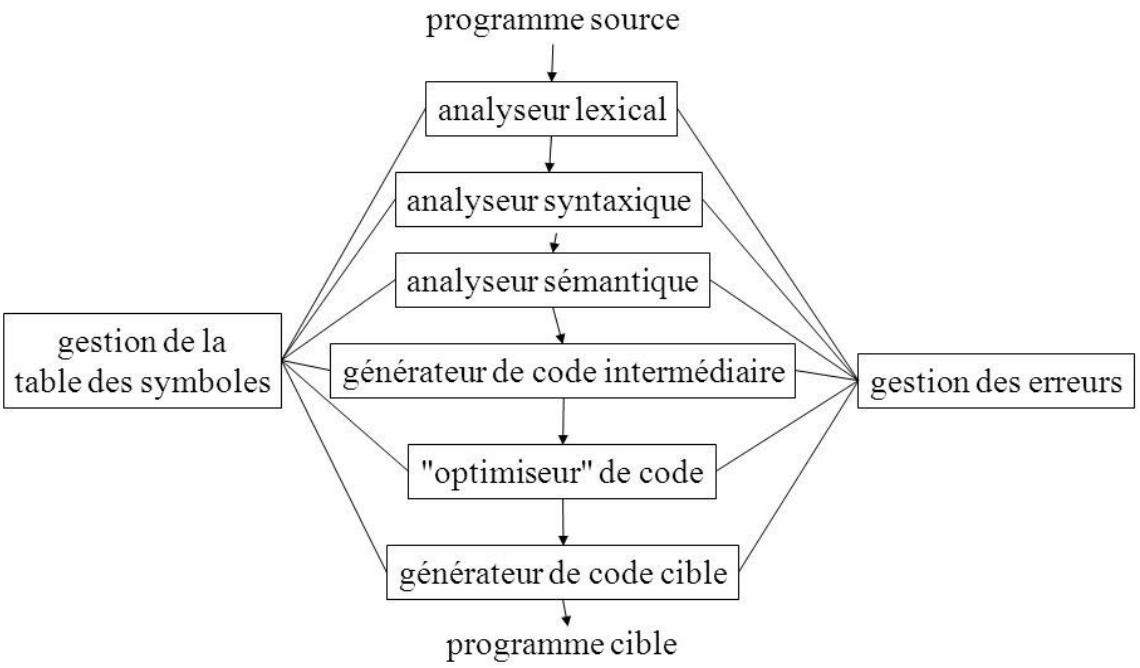
Gruaux Apolline
Boujon Axel

Rapport de projet Compilation



Dans le cadre de l'UE Compilation encadré par M. Meunier

Année universitaire 2019-2020

Sommaire

Schéma de l'architecture

Compilation/
- src/
  - Tests/
    - Gencode/
      - KO/
      - OK/
    - Syntaxe/
      - KO/
      - OK/
    - Verif/
      - KO/
      - OK/
  - utils/
    - context.h
    - env.h
    - libminiccutils.a
    - mips_inst.h
    - registers.h
  - analyse.c
  - analyse.h
  - common.c
  - common.h
  - def.h
  - grammar.y
  - lex.yy.c
  - lexico.l
  - y.tab.c
  - y.tab.h
  - Makefile

Analyse lexicale

Il s'agit d'abord ici de compléter le fichier lexico.l. Il va nous permettre de transformer une chaîne de caractères en tokens.

Pour ce faire on utilisera ces identificateurs définis de la façon suivante :

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

TOUT            [\x20-\x7E]
PONCTUATION     [!#$%&'()*+,-./:;<=>?@[]_`{|}~ ]

