STRUCIT, mini Compilateur C

Dima Elisabeta Iulia

Modjom Diane Horly

SLUIN602: Compilation

Professeur
Pr. Sid Touati

Licence 3 - Informatique

2021-2022

UNIVERSITÉ CÔTE D'AZUR

Contenu

Chapitre 1. Analyse lexicale. L'outil LEX .

Le programme lex ANSI-C.l représente la première étape de notre Compilateur. Pour utiliser les lexèmes renvoyés par LEX dans la grammaire du programme YACC qui contient la définition des *lexemes(tokens)*, on inclue la bibliothèque `"y.tab.h"`, (`"structfe.tab.h"` sur Fedora).

On s'intéresse aussi à offrir quelques détails en plus concernant les erreurs/avertissements présentes dans le langage frontend. Pour cela, on définit deux fonctions qui compte le nombre de lignes et de colonnes après la lecture de chaque *token* ou d'un commentaire sur plusieurs lignes, en vérifiant pour ce dernier s'il est fini par */.

`void update_line_col();` //Function qui compte le nombre de lignes et colonnes selon le token reconnu.

`void multi_line_comment();` //Fonction qui compte le commentaire sur plusieurs lignes.

Exemple des reconnaissances des lexèmes :

```
 {L}({L}|{D})*   {
                        for(int i=0;i<10;i++){
                        if(strcmp(yytext, keywords[i])==0)
                        yyerror("Un identificateur ne peut pas prendre
le nom'dun mot clé."); }
                        strcpy(yylval.ystr,yytext);
                        update_line_col(); return(IDENTIFIER);}

     "!="          {    update_line_col(); return(NE_OP);}
```

Chapitre 2. Analyse syntaxique. L'outil YACC.

Conflicts shift/reduce et associativite

Pour ne pas avoir de conflits de type shift/reduce ainsi que pour évaluer correctement des expressions booléennes ou arithmétiques, on rajoute des règles de priorité gauche et droite comme vu en cours ainsi que des règles de (non)associativité pour le '-' unaire.

La dérivation du non-terminal selection_statement produit un conflit shift/reduce, causé par l'ELSE en suspens. Le conflit est résolu par l'insertion des règles suivantes:

`%nonassoc ELSE`

`%nonassoc PRIORITY_LOWER_THAN_ELSE` (pour permettre la continuation de l'analyse lexicale d'un ELSE s'il existe).

`%prec PRIORITY_LOWER_THAN_DECL_LIST` (pour STRUCT IDENTIFIER)

Changement de la grammaire initiale

On a décidé de changer quelques parties de la grammaire initiale du projet à cause des certaines dérivations qui parait incontrôlables, notamment celles du non-terminal *declarator* et *direct_declarator* (le fait que ces non-terminaux forment une récursivité relativement complexe) ainsi que dans la dérivation du *parameter_list* ou *struct_declaration*.

Concernant les expressions, on a modifié les non-terminal *unary_operator*, en enlevant l'option '-'. Ce changement nous a facilité aussi la tâche dans le sens où il n'y a plus besoin de vérifier si l'opérant gauche a un operateur unaire '-' pour interdire son affectation ou le calcul `sizeof(unary_expression)`.
Pourtant, pour permettre des opérations unaires, on a rajoute la dérivation suivante :

```
additive_expression
    : '-' multiplicative_expression %prec UNARY_MINUS
```

Dans la suite, on donne quelques exemples du code considéré erroné par un compilateur GCC de C, pourtant accepté en STRUCIT- frontend :

```
1. int f()()()()()(){/* compound_statement */}
// C: error: function cannot return function type 'int ()'
```

Cause: *direct_declarator* -> *direct_declarator* ( ) -> .... -> (((((... IDENT ...)))))
Solution: changement de la grammaire

```
2. int just_function_name{ /* compound_statement */ }
// C: error: expected ';' after top level declarator
```

Cause: dès qu'il n'y a pas de liste de paramètres, le compilateur attend une déclaration d'un identificateur, d'où l'erreur provoquées par les crochées.
Solution: changement de la grammaire.

```
3. void (*not_a_function)(int a,int b){ int i;}
// C: error: expected ';' after top level declarator
```

