

Projet Compilation

EISE4

Yining BAO - Qianhui JIN

Encadrant: M.Meunier, M.Hilaire

Année: 2020 - 2021

Sommaire

I. Introduction	3
II. Analyse lexicale (lexico.l)	4
III. Analyse syntaxique (grammar.y)	5
IV. Analyse sémantique (passe 1)	8
V. Génération du code (passe 2)	10
VI. Ligne de commande	11
VII. Messages d'erreur	12
VIII. Test	12
1. Test Syntaxe	13
2. Test Verif	13
3. Test Gencode	13
IX. Les Difficultés	14
X. Conclusion	14
XI. Bibliographie	14

I. Introduction

Le **compilation** est un module d'enseignement de notre deuxième année du cycle ingénieurs en EISE (électronique, informatique et système embarqué). Pendant ce module, nous devons réaliser un projet. Le but de ce projet est de réaliser un **compilateur** qui permet de transférer un programme source en programme assembleur sans erreur. Nous devons coder ce compilateur en langage **MiniC** qui est un sous-ensemble du langage C, en utilisant les fichiers sources fournies par **M.Meunier** sous forme suivant:

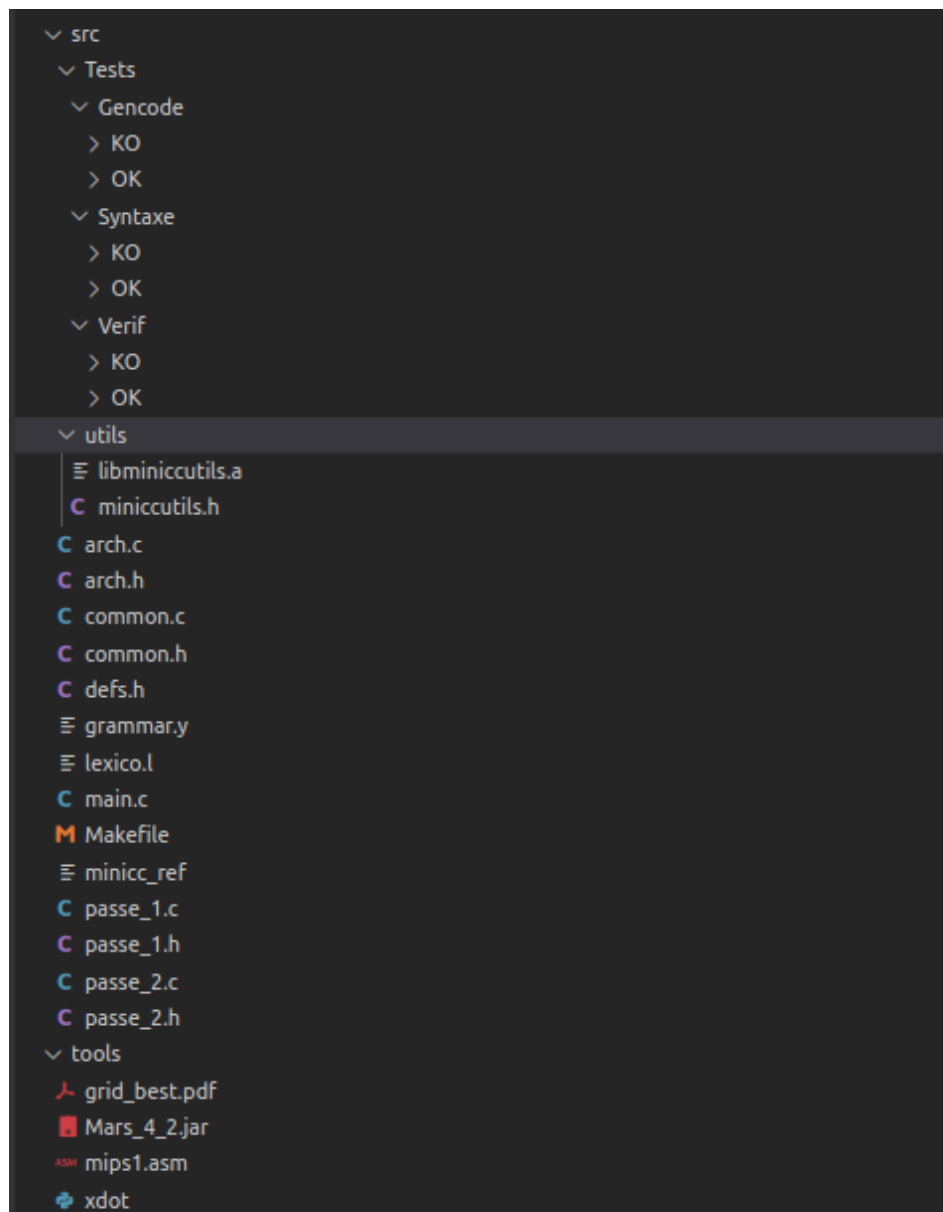


Figure 1: L'architecture du programme compilateur

Pour réaliser le compilateur, dans un premier temps, nous devons remplir le fichier *lexico.l* qui permet de faire l'analyse lexicale. Ensuite, nous devons compléter le fichier *grammar.y* afin de permettre à faire l'analyse syntaxique. Après nous devons réaliser le passe 1 (*passe_1.c*) qui permet de faire une analyse sémantique. A la fin, nous devons faire la génération du code (*passe_2.c*). En même temps, nous devons faire les tests pour vérifier que le programme a bien été réalisé qui est stocké dans le fichier *Tests*. Pour compiler, nous devons utiliser les fichiers *arch.c*, *defs.c*, *minicutils.c* et *common.c*. Dans le fichier *common.c*, nous devons ajouter deux fonctions, un est pour vérifier si la commande tapée dans le terminal est correcte, un est pour désallouer les allocations.