CHAINE_CAR      [\x20-\x21]|[\x23-\x5B]|[\x5D-\x7E]
CHAINE          "({CHAINE_CAR}|\"|\n)*"
COMMENTAIRE     "//"({TOUT})*\n
```

Également les tokens suivant :

"void"	TOK_VOID
"int"	TOK_INT
"for"	TOK_FOR
"if"	TOK_IF
"else"	TOK_ELSE
"while"	TOK_WHILE
"do"	TOK_DO
"print"	TOK_PRINT
"bool"	TOK_BOOL
"true"	TOK_TRUE
"false"	TOK_FALSE
"="	TOK_AFFECT
"+"	TOK_PLUS
"-"	TOK_MINUS
"*"	TOK_MUL
"/"	TOK_DIV
"=="	TOK_EQ
"!="	TOK_NE
"&"	TOK_BAND
"|"	TOK_BOR
"&&"	TOK_AND
"||"	TOK_OR
"%"	TOK_MOD

"<"	TOK_LT
">="	TOK_GE
"<="	TOK_LE
"!"	TOK_NOT
"~"	TOK_BNOT
"^"	TOK_BXOR
";"	TOK_SEMICOL
","	TOK_COMMA
"("	TOK_LPAR
")"	TOK_RPAR
"{"	TOK_LACC
"}"	TOK_RACC
">>"	TOK_SRA
"<<"	TOK_SLL
">>>"	TOK_SRL
">"	TOK_GT
"<"	TOK_LT

Analyse syntaxique

Dans cette partie nous allons compléter le fichier grammar.y, dans lequel nous allons définir la grammaire de notre langage. Le programme permettant l'analyse va être généré par Yacc à partir des règles "de bases" qu'on lui a donnée et va déterminer toutes les sous-règles qui en découlent.
L'analyse syntaxique va produire un arbre, à partir de la séquence de tokens, respectant les règles de la grammaire définie.
Pour créer ces règles de grammaire, nous définissons d'abord de quel type de déclaration il s'agit, puis de la syntaxe qui lui est associé. Par exemple :

```
program:
        listdeclnonnull maindecl
        | maindecl
        ;
```

Ici on déclare deux règles syntaxiques différentes pour program. On ajoute ensuite des actions en fonction des règles. Toujours dans cet exemple :

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

Ici les deux règles vont effectuer les mêmes actions : créer une node "NODE_PROGRAM", mais dans un cas en prenant en compte la liste de déclarations non nulles définie avant et dans l'autre cas en initialisant à NULL. Ainsi, on peut accéder aux argument de la règle avec $ suivi du numéro de l'argument. Ainsi, pour la première règle maindecl correspond à $2 et $1 pour la deuxième.

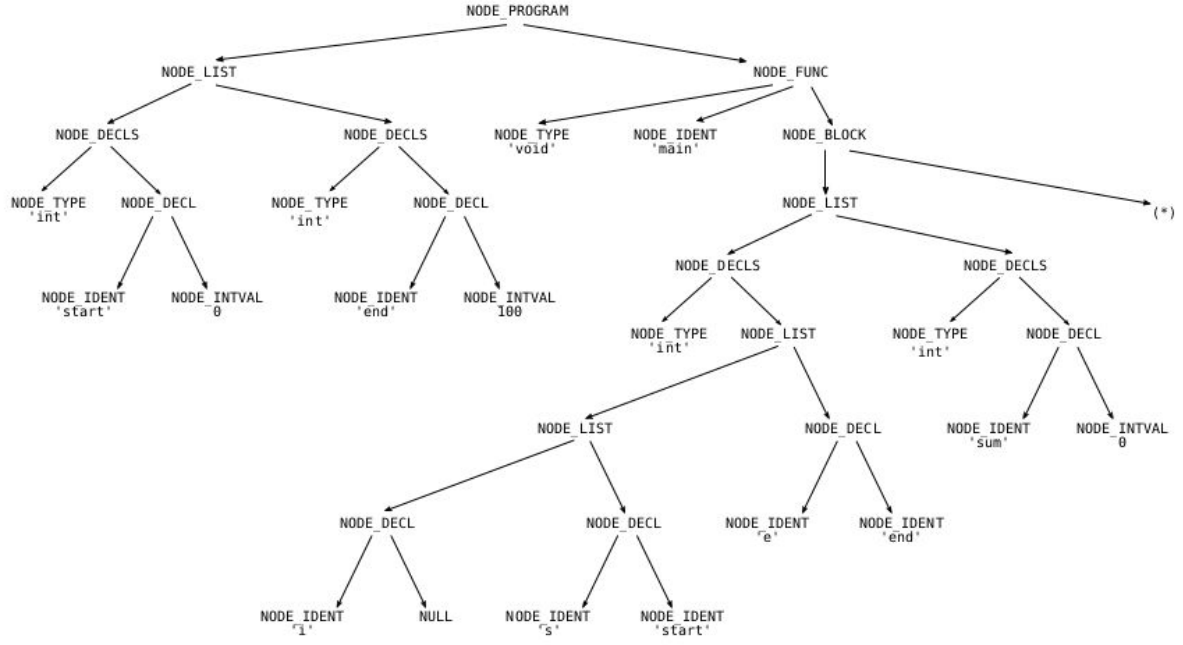
On va ainsi écrire une fonction make_node qui crée un noeud de l'arbre. Elle va nous permettre de faire l'allocation. On va l'utiliser pour associer tous les tokens définis précédemment avec les noeuds qui conviennent avec la quantité de registres utiles. On considère l'allocation réussie si on peut générer l'arbre en utilisant dump_tree puis en le visualisant.

La fonction node_t make_node(node_nature nature, int nops, ...), va d'abord faire à une autre fonction que nous avons créée, node_t new_node(). Celle-ci a pour but d'initialiser une nouvelle node et de la renvoyer. Si l'opération s'effectue sans problème, un switch va permettre de créer la node selon sa nature :