Cause: les dérivations de *declarator* et *direct_declarator*. Not_a_function est un pointeur vers une fonction mais pas une fonction. La partie sans {...} est correcte que s'il s'agit de déclarer un pointeur vers une fonction.
Solution: dès l'apparition d'une définition d'un fonction, on vérifie si elle est de type pointeur_sur _onction. Dans ce cas, c'est une erreur. Sinon, il s'agit d'une déclaration d'un pointeur sur une fonction et donc juste. Ces vérifications sont facilitées aussi par le changement de la grammaire.

```
4. int a;
     a = 10(2+3);
// C: error: called object type 'int' is not a function or
function pointer
```

Cause : *postfix_expression* ( ) -> *primary_expression* ( ) -> CONSTANT ( )
Solution succincte : actions sémantiques adéquates pour vérifier si la variable avant () est une constante.

```
5. a(1+2)=b(2+3)=4;
// C: error: expression is not assignable
```

```
6. int a;
     a=1=2;
// C: error: expression is not assignable
```

Cause: *unary_expression*= *expression*-> *unary_expression* = *unary_expression*= *expression* -> … -> IDENTIFIER = CONSTANT = CONSTANT
Solution : *Est-ce que pour chaque production expression -> unary_expression = expression, le token à gauche est une valeur constante?*

```
7. *&*&*&id=-1+2;
8. 1+2*4;
```

Solution: verification dans l'analyse semantique. *Expression_statement* doit être un appel de fonction, ou un assignemnt.

Observation :
Dès que le non terminal *declaration* peut se dériver en *struct_specifier declarator*; la définition d'une structure peut devenir à son tour incorrecte à cause de la dérivation du *declarator* . Exemple :

```
struct test{
    int a;
    int b;
}ttt()()()();
```

Au lieu d'avoir une tolérance relativement grande pour la définition/déclaration d'une fonction, on restreint le spectre des possibilités en changeant la dérivation du non-terminal *declarator* et *direct_declarator*. Notre intention est d'avoir une grammaire qui soit toujours en concordance avec les contraintes du projet et qui peut être assez facilement vérifiée par l'analyse sémantique.

Remarque : Le changement de la grammaire nous permet de ne pas arriver à faire une l'analyse compliquée d'une déclaration/définition d'une fonction qui de tout façon ne sera pas acceptées en STRUCIT-frontend.

Exemple: `int (* idd(int a,int b))(int c){ int i; }` // `idd` est une fonction qui prends deux paramètres entières et qui retourne un pointeur vers une fonction qui prends un paramètre entier et qui retourne un entier. (Évidement, le type de retour de cette fonction n'est pas un type de base ou pointeur sur structure ou constante. Cela veut dire que cette définition va générer une erreur).

La grammaire changée est plus exigeante, ne pas en acceptant des définitions/ déclarations des fonctions qui retourne un pointeur sur pointeur ou sur fonction. De même façon, elle ne va pas accepter des parenthèses redondantes pour les déclarations des variables ou définitions des fonctions, même si elles sont acceptées par GCC. Exemple : `int ((((test(int a,int b))))) `

Cependant, elle acceptera quelques parties de code incorrectes mais qui vont être passées à l'analyse sémantique.

Pour ne pas alourdir la grammaire, on va partager des dérivations entre une déclaration d'une variable et une déclaration ou définition d'une fonction. En effet pour nous, une déclaration d'une fonction est une déclaration d'une variable suivie par une liste des paramètres entre parenthèses.
De plus, une définition d'une fonction est une déclaration d'une fonction (mais pas un pointeur sur fonction), en ayant comme continuation le *compound_statement* au lieu de ';', en vérifiant dans ce temps à l'aide des actions sémantiques si la syntaxe est bien juste.

↘ Il y a aussi d'autres changements qui peuvent être observées en comparant les deux grammaires.

Chapitre 3. Bibliothèque personnelle, definitions.h

Pour la gestion de la table de symboles, ainsi que la création de l'arbre syntaxique décorée qui facilite la génération de code cible, on crée une bibliothèque personnelle qui va contenir les définitions des structures qu'on va utiliser, des constantes pour l'arbre syntaxique et des fonctions présentes en structfe.y et gen_code.c.
Le fichier header commence par la définition des constantes nommée explicitement selon usage. Comme chaque non-terminal de la grammaire est un nœud, son type sera l'un des types définis comme constantes. Dès qu'on crée un nœud, on va lui attacher le type qui le représente.
Exemple :

```
#define astProgram 500
#define astIntConst 485
```

On définit aussi une énumération pour les types des variables, les structures qu'on veut utiliser pour les tables de symboles ou l'arbre syntaxique, ainsi que des déclarations des fonctions utilisées dans tous les sous-programmes.