II. Analyse lexicale (lexico.l)

L'analyse lexicale sert à transformer une suite de caractère en une suite de tokens, en utilisant **Lex**.

Pour réaliser cette analyse, tout d'abord, nous définissons les différents langages qui est écrire en majuscule comme le figure suivant:

```

LETTRE      [a-zA-Z]
LETTRE_HEX  [a-fA-F]
CHIFFRE     [0-9]
CHIFFRE_NON_NUL [1-9]
IDF         {LETTRE}({LETTRE}|{CHIFFRE}|" ")*
ENTIER_DEC  "0"|{CHIFFRE_NON_NUL}({CHIFFRE})*
ENTIER_HEX  "0x"({CHIFFRE}|{LETTRE_HEX})+
ENTIER      {ENTIER_DEC}|{ENTIER_HEX}
CHAINE_CAR  [\x20-\x21\x23-\x5b\x5d-\x7e]
CHAINE      \"({CHAINE_CAR}|\"\\\"|\"\\n\")*\"
COMMENTAIRE \"/\".*

```

Figure 2: Les identificateurs

Parmi ces langages, il y a 4 classes différentes. Ce sont l'identificateur, l'entier, la chaîne de caractère et le commentaire. L'identificateur *IDF* est composé par les lettres allant de a à z en minuscule et en majuscule (*LETTRE*) et les chiffres allant de 0 à 9 (*CHIFFRE*). Le littérale entier *ENTIER* est composé par les entiers décimaux (*ENTIER_DEC*) et les entiers hexadécimale (*ENTIER_HEX*). Pour le entier décimal est composé par les chiffres non null (*CHIFFRE_NON_NUL*). Et pour le entier hexadécimale est construit par les chiffres et les lettres hexadécimales (*LETTRE_HEX*). La chaine de caractère *CHAINE* contient l'ensemble des

caractères imprimables (*CHaine_CAR*) et les caractères " et \ . Le commentaire *COMMENTAIRE* est une suite de caractères imprimables et tabulations qui commence par // et s'étend à la fin de la ligne.

Ensuite, nous définissons les tokens suivant:

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

Figure 3: Les tokens

Parmi les tokens, il y a 2 parties différentes, ce sont les mots réservés et les symboles spéciaux. Les mots réservés dans la figure 3 sont allant de *void* jusqu'à *print*. Et les restes sont des symboles spéciaux.