```
switch(nature){
        case(NODE_BOOLVAL):
        case(NODE_IDENT):
        case(NODE_INTVAL):
        case(NODE_STRINGVAL):
        case(NODE_TYPE):
        default:
}
```

Pour libérer la mémoire ainsi allouée la méthode la plus rapide semble être d'utiliser l'algorithme de recherche en profondeur car on peut facilement l'appliquer sur notre arbre (se comporte comme un graphe)

Finalement à la fin de cette analyse on devrait avoir un arbre de la forme suivante :


NODE_PROGRAM
NODE_LIST
NODE_FUNC
NODE_DECLS
NODE_DECLS
NODE_TYPE
'void'
NODE_IDENT
'main'
NODE_BLOCK
NODE_TYPE
'int'
NODE_DECL
NODE_TYPE
'int'
NODE_DECL
NODE_LIST
(*)
NODE_IDENT
'start'
NODE_INTVAL
0
NODE_IDENT
'end'
NODE_INTVAL
100
NODE_DECLS
NODE_DECLS
NODE_TYPE
'int'
NODE_LIST
NODE_TYPE
'int'
NODE_DECL
NODE_LIST
NODE_DECL
NODE_IDENT
'sum'
NODE_INTVAL
0
NODE_DECL
NODE_DECL
NODE_IDENT
'e'
NODE_IDENT
'end'
NODE_IDENT
'i'
NULL
NODE_IDENT
's'
NODE_IDENT
'start'

Ligne de commande

Dans cette partie nous allons compléter la fonction `void parse_args(int argc, char ** argv)`, qui va nous permettre d'implémenter différentes options de compilation. Pour cela nous avons à implémenter un certain nombre d'options :

• **-b** : Affiche une bannière indiquant le nom du compilateur et des membres du binôme
• **-o <filename>** : Définit le nom du fichier assembleur produit (défaut : out.s).
• **-t <int>** : Définit le niveau de trace à utiliser entre 0 et 5 (0 = pas de trace ; 5 = toutes les traces défaut = 0).
• **-r <int>** : Définit le nombre maximum de registres à utiliser, entre 4 et 8 (défaut : 8).
• **-s** : Arrêter la compilation après l'analyse syntaxique (défaut = non).
• **-v** : Arrêter la compilation après la passe de vérifications (défaut = non).
• **-h** : Afficher la liste des options (fonction d'usage) et arrêter le parsing des arguments.

Avec comme particularités que -s et -v ne sont pas compatibles sur une même compilation, -b doit être la seule option sur la ligne de commande et enfin que -h arrête la lecture de la ligne de commande et affiche l'aide.

Pour cela nous avons utilisé la fonction `int getopt(argc, argv, "bo:t:r:svh")`, qui nous a permis d'obtenir chacune de ces options. Ainsi, à l'aide d'un switch sur le retour de cette fonction, on obtient un caractère et on peut coder l'action qui lui correspond. Certaines options contiennent des informations importantes à utiliser lors de la compilation, comme le niveau de trace ou le nombre de registres à utiliser. Pour cela nous avons ajouté des variables volatiles et exporté des variables externes :

- extern char * infile : le nom du fichier à compiler
- extern char * outfile : le nom du fichier en sortie de la compilation
- volatile int trace : le niveau de trace à utiliser
- volatile int registers : le nombre de registres à utiliser
- volatile bool syntaxicAnalysis : définir si la compilation s'arrête après l'analyse syntaxique
- volatile bool verificationAnalysis : définir si la compilation s'arrête après la passe de vérification

Au début de parse_args, chacune de ces variables sont initialisées par des valeurs par défaut. On fait ensuite appel à une autre fonction que l'on a créé, `char * getInfile(int argc, char ** argv)`, qui cherche et renvoie le nom du fichier d'extension ".c" à compiler. Si elle n'en trouve aucun, la fonction renvoie `NULL` et la commande renvoie une erreur. S'il y en a plusieurs, elle renvoie le premier trouvé.