Chapitre 4. Analyse sémantique.

Actions sémantiques en `structfe.y`.

Avant de commencer les actions sémantiques, on établit le type possible d'un terminal et non-terminal. Pour cela, on construit l'union de YYSTYPE qui peut avoir 3 types : `int yint, char* ystr,` et `struct Ast_node* node.`

Le type ***int*** est réserve que pour le terminal CONSTANT et le type **char*** pour le nom du terminal IDENTIFIER. Tous les non-terminaux, ainsi que quelques autres variables rajoutées pour faciliter l'analyse sémantique sont de type **node** (un nœud de l'arbre abstrait syntaxique).

Chaque nœud contient les champs suivants :
`int node_type` - le type **astType** déclaré du façon explicit avec #define dans la biblioteque personelle.
`struct Symbol *symbol_node` – un symbole éventuellement rattaché à un nœud, utilisé pour faciliter l'analyse sémantique et la génération du code.
`struct Ast_node *child_node[4]` – au plus 4 nœuds-fils de type **node**, selon la dérivation.
`enum Type expression_type` – champ utilisé que pour la vérification du type-résultat d'une expression.
`char *result_var;` – variable utilisée pour la génération de code.

Déclarations

On commence l'analyse sémantique à partir des non-terminaux qui se dérivent directement dans des terminaux, pour faciliter l'analyse ascendante. Les plus importants pour nous sont *declarator* et *declaration_specifiers,* parce que cela va nous aider à *coller* les informations d'un type reconnu et d'un identifiant reconnu par la suite.

Analysons-nous la dérivation *declarator -> …*

Cas 1: `IDENTIFIER` – on a aucune information sur cet identifiant dans cette dérivation, donc on va créer un symbole qui porte son nom et on va lui donner le type UNDEFINED, qui va être change selon le type reconnu dans *declaration_specifiers*.
Cas 2: `'(' '*' IDENTIFIER ')'` – ici, notre grammaire force une déclaration d'un pointeur sur fonction. Donc, on connait déjà le fait que cet identifiant est de type POINTEUR_SUR_FONCTION.
Cas 3 : `'*' IDENTIFIER` – on sait juste que l'identifiant peut être un pointeur sur *declaration_specifiers* ou une fonction qui retourne un pointeur sur *declaration_specifiers*.
Cas 4: `'*' '(' '*' IDENTIFIER ')'` – ici l'identifiant est un POINTEUR_SUR_FONCTION qui va retourner un pointeur vers le type *declaration_specifiers.*

La seule dérivation qui contient les deux non-terminaux mentionnés au-dessus est *var_declaration*. Dès qu'on spécifie si un type est aussi externe ou non, on complète le type de l'identifiant dans la fonction:

```
void complete_type (struct Symbol *sym, struct Ast_node *child_node)
```

Il est important de préciser le fait que ce non-terminal est utilisé aussi pour une déclaration et une définition de fonction.
Comme vu dans la spécification de changement de la grammaire, une déclaration de fonction est une déclaration d'une variable qui est suivie par une liste des paramètres entre parenthèses. On change son (quasi)type en FONCTION ou POINTEUR_SUR_FONCTION, selon son (quasi)type antérieur.
De même façon, une définition d'une fonction est une déclaration de fonction qui n'est pas de type POINTEUR_SUR_FONCTION et qui est suivie par un block d'instructions.

Dès qu'on finit à conclure le type d'un identificateur, on est prêt pour le rajouter dans la table de symboles.