A la fin, nous définissons 3 tokens spéciaux qui associent l'identifiant, l'entier et la chaîne de caractère comme *TOK_IDENT*, *TOK_INTVAL* et *TOK_STRING*.

III. Analyse syntaxique (grammar.y)

L'analyse syntaxique permet de vérifier la suite de tokens est valide et construire l'arbre du programme, en utilisant **Yacc**. Aussi dans cette partie, nous allons définir la grammaire de notre propre langage.

Pour réaliser cette analyse, tout d'abord, nous définissons les tokens en leur donnant leur association et leurs priorités, et leurs types de retour (*tokens*, *nonassoc*, *right*, *left*, *type*).

Ensuite, nous suivons les règles syntaxiques de MiniC pour réaliser la grammaire de notre langage. Dans cette étape, nous choisissons d'abord le type de déclaration à déclarer, après nous définissons la syntaxe qui lui est associée. Dans la figure 4, nous présentons un exemple sur la déclaration de *program* et *listdecl*.

```

program:
    listdeclnonnull maindecl
    {
        $$ = make_node(NODE_PROGRAM, 2, $1, $2);
        *program_root = $$;
    }
    | maindecl
    {
        $$ = make_node(NODE_PROGRAM, 2, NULL, $1);
        *program_root = $$;
    }
    ;

listdecl:
    listdeclnonnull
    { $$ = $1; }
    |
    { $$ = NULL; }
    ;

```

Figure 4: Exemple de déclaration

Dans le *program*, nous définissons 2 règles syntaxiques différentes. Dans ces deux règles, nous utilisons la même fonction **make_node** pour créer la node *NODE_PROGRAM*. La seule différence est sa liste de déclaration, dans le deuxième cas, il n'y a qu'une seule déclaration *maindecl*, donc nous définissons NULL dans la position du *listdeclnonnull*. Et pour associer la liste dans la fonction **make_node**, nous utilisons \$ pour référencer la déclaration selon leur position dans la liste, par exemple, pour la premier cas, *maindecl* est la deuxième dans la liste, donc nous utilisons \$2 pour la référencer, mais dans la deuxième cas, il est le premier dans la liste, donc nous utilisons \$1. En général, chaque déclaration est définie de la même manière, sauf dans le cas comme *listdeclnonnull*. Dans ce cas, nous définissons directement la déclaration en NULL ou à sa liste déclaration.

Puis, pour faire fonctionner la fonction **make_node** afin de créer les nodes dans l'arbre. Nous allons d'abord allouer un variable de type *node*. Ensuite, on lui affecte la nature, la ligne et le nombre de fils pour cette node. S'il existe des fils dans cette node, nous allouons pour ses fils, et on appelle la macro *va_start* pour initialiser la variable *ap* de type *va_list*. Après on appelle la macro *va_arg* pour extraire ses fils. Et à la fin, on libère la liste.

[illegible]

7

IV. Analyse sémantique (passe 1)

L'analyse sémantique sert à vérifier les erreurs d'exécution, même si le programme a réussi à compiler, il y a peut être des erreurs d'exécution qui existent.

Pour faire cette vérification, nous allons suivre la même procédure dans la partie grammaire. Comme nous savons que le programme va commencer par *NODE_PROGRAM*, donc dans la fonction principale **analyse_passe_1**, on n'exécute que la fonction **analyse_program**. Et dans la fonction **analyse_program**, il y a 2 cas possibles comme la grammaire, n'importe dans quelle cas, il va exécuter la fonction **analyse_xxx** associer (figure 7). Ensuite faire la même procédure. Donc en général, le passe 1 est une fonction récursive, il va complètement parcourir l'arbre.

