Compilation
Compte-rendu

BOUCHARD Paul
UT Robert

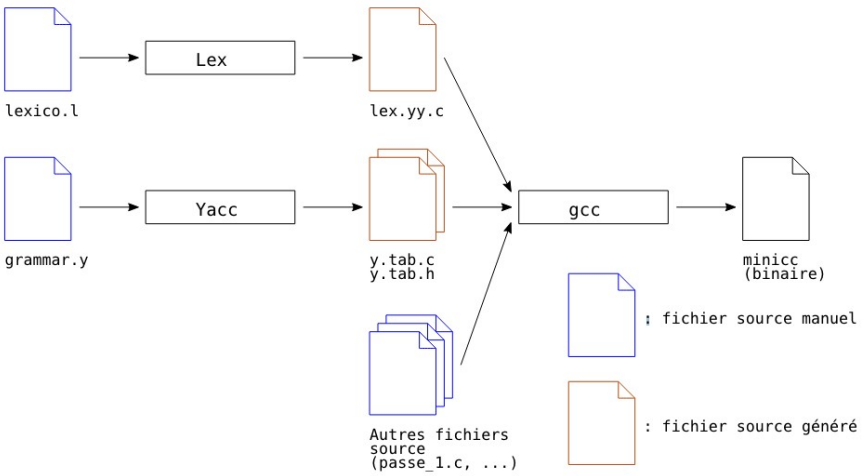
EISE-4

Introduction

L'objectif de ce projet est de développer un compilateur MiniC, qui est une version simplifiée du langage C, comportant quelques variations et restrictions. Ce qui implique la création d'un programme capable de convertir un code source en langage assembleur. Le langage utilisé pour écrire les programmes source est appelé. Le langage assembleur ciblé est le langage Mips.

Nous allons dans un premier temps nous concentrer sur l'analyse lexicale de notre programme source en associant aux caractères de ce dernier une suite de tokens pour chaque élément : variables, fonctions, opérateurs, mots réservés, etc. Puis vient l'analyse syntaxique qui vérifie que la suite de tokens en sortie de l'analyse lexicale est valide. Par exemple, une succession de deux tokens associés au mot-clé for n'est pas valide syntaxiquement.

Par la suite, une analyse sémantique (passe de vérification) permettra de gérer les contextes, un nom de variable peut être utilisé sans avoir été déclaré, ou une variable booléenne additionnée avec une variable entière. Enfin, une deuxième passe aura pour but de générer le code assembleur du programme source



Analyse lexicale et syntaxique

Pour commencer, nous avons mis en place l'analyse lexicale à l'aide de l'outil Lex. Dans le fichier lexico.l on définit avec la syntaxe Lex ce que représente un caractère ou un ensemble de caractères : une lettre, un chiffre, un identificateur, une chaine de caractères, etc. Voici quelques exemples :

```
LETTRE          [a-zA-Z]
CHIFFRE         [0-9]
IDF             {LETTRE}({LETTRE}|{CHIFFRE}|_)*
```

On associe des tokens aux mots réservés, caractères spéciaux et opérateurs ce qui va permettre par la suite de construire notre arbre.

```
"void"          RETURN(TOK_VOID);
"int"           RETURN(TOK_INT);
"}"             RETURN(TOK_RACC);
"="             RETURN(TOK_AFFECT);
```

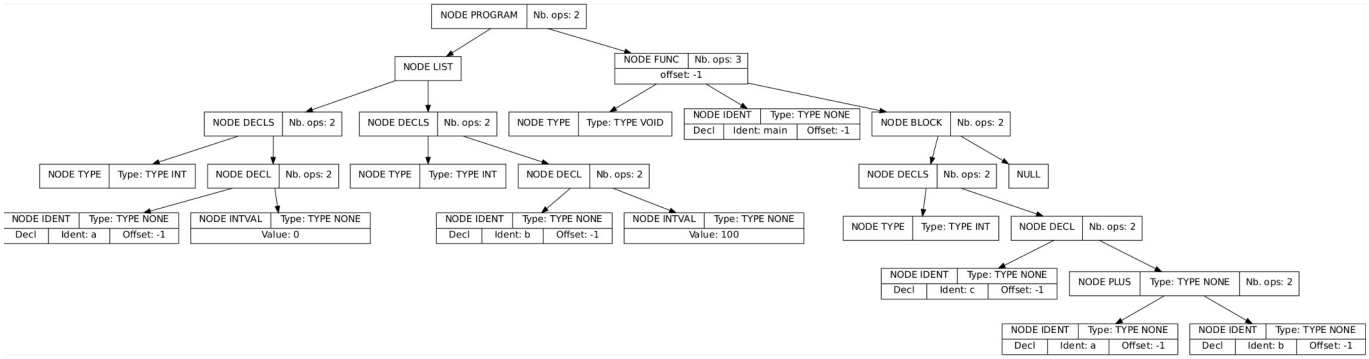
Dans l'analyse syntaxique (fichier grammar.y), l'outil Yacc nous permet de définir les tokens et les règles de MiniC en définissant ce que sont les différentes « entités » du programme source : une déclaration, une liste de déclaration, une expression, un bloc, etc. Et ainsi créer des nœuds (nodes) ayant différents paramètres (type node_t) permettant de construire l'arbre. Nous avons aussi les fonctions make_node et make_node_special (pour les nodes qui ont besoin d'être modifiés en profondeur) qui créent les nodes.