Cependant avoir plusieurs noms de fichiers dans une même commande doit retourner une erreur. C'est pourquoi nous avons implémenté un compteur dans parse_args, qui va compter le nombre d'arguments en entrée et le nombre d'arguments traités grâce à getInfile et getpot. Le compteur s'incrémente de 1 pour le infile et les options sans arguments et de 2 pour les options suivies d'un argument correct. Ainsi, si le compteur est égal à argc à la fin de la lecture de la ligne de commande, celle-ci ne possède pas d'arguments supplémentaires parasites et est donc valide. La fonction renvoie 0 pour indiquer que tout s'est bien déroulé. A contrario, s'il y a eu une erreur, la fonction affiche un message d'erreur sur stderr et renvoie -1 en terminant le programme.

Test de syntaxe

Pour vérifier que nous avons correctement codé les étapes précédentes, nous allons écrire des tests.

Les tests suivants doivent générer des erreurs et sont situés dans le répertoir KO :

- declaration.c
  pas de déclaration d'une variable

- for_incomplet.c
  pas assez d'expressions passées dans le for

- instruction_hors_main.c
  instructions en dehors du main

- nom_de_variable.c
  nom de variable commençant par _

- declaration_main.c
  déclaration d'un main dans un main

- declaration_int_hors_main.c
  déclaration de variable en dehors du main

- main_supp.c
  plusieurs main dans le même fichier

- accolades_main.c
  pas d'accolade après le main

- erreur_token.c
  erreur du traitement du token

- no_main.c :
  pas de main dans le fichier

Ces deux tests ne doivent pas renvoyer d'erreur et sont situés dans le répertoire OK :

- main.c :
  programme vide en dehors de la fonction main

- token_main.c
  on teste tous les tokens définis

Passe 1

Dans cette partie nous avons écrit la première passe qui va servir d'analyse syntaxique et de vérifier les problèmes d'utilisation de variables/fonctions. Pour cela, nous avons pensé à utiliser un système d'analyse récursive qui utilise plusieurs fonctions différentes selon le type de donnée à analyser.

En premier, la fonction `void next_analyse(node_t n, void f())` a pour but de lancer l'analyse de chaque noeud suivant le noeud actuel à l'aide d'une fonction f passée en argument. Ainsi, la fonction f dépend du type de noeud à analyser. Par exemple, si c'est un bloc, il va falloir créer un nouveau contexte, puis analyser les noeuds qui lui sont dépendants.

Une deuxième fonction importante est la fonction `void type_rec(node_t n, node_type type)` qui permet de vérifier si les variables int et bool ont été bien initialisées avec de bonnes valeurs. (int avec des entiers et bool avec des booléens). La fonction marche de manière récursive. S'il ne s'agit ni d'un int, ni d'un bool, ni d'un ident, on parcours la liste des descendants pour les analyser avec la même fonction.

A l'aide de ces deux fonctions, nous allons pouvoir lancer une analyse globale du programme dans la fonction `void analyse(node_t n)`. Qui prend en argument un noeud et plus précisément la racine au lancement de la passe. Ainsi, en fonction du type du noeud, on lancera une `next_analyse`, avec en argument une des fonctions suivantes :