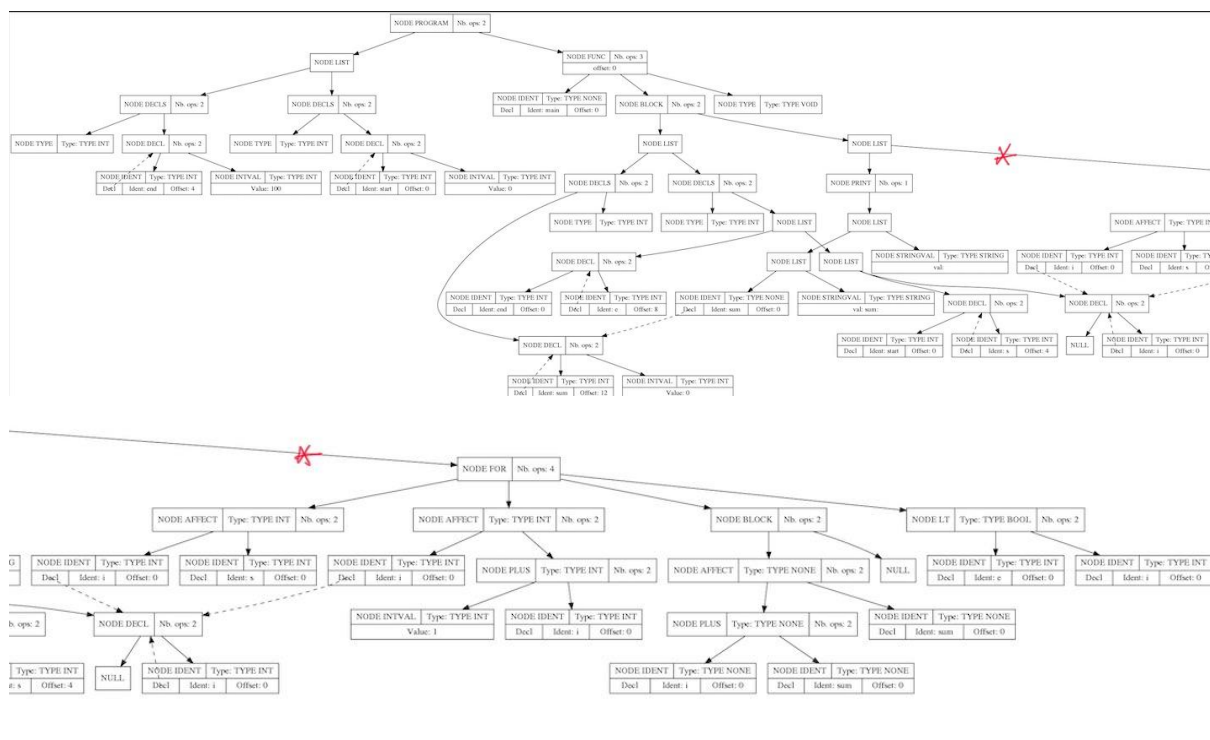
```
void analyse_program(node_t node){
    push_global_context();
    glob = true;
    printf_level(1, "NODE_PROGRAM\n");
    if(node->opr[0]){
        analyse_listdeclnonnull(node->opr[0], glob);
    }
    analyse_func(node->opr[1], glob);
}
```

Figure 6: La fonction analyse_program

```
void analyse_passe_1(node_t root);
void analyse_program(node_t node);
void analyse_listdecl(node_t node, bool g);
void analyse_listdeclnonnull(node_t node, bool g);
void analyse_vardecl(node_t node, bool g);
void analyse_type(node_t node);
void analyse_listtypeddecl(node_t node, bool g, node_type t);
void analyse_decl(node_t node, bool g, node_type t);
void analyse_func(node_t node, bool g);
void analyse_listinst(node_t node);
void analyse_listinstnonnull(node_t node);
void analyse_inst(node_t node);
void analyse_block(node_t node, bool g);
void analyse_expr(node_t node);
void analyse_listparamprint(node_t node);
void analyse_paramprint(node_t node);
void analyse_ident(node_t node);
void analyse_intval(node_t node);
void analyse_string(node_t node);
void analyse_bool(node_t node);
```

Figure 7: La fonction analyse_xxx

Parmi ces fonctions **analyse_xxx**, il y a certaines fonctions qui ont effectué des procédures différentes. Ce sont les fonction suivantes:



Après l'exécution du programme, en prenant l'exemple donnée dans le sujet, nous avons obtenu l'arbre de forme comme la figure 8 qui est similaire avec la figure 5, la seule différence est l'ajout de *offset*.