Voici un exemple pour les expressions de multiplication et division :

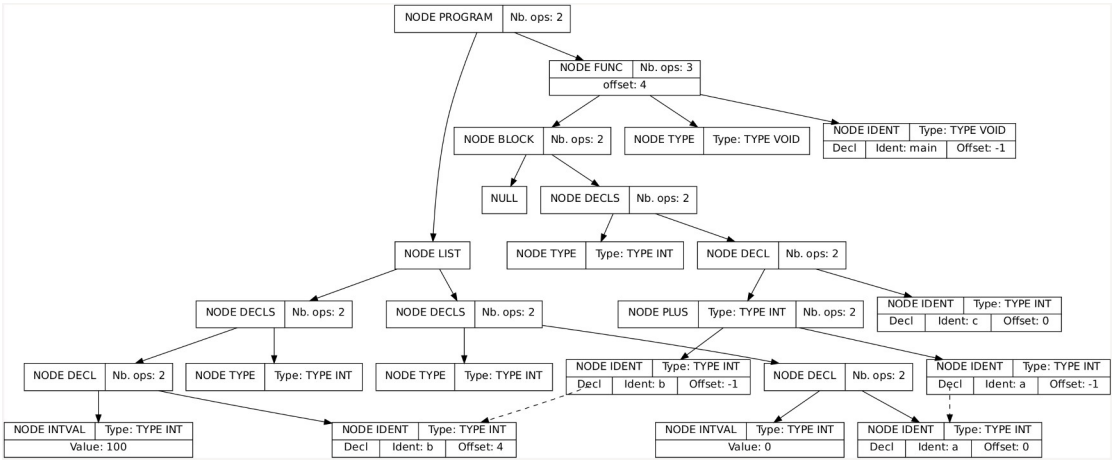
```
expr    :expr TOK_MUL expr { $$ = make_node(NODE_MUL, 2, $1, $3);}
        |expr TOK_DIV expr { $$ = make_node(NODE_DIV, 2, $1, $3);}
```

Passe de vérification contextuelle

Cette passe a pour objectif d'explorer l'arbre en profondeur afin de vérifier si le programme à l'origine de l'arbre est syntaxiquement correct. Les erreurs de syntaxe peuvent venir de l'utilisation de variables non déclarées, de valeurs affectées à une variable non compatible avec le type de cette dernière ou encore d'opérations entre variables de type différents. En plus des vérifications diverses à faire, cette passe va permettre de modifier les champs de différentes nodes.



De sorte à ce que l'on passe d'un arbre comme ci-dessus à ci-dessous :



Pour réaliser cette passe, nous avons décidé de d'abord nous attaquer aux modifications des champs et ainsi obtenir l'arbre après passe 1 pour ensuite vérifier le bon fonctionnement de cet arbre. La première fonction (faite par nous) à être appelée est nommée analyse_decl_globloc(node,bool,TYPE) et va modifier les champs global_decl (selon l'argument bool), type (selon l'argument TYPE) et offset des NODE_IDENT lors des occurrences de déclarations. La fonction sera donc ainsi appelée deux fois : une première fois pour les variables globales et une seconde fois pour les variables locales. La fonction fait un parcours en profondeur récursif sur les fils de la node prise en argument.

La seconde fonction (faite par nous) va mettre à jour les types de toutes les nodes qui en ont besoin (NODE_PLUS,NODE_NOT,etc) et les offset de certaines nodes (NODE_FUNC,NODE_STRINGVAL,etc) et tout cela via un nouveau parcours récursif en profondeur et un switch case sur la nature des nodes. Pour mettre à jour les champs des autres NODE_IDENT d'occurrence d'utilisation, on va chercher les nodes des occurrences de déclarations. Une fois trouvées, on appellera une nouvelle fonction parcourant toujours récursivement les nodes et prenant en en argument les NODE_IDENT qu'on vient de trouver. Cette fonction va chercher les NODE_IDENT d'occurrence d'utilisation faisant appel aux nodes prises en argument et en conséquence mettre à jour les types et le champ decl_node.

Après tout ceci, l'arbre est censé être terminé. On va donc lui faire passer des vérifications via parcours récursif et switch case sur la nature des nodes à l'aide de la fonction que l'on a nommée verif_context. Pour la majorité de ces dernières, il suffit de vérifier le champ type des fils des nodes (mauvaise assignation au niveau des types)). Pour vérifier la présence de variables non déclarées, il suffit de chercher la présence de NODE_IDENT du type TYPE_NONE (toutes les NODE_IDENT sont censées avoir un type attribué TYPE_BOOL ou TYPE_INT). Nous avons aussi créé la fonction checkup_inception qui va vérifier si une variable a été utilisée en même temps d'être déclarée (ex : int a = a + 1;) en comparant les champs ident de la variable déclarée et de la variable attribuée (si il y'en a une).