Spécification: Dans la suite, le mot *symbole* signifie une variable de type struct Symbol* qui représente un identificateur et qui continent des divers champs, initialisés selon le cas (pour une variable, fonction, structure).
Pour faciliter la gestion des symboles qui sont des champs des structures ou des paramètres d'une fonction, on utilise une pile des variables qui peut être manipulée à l'aide des fonction-utilitaires suivantes:

- `struct Symbol* popV()` – retourne le symbole du haut de la pile
- `void Empty_Stack(int nb_elements)` – vider la pile.(nb_elements est le nombre des paramètres de la fonction qui vient d'être déclarée ou le nombre des champs d'une structure qui a été définie)
- `void pushV(struct Symbol *p)`
- `void printVStack()`

S'il s'agit d'une définition d'une fonction, on rajoute l'identifiant dans la table de symboles et on actualise le nom de la fonction courante à l'aide d'une variable globale curFunctionSym de type `struct Symbol*` qui copie ce symbole et qui va nous servir à établir le nom de la fonction pour une variable qui probablement va être déclarée dans son bloc d'instructions.

Par contre, s'il s'agit d'une déclaration d'une variable, on rajoute le symbole dans la table de symboles, en spécifiant aussi la fonction dont il fait partie (donnée par le symbole curFunctionSym). En sachant le nom de la fonction courante, on peut facilement vérifier si le symbole courant se trouve déjà dans le bloc de la fonction courante (ou dans sa liste des paramètres) ou parmi les variables déclarées globalement. Pour une variable déclarée globalement, curFunctionSym=NULL.

→ Exception fait une déclaration dans une structure. Ici, tous les champs de la structure sont déjà empiles dans une pile des variables, après l'évaluation du non-terminal *struct_declaration_list*. Il nous reste qu'à copier la pile des variables dans la variable-champ *fields_list* du symbole. Dans ce cas, un symbole qui est un champ de la structure n'est pas inséré dans la table des symboles, mais dans la pile des champs de la structure courante. (Le même principe est utilisé pour les paramètres d'une fonction).

Pour la vérification de l'existence d'une variable qui vient de se déclarer ou une variable utilisée dans des diverses expressions, on fournit la fonction `struct Symbol *find_to_add(char *s)` qui retourne NULL si la variable n'existe pas déjà dans la même portée.

`struct Symbol *find_variable(char *s)` est tune fonction similaire mais qui retourne NULL dans le cas où la variable n'est pas accessible.

On a deux cas à gérer quand on vérifie l'existence d'une variable/fonction :

- On veut la déclarer. Dans ce cas, on regarde si elle existe déjà dans la fonction courante, dans les variables globales, ou si elle porte le nom de la fonction courante elle-même. Dans ces cas,
- On veut vérifier si elle existe et est *accessible* quand elle est utilisée pour des expressions et déclarée globalement, dans la liste de déclarations de la fonction elle-même, ou dans la liste des paramètre de la fonction.

Selon notre algorithme, on cherche d'abord de trouver le symbole dans la fonction courante, et donc donner une priorité aux variables qui peuvent être trouvées dans la fonction, au lieu de retourner un symbole qui signifie une variable globale dont on trouve la première mais à cause de qui on risque de déclarer plusieurs fois une variable avec le même nom dans une fonction.

Expressions

On a déjà mentionné des règles de priorité, d'associativité et le changement de la grammaire pour les expressions.
C'est important aussi de vérifier que deux opérandes sont *éligibles* pour une certaine opération.

`<  <=  >  >=`
Les expressions relationnelles sont autorisées qu'entre deux opérandes du même type entier ou du type pointeur (sur un même type).
Les opérations d'égalité , `==` et `!=` sont autorisées qu'entre deux opérandes du même type entier ou du type pointeur (sur un même type, étant donné qu'il n'y a pas de variables de type **void *** en STRUCIT).
`&&  ||`
Les expressions logiques sont autorisées entre n'importes quels types d'opérandes.

Expressions binaires

L'addition et la soustraction sont autorisées qu'entre deux opérandes **int** et **pointeur**, ou **int** et **int**. La multiplication et la division est autorisé qu'entre deux variables de type **int**.
On autorise les opérations entre pointeur et pointeur que si leurs types sont égaux ou ils sont **void ***.

Assignments

On autorise les assignements qu'entre deux variables du même type (ou entre un type **void*** et un pointeur).

Types des expression acceptées à gauche et à droite d'un assignement:
IDENT, ptr->champ, *p , &var.

Types des expression acceptées qu' à droite d'un assignement:
Sizeof(p) , CONSTANT, fonction(...), (expr)

(expr) - acceptée à gauche que si (expr) se matérialise en (*p) (où p est un pointeur sur **int**).