V. Génération du code (passe 2)

La génération du code permet de parcourir l'arbre du programme en le transformant en programme assembleur.

Pour réaliser cette génération, tout d'abord nous déclarons une variable globale de type *bool fin* pour détecter si nous avons atteint la fin de l'arbre. Si c'est le cas, *fin=true* sinon *fin=false*. Ensuite, pour parcourir les nodes, nous définissons une fonction **node_suivant(node_t root)**. Dans cette fonction nous utilisons la boucle for et le *bool fin*, dans cette boucle, si nous n'avons pas atteint la fin du node, nous allons parcourir la fonction **node_suivant(node_t root)** de manière récursive. Puis, nous utilisons la fonction **create_inst_data_sec()** pour créer **.data** dans le fichier assemble **.s**. Alors pour traiter les différents nodes, nous définissons la fonction **node_traite(node_t root)** en utilisant switch case pour examiner la nature du node.

Si la nature du node est *NODE_DECL*, nous passons à la fonction **gen_code_decl(node_t root)**. Dans cette fonction, nous allons utiliser une autre variable globale *bool global*. Si cette variable est true, c'est-à-dire qu'elle est dans le **main.c**, c'est la déclaration globale, alors nous entrons dans le **.word** par la fonction **create_inst_word()** avec l'identifiant et la valeur. Sinon, on crée l'instruction ori et sw par la fonction **create_inst_ori()** et **create_inst_sw()**.

Si la nature du node est *NODE_FUNC*, nous crée le **.text** par **create_inst_text_sec()**, puis nous alloue la pile pour les variables locales à l'aide les 2 fonctions suivant:

```
set_temporary_start_offset(root->offset)
```

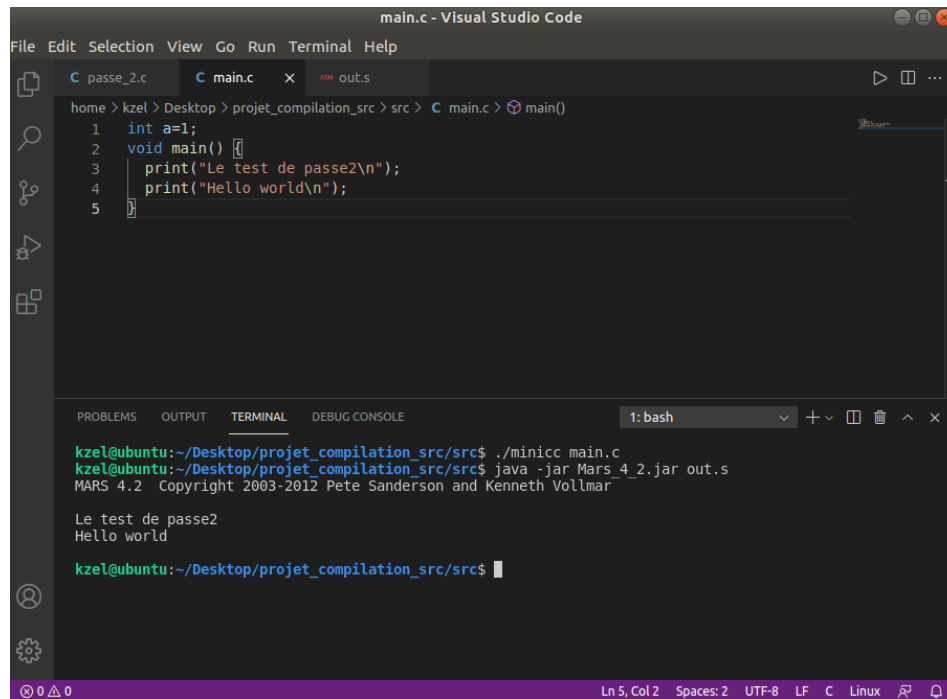
```
set_max_registers(get_num_registers())
```

Et le distribue **label "main"** et l'instruction en utilisant la fonction **create_inst_addiu(29,29,-root->offset)**.