```
void analyse_block(node_t n);
void analyse_glob(node_t n);
void analyse_func(node_t n);
void analyse_print(node_t n);
void analyse(node_t n);
```

La fonction `analyse_glob`, sert à analyser et détecter les noeuds qui peuvent être définis dans le contexte global. En faisant attention sur le fait que seule des variables à valeur constantes peuvent être définies.

La fonction `analyse_block`, a tout simplement pour but de créer un contexte quand l'analyse arrive sur un noeud de type NODE_BLOCK.

La fonction `analyse_func`, a le simple but d'analyser la syntaxe d'une fonction. Dans notre cas la seule fonction faisable étant la fonction main, on vérifiera alors les caractéristiques de celle-ci et qu'elle n'existe qu'une seule fois.

La fonction `analyse_print`, a pour but d'analyser les noeuds de type NODE_PRINT.

Ces quatres fonctions ont été complétées en entier, cependant, suite à la situation de notre binôme, nous n'avons pas pu finir la fonction analyse qui donc ne peut pas analyser tous les types de noeuds.

Test Verif

L'option -v permet de visualiser les étapes de la compilation.
Les tests suivants servent à vérifier que la passe 1 a été correctement implémentée.
On test ici la sémantique et non plus la syntaxe c'est à dire que le compilateur comprend ce qui est écrit, mais cela ne correspond pas aux règles qu'il connaît.

Les tests suivants sont contenus dans le dossier KO et doivent générer une erreur :

Opérations hors main

- addition, soustraction, modulo, division et multiplication sur des int
- opérations bit à bit (and, or ,not , complément à 1 , xor)
- opérations sur les booléens (and , or not , xor)
- décalages
- mauvais type d'argument dans les boucles (if , for , while , dowhile)
- mauvaises affectations (entre int et booléens , entre chaîne de caractères et int et entre chaîne de caractères et booléens)
- déclaration de variables ayant le même nom (types identiques et différents)
- mauvaise déclaration du main

Les tests suivants sont contenus dans le dossier OK et ne génère pas d'erreur :

- affectation correctes : pour chaque type on effectue des affectations directes puis à variables.
- déclaration seule : pour chaque type
- boucles avec arguments corrects (if , elsif, else, while , for, do while)

Passe 2

Cette passe a pour but de faire le lien entre l'arbre créé dans la passe 1 et de le faire correspondre à un code MIPS en allouant le bon nombre de registres. Elle n'a malheureusement pas pu être traitée.

Allocateur de registre

L'allocation de registres constitue un étape très important de le génération de code. Durant celle-ci on choisit dans quels registres les variables seront placées pendant l'exécution.

Test Gencode

Les tests suivants doivent générer un erreur à l'exécution et sont contenus dans le fichier KO :

- division par 0
- entier non initialisé
- modulation par 0

Dans le dossier OK on reprend les codes utilisés dans les tests Verif notamment.
On va tester tous les opérateurs, toutes les formes de déclarations et d'allocation pour tous les types.
On reprend également le code token_main.c utilisé dans le dossier de tests Syntaxe pour pouvoir tester un code complet appelant tous les éléments que nous avons testé précédemment dans un même code.

Sources

http://gallium.inria.fr/~maranget/X/compil/poly/lexical.html
https://slideplayer.fr/slide/5373858/
https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Overall-Options.html#Overall%20Options
Documents de cours et de projet - M. Meunier