Pointeur sur une variable qui appartient à «une_structure»:
postfix_expression -> IDENTIFIER

Les possibilités sont :
p->champ //acceptée que si p est un pointeur sur «une_structure», où se trouve la variable *champs*
CONSTANT->champ //n'est pas acceptée
(expression)->champ
// n'est pas acceptée, même pas pour (*p)->champ ou (&p)->champ

postfix_expression(...)->champ // acceptée que si *postfix_expression* se dérive dans un nom d'une fonction ou un pointeur sur fonction qui retourne une type *pointeur sur une_structure*.
ptr->champ->champ // pointeurs multiples ne sont pas acceptés en STRUCIT

Appels de fonction (function designator)

ISO/IEC 2011, section 6.3.2.1 Lvalues, arrays, and function designators, paragraph 4

Un *désignateur de fonction* est une expression qui le type *fonction*. Sauf lorsqu'il s'agit de l'opérande de l'opérateur `sizeof` ou de l'opérateur unaire `&`, un désignateur de fonction de type « *type* de retour de fonction » est converti en une expression de type « pointeur vers le type de retour de fonction ».

```
int f(void);
int  *pf  = f;
int  *p  = &f; /* Les initialisations sont équivalentes */
```

En conclusion, `func()` `(*func)()` ou `(***func)()` sont équivalentes et acceptées en STRUCIT, ainsi que `(&func)()` , parce qu'il est équivalant à `func()` .
L'appel de fonction empile les paramètres de façon inverse.

Chapitre 5: Génération de code

Le fichier gen_code.c réalise la génération de code, en parcourant l'arbre abstrait syntaxique crée par l'analyseur sémantique dans le fichier structfe.y. Donc il va inclure aussi la bibliothèque definitions.h.
Dans ce fichier se trouve la fonction principale `int main (int argc, char* argv[])` qui attends trois arguments dans la ligne de commande afin de débuter la compilation suivie par la génération de code:

- argv[0] : L'exécutable obtenu après compiler les analyseurs LEX et YACC
- argv[1] : Le fichier .c qui contient le code en langage STRUCIT-frontend --> `yyin`
- argv[2] : Le fichier destiné à la génération du code cible --> `yyout`

Les fonctions de base dans ce programme sont :
`void generateCode(struct Ast_node *p, int level)` - génère le code pour le nœud courant
`void traverse(struct Ast_node* p, int n)` - traverse l'arbre syntaxique en affichant les nœuds.

On commence la génération de code par la fonction generateCode() qui a le nœud ast_root. Cette fonction appelle ses nœuds-fils par la même fonction generateCode().

En effet, cette fonction parcourt l'arbre de façon Depth-First-Search, en générant du code au fur à la mesure, facilitée par la boucle switch de generateCode() qui analyse le nœud courant et appelle la fonction chargée de la génération du code des fils du nœud courant, appelée `void processNoeud_Courant(struct Ast_node *p, int level)` pour un certain nœud `astNoeud_Courant`.
Il y a des exceptions ou on n'appelle pas la génération de code des nœud-fils, c'est dans le cas où on a l'information suffisante pour proprement générer du code.
Exemple : `void processVarDecl(struct Ast_node *p, int level)`

Les nœuds qui représentent les structures sont ignorées pour la génération de code. Leur usage était le suivant :

- les définir (dans ce cas on n'affiche plus leur code).
- déclarer une variable qui est pointeur sur la structure (dans ce cas la variable va avoir le type **void ***).
- d'accéder à un champ par un operateur '->' (on calcule le déplacement selon l'ordre des champs déclarées et leur type).
- pour la fonction SIZEOF (dans ce cas on calcule la taille de la structure pointée comme une somme de toutes les taille des types de ses champs).

La taille d'un champ d'une structure est 4 pour le type **INT** et 8 pour tout autre **pointeur**.

On utilise une pile des variables pour faciliter la génération des expressions. Dès qu'on trouve une variable ou constante, en général on la rajoute dans la pile (sauf si elle se trouve à gauche d'une expression de type assignement où il n'y a pas besoin d'une variable temporaire. On analyse ce cas dans les fonctions chargées de leur analyse.

Pour chaque nœud qui représente une expression, on génère le code de ses fils et en utilisant la pile on génère le code qui correspond à l'expression courante. La raison pour laquelle on utilise la pile este pour générer proprement des variables temporelles quand il y a besoin.