Pour ajouter le string dans **.asciiz**, nous utilisons la fonction **get_global_strings_number()** pour obtenir le nombre de string. Puis nous ajoutons le string dans **.asciiz** avec la fonction **create_inst_asciiz(NULL, get_global_string(char))**. Et à la fin, nous changeons le *bool global* en false, car nous entrons dans la main.

Dans le cas de *NODE_STRINGVAL*, nous définissons la fonction **gen_code_print(node_t root)** avec la création de **create_inst_lui(4,0x1001)** à l'adresse **0x1001** et les instructions **ori(4, 4, root->offset)** et **ori(2, 0, 4)**.

Dans ce cas, nous avons bien réussi à afficher les strings



The screenshot shows the Visual Studio Code editor with a file named `main.c` open. The code in the editor is as follows:

```
1 int a=1;
2 void main() {
3     print("Le test de passe2\n");
4     print("Hello world\n");
5 }
```

Below the editor, the terminal window shows the execution of the program. The commands and their outputs are:

```
kzel@ubuntu:~/Desktop/projet_compilation_src/src$ ./minicc main.c
kzel@ubuntu:~/Desktop/projet_compilation_src/src$ java -jar Mars_4_2.jar out.s
MARS 4.2 Copyright 2003-2012 Pete Sanderson and Kenneth Vollmar
Le test de passe2
Hello world
kzel@ubuntu:~/Desktop/projet_compilation_src/src$
```

Figure 9: affichage des strings

Dans le cas de *NODE_AFFECT*, nous utilisons **ori(current_reg, r0, valeur de variable)** puis **sw(current_reg, root->opr[0]->offset, 29)**.

Et dans le cas des opérateurs, nous définissons la fonction **gen_code_ope(node_t root)**. Nous vérifions aussi la condition entre les deux *TOKEN_INTVAL* et la nature des opérateurs avec les différentes instructions. Puis nous allouons le registre *reg_1* avec le registre *reg_0* en avant pour compléter les instructions et lâcher le registre à la fin.

VI. Ligne de commande

Pour la ligne de commande, tout d'abord nous vérifions si le premier argument entré est un fichier `.c`, si c'est le cas, nous allons mettre à jour de la variable *infile*. Sinon, nous le comparons avec les différentes options possibles à l'aide de la fonction **strcmp()**. Après la comparaison, si les deux string sont pareil, retourner 0, sinon 1.

Dans le cas où le retour de `strcmp` égale à 0, nous allons effectuer la procédure suivant selon les différentes options:

- -b: On affiche notre nom.
- -h: On affiche l'aide de toutes les commandes.
- -o: On compare le suffix `.s` pour détecter si le type du mips assembler est correct ou pas, si oui on change le nom de outfile, sinon afficher l'erreur.
- -t: On change le niveau de trace allant de 1 à 5 avec **`atoi(option)`**.
- -r: On change le max de registre à utiliser entre 4 et 8 avec **`atoi(option)`** et **`set_max_registers()`**.
- -s: On arrête la compilation après l'analyse de syntax, la variable `stop_after_syntax` change en true.
- -v: On arrête la compilation après la passe 1, la variable `stop_after_verif` change en true.

VII. Messages d'erreur

Pour trouver les erreur en faisant le test, nous avons ajouté des messages d'erreur dans les différents fichiers en utilisant `fprintf` de forme suivant:

Error line <numéro de ligne>: <description informelle du problème>

La figure 10 représente le message d'erreur produit lorsqu'on entre dans la fonction **main** mais son type de déclaration n'est pas void dans la passe 1. Et après chaque erreur, nous avons ajouté la fonction **`exit(1)`** pour arrêter le programme.

```
fprintf(stderr, "Error line %d: Le type doit être void\n", node->lineno);  
exit(1);
```

Figure 10: Formatage des messages d'erreur

VIII. Test

Dans le fichier Tests, il y a deux types différents de fichier, ce sont KO et OK. Le fichier KO contient les tests qui génèrent des erreurs, nous devons le trouver à travers notre compilateur. Le fichier OK contient les tests qui ne renvoient pas d'erreur. Pour lancer taper la commande `$ sh xxx.sh`

1. Test Syntaxe

Pour vérifier que nous avons bien réalisé l'analyse syntaxique. Nous avons créé les fichiers .c dans le **Syntax_KO** et **Syntax_OK**.

