32_t r_src_1, int32_t r_src_2, int32_t label);`
- `void create_bne_inst(int32_t r_src_1, int32_t r_src_2, int32_t label);`
- `void create_mflo_inst(int32_t r_dest);`
- `void create_mfhi_inst(int32_t r_dest);`
- `void create_j_inst(int32_t label);`

- `void create_teq_inst(int32_t r_src_1, int32_t r_src_2);`
- `void create_syscall_inst();`

Il y a de plus 3 fonctions :

- `void create_program()` : à appeler au début pour créer un programme
- `void free_program()` : à appeler à la fin pour libérer les structures allouées à la création
- `void dump_mips_program(char * filename)` : pour écrire le programme au format texte dans le fichier `filename`

Cependant, pour les fonctions arithmétiques telles que les additions ou les soustractions, nous avons développé une fonction get_load() qui permet d'accéder aux registres de manières à éviter trop de redondance dans le code.

IX. Script de test

Au cours du développement de notre compilateur, il nous fallait tester son fonctionnement. Pour ce faire, nous avons élaboré une série de programmes de tests qui permettrait de vérifier le bon déroulement de l'analyse syntaxique, de la passe de vérification ainsi que la génération du code.

Chaque code de test est stocké dans un dossier en fonction du résultat attendu lors de sa compilation. Dans le dossier KO, on retrouve tous les codes MiniC supposés échouer tandis que dans le dossier OK se trouvent tous les fichiers dont la compilation ne pose pas problème. Il y a trois paires de dossiers KO/OK pour les trois phases du compilateur : analyse syntaxique (dossier Syntaxe), passe de vérification (Verif) et génération du code (Gencode).

Pour accélérer les tests nous avons mis au point trois scripts : un pour chaque phase de la compilation. Ces scripts utilisent le compilateur avec certaines options spécifiques sur chaque fichier test. Ainsi pour vérifier si le fonctionnement de l'analyse syntaxique, on utilise minicc avec l'option -s qui arrête la compilation à cette étape. Pour la passe de vérification, l'option utilisée est -v et on laisse le fonctionnement par défaut du compilateur pour la génération du code.

En ce qui concerne les fichiers de tests, ceux pensés pour tester l'analyse syntaxique ont pour but de vérifier que les lexèmes sont bien repérés par le compilateur et le cas échéants qu'ils respectent la syntaxe du MiniC. Les codes dits KO sont donc voués à l'échec. À noter que certains fichiers test passent l'analyse syntaxique sans pour autant

passer la passe de vérification. Ces fichiers sont donc dans le dossier OK de Syntaxe mais dans les dossiers KO de Verif et Gencode.

La mise au point des fichiers de test est semblable pour la passe de vérification et la génération du code suit la même logique : des fichiers conformes pouvant passer à l'étape suivante (OK) ou des fichiers défaillants générant des messages d'erreur (KO). Idem, des fichiers conformes lors de la passe de vérification peuvent être refusés au moment de générer le code.

Tous ces fichiers de test ont été eux-mêmes testés grâce au compilateur MiniC de référence (minicc_ref) fournis et développé par M. Meunier.

X. Critique sur notre travail

Bien que l'ensemble des analyses se soit généralement déroulé de manière optimale, nous avons décelé quelques points négatifs que nous n'avons pas su corriger.

Lors de la déclaration de variable, notre programme construit bien l'arbre avec les noeuds adéquats, mais indique que la variable qui vient d'être créée est déjà déclarée ce qui est une erreur étant donnée que nous venons juste de la déclarer. Nous pensons que le message survient lors de la remontée du noeud d'identité NODE_IDENT vers le noeud de déclaration de la variable en question NODE_DECL.

Le second souci est la génération du code MIPS à l'infini dans notre fichier de sortie. Ceci correspond probablement à un mauvais parcours de notre arbre menant à une faute de segmentation. En effet, cette cause sera la source du premier problème explicité plus haut. Néanmoins cela ne dégrade en aucun cas la qualité de notre code assembleur.

XI. Conclusion

A travers ce projet, nous avons pu accéder concrètement au concept de la compilation, de l'analyse lexicale jusqu'au code assembleur. Malgré les conditions difficiles de réalisation de ce projet, nous avons su faire preuve d'esprit d'équipe afin de produire un compilateur satisfaisant.

Remerciements

à M. Meunier, encadrant technique de notre projet de compilation. En plus de sa disponibilité et de son implication dans le suivi de notre projet, il nous a permis d'approfondir nos connaissances sur le sujet des étapes de la compilation.