L'exception de l'empilement est quand on a l'un des assignements suivants :
A = B,

A - a, *a, &a, (*a)
B - b, CONSTANTE, (*b), (&b), *a, *b, function_call(a), (*****f)(...), (&f)(...), x op y (a , b – variables)
Dans ce cas, on est sûr qu'on n'a pas besoin des variables temporelles donc on ne rajoute rien dans la pile, et on affiche directement le code correspondant.

De plus, une expression comme (*****f)(...) et traduite en STRUCIT backend en f(). On ne génère pas de code redondant juste pour déréférencer une fonction en sachant qu'elle reste toujours un pointeur sur fonction.

Prenons un exemple où on a besoin de la pile des variables :

```
*p = 1+sizeof(q)*(1+2)
```

Cette expression correspond à l'arbre syntaxique suivant :

```
astAssignStmt
  |_astUnExpr
     |_astStar
     |_astStar
     |_astId
  |_astAdd
     |_astConst
     |_astMul
        |_astSize
        |_astAdd
           |_astConst
           |_astConst
```

On génère le code de fils gauche de astAssignStmt : *p (on garde son résultat qui est *p).

On génère le code de fils droite : `1+sizeof(q)*(1+2)`
D'abord, chaque constante représentée par le nœud **astConst** est empilée sur la pile des variables (dans la fonction `processIntConst`, appelée par `processAdd` dans notre cas). Donc, dans la fonction `processAdd`, après la génération du code de ses fils, les deux constantes se trouve déjà dans la pile.

On a « généré » le code de **astAdd**, maintenant on sait qu'on a besoin d'une variable temporelle. On crée une variable temporaires et on affiche le code correspond a l'expression de type code a trois adresse courante (dès qu'on crée une variable temporaire, on la rajoute dans la liste des variables temporaires de la fonction courante par la fonction `void add_temp(struct Symbol *f, char *name, enum Type type)`). On répète ce principe. À la fin, _tz = _tx+_ty.

Conditions et itérations

Les conditions if sont représentées par des étiquetés qui peuvent porter le nom *Letiq*, *Lelse*, *LFor*, *LWhile* suivi par un nombre.

`IF`
Le model d'une condition IF est similaire avec celui présenté dans le fichier backend :

1. `« code de la condition »`
2. `if (!condition) goto LelseN ;`
3. `{corps de IF}`
4. `LelseN :` … (les instructions suivants)

Un schelet d'un IF-ELSE est le même, sauf pour le point 4, ou on a :
4. `LesleN :`
5. `{corps de ELSE}`

Pour le point 2., dans le cas où la condition est sous forme `x op z` (ici op n'est pas l'assignement ou operateur logique) ou `x` ou `op x`, elle est écrite en changeant l'opérateur avec le contraire Ainsi, `x>y` devient `x<=y`.

Cela s'applique aussi pour les boucles for et while.
`WHILE`
Le model d'une boucle WHILE est le suivant :

1. `goto LWhileN ;`
2. `LWhileN : if ( !(condition) ) goto LelseM`
3. `{corps du WHILE}`
4. `goto LWhileN;`
5. `LelseM :` …

FOR

Le model d'une boucle FOR est le suivant :

1. <<expression code >>
2. Goto LtestN ;
3. LForM :
4. << corps du FOR >>
5. LtestN : if (expr) goto LforM;
6. … (les instructions suivants)

Difficultés rencontrées pendant la génération du code :

- Parfois, le fichier sortie n'arrive pas à écrire la 2ème étape du génération du code (c.à.d. déclaration des variables temporaires au début d'une fonction. Par exemple, pour le programme cond.c, listes.c ou pointeur.c les variables sont déclarées proprement, mais pour le programme main.c souvent les déclarations ne sont pas rajoutées. En effet, on observe que si la fonction main dépasse environ 160 lignes du code, la 2ème étape n'est pas faite (fermeture du fichier sortie, recommencement de l'écriture dans le même fichier). Notre fichier main.c a environ 210 lignes du code.

  Concernant ce problème, on aurait pu utiliser un autre algorithme d'ajouter les variables temporaires qui est plus *naïf* et qu'on peut détailler si besoin.