Dans le **Syntax_KO**, nous testons les fausses conditions pour *if*, *else*, *for* et *while*, le manque du virgule et les erreurs de type de déclaration.

Dans **Syntax_KO**, nous testons les opérateurs logiques comme *and*, *nand*, *or*, *nor*, *xor*, *xor* etc; les conditions pour: *if*, *else*, *for*, *while* et *dowhile*; les opérateurs algorithmiques: *plus*, *minus*, *mul*, *div*, *mod*; le *print* et les opérateurs relationnels: *ge*, *gt*, *le*, *lt*.

2. Test Verif

Afin de vérifier que la passe 1 est correctement implémentée, nous avons créé les fichiers .c dans le **Vérif_KO** et le **Vérif_OK**.

Dans le **Vérif_KO**, nous avons créé les fausses conditions pour *if*, *else*, *for*, *while*, *if_egale*; les erreurs entre les types pour les opérateurs logiques, comme *and*, *band*, *bor*, *bxor* etc; les opérateurs algorithmiques: *plus*, *minus*, *mul*, *div*, *mod*; et les opérateurs relationnels: *eq*, *le*, *lt*, *ge*, *gt*.

Dans le **Vérif_OK**, nous avons testé les opérateurs logiques comme *and*, *nand*, *or*, *nor*, *xor*, *xor* etc; les conditions pour: *if*, *else*, *for*, *while*, *dowhile*, *if_for*, *if_if*, *dowhile_if*, les opérateurs algorithmiques: *plus*, *minus*, *mul*, *div*, *mod*; le *print* et les opérateurs relationnels: *ge*, *gt*, *le*, *lt*, *eq* entre *bool* et *int*.

3. Test Gencode

Pour vérifier que la passe 2 est bien implémentée, nous avons créé les fichiers .c dans le **Gencode_KO** et le **Gencode_OK**.

Dans le **Gencode_KO**, nous avons créé 2 fichiers qui représentent la division par 0 et le module par 0. C'est impossible de calculer.

Dans le **Gencode_OK**, nous suivons l'indication du poly, et nous prenons le même fichier qui se situe dans le **Vérif_OK** et nous les modifions un peu avec *print*.

IX. Les Difficultés

Dans la passe_1, nous avons encore des problèmes qui ne sont pas résolus. Nous remarquons que quand le programme appelle les variables qui sont déjà déclarées, la valeur offset pour ces variable n'est pas correcte, elle égale toujours à 0. En plus, l'offset dans *NODE_FUNC* n'est pas correct. Donc nous pensons qu'il y a des problèmes d'affectation dans la fonction *analyse_func* et *analyse_ident*.

Pour le passe_2, nous avons rencontré aussi le problème sur la valeur de offset qui reste toujours en 0. En plus, nous ne pouvons pas allouer les piles. Aussi nous n'avons pas de temps pour finir les *NODE_IF*, *NODE_FOR*, *NODE_WHILE* et *NODE_DOWHILE*.

X. Conclusion

A partir de ce projet, on comprend comment construire un **compilateur** qui transfère un programme source en programme assembleur sans erreur en utilisant la langage **MiniC** et les outils **Lex** et **Yacc**. Cette expérience nous donne plus envie d'étudier les différents langages de programmation. En plus, dans ce projet, nous avons appris comment définir notre propre langage et vérifier les erreurs qui peuvent être produites en exécution.

XI. Bibliographie

- [Write text parsers with yacc and lex – IBM Developer](#)
- Document du projet compilation, Auteur: M.Meunier