Génération de code

La passe 2 consiste à générer le code en assembleur MIPS du programme en miniC de sorte à obtenir un fichier contenant un programme comme ci-dessous :

```
# Declaration des variables globales
.data

start: .word 0
end:   .word 100
.asciiz "sum: "
.asciiz "\n"


# Programme
.text


main:
    # Prologue : allocation en pile pour les variables locales
    # i se trouve a l'emplacement 0($29)
    # s se trouve a l'emplacement 4($29)
    # e se trouve a l'emplacement 8($29)
    # sum se trouve a l'emplacement 12($29)
    addiu $29, $29, -16
    # s = start
    lui   $8, 0x1001
    lw    $8, 0($8)
    sw    $8, 4($29)
```

Malheureusement, nous n'avons pas pu finaliser cette passe par manque de compétences et de compréhension du sujet. Cependant, nous avons tout de même commencé à écrire quelques fonctions ici et là.

Tout d'abord, nous avons créé deux fonctions permettant de remplir la section .data : search_string() et gen_decl(node). La première va chercher toutes les chaînes de caractères et créer des .asciiz correspondants. La seconde va chercher les NODE_IDENT correspondant aux déclarations de variables globales et créera ensuite les .word correspondant.

Ensuite, nous avons créé une fonction qui a pour but d'afficher les chaînes de caractères lorsque cela est fait dans le programme miniC (NODE_STRINGVAL pour .asciiz ou NODE_IDENT pour l'affichage des valeurs des variables). Notre dernière fonction s'appelle gen_main et elle devait consister en la génération du code assembleur du main présent dans le programme miniC. Fonctionnant sur un parcours en profondeur récursif, un switch case sur la nature des nodes aurait permis de créer les instructinos assembleur correspondantes.

Tests et automatisation

Afin de tester nos programmes, nous avons réalisé des fichiers tests C. Trois dossiers de tests sont présents : Syntaxe, Verif et Gencode pour chacune des étapes de l'élaboration du compilateur avec des dossiers KO et OK, OK étant des tests censés passer et KO censés échouer.

A l'aide d'un script shell, nous allons automatiser l'exécution de ces tests afin de voir directement lesquels passent et où sont les erreurs pour les tests KO. Les deux lignes suivantes permettent de récupérer la commande ./minicc avec le chemin et les flags voulus (2>&1 permet de récupérer stderr dans stdout pour afficher correctement l'erreur) puis l'afficher avec echo.

```
result=$(./minicc "$flags" "$root" 2>&1)
echo "${root}: ${result}"
```

Pour passer les tests, il suffit de lancer la commande suivante :

./test_auto.sh

Les tests KO détaillent la ligne de l'erreur et le type d'erreur et les tests OK incrémentent un compteur s'ils passent. On obtient un affichage dans ce genre :

```
utrobert@Lenovo-R:~/Polytech/Compilation/Compilotron/projet_compilation_src$ ./test_auto.sh
-- Tests/Syntaxe --
- KO -
Tests/Syntaxe/KO/ko10print.c: Error line 6: Lexical error
Tests/Syntaxe/KO/ko11semicol.c: Error line 4: syntax error
Tests/Syntaxe/KO/ko12comma.c: Error line 3: syntax error
Tests/Syntaxe/KO/ko13lpar.c: Error line 5: syntax error
Tests/Syntaxe/KO/ko14rpar.c: Error line 5: syntax error
Tests/Syntaxe/KO/ko15lacc.c: Error line 8: syntax error
Tests/Syntaxe/KO/ko16racc.c: Error line 9: syntax error
Tests/Syntaxe/KO/ko1void.c: Error line 3: syntax error
Tests/Syntaxe/KO/ko2.c: Error line 4: syntax error
Tests/Syntaxe/KO/ko3.c: Error line 2: syntax error
Tests/Syntaxe/KO/ko4.c: Error line 3: syntax error
Tests/Syntaxe/KO/ko5.c: Error line 4: syntax error
Tests/Syntaxe/KO/ko6.c: Error line 7: syntax error
Tests/Syntaxe/KO/ko7for.c: Error line 7: syntax error
Tests/Syntaxe/KO/ko8while.c: Error line 8: syntax error
Tests/Syntaxe/KO/ko9print.c: Error line 5: syntax error


- OK -
12 tests passés sur 12



-- Tests/Verif --
- KO -
Tests/Verif/KO/ko1bool_int.c: Error line 3: context error, type conflict
Tests/Verif/KO/ko2addboolint.c: Error line 6: context error
Tests/Verif/KO/ko3varnondecl.c: Error line 5: context error
Tests/Verif/KO/ko4for.c: Error line 6: context error !!
Tests/Verif/KO/ko5decl.c: Error line 4: context error
Tests/Verif/KO/ko6decl.c: Error line 3: context error
Tests/Verif/KO/ko7main.c: Error line 2: context error, function is not main
Tests/Verif/KO/ko8not.c: Error line 3: context error
Tests/Verif/KO/ko9bnot.c: Error line 4: context error !


- OK -
6 tests passés sur 6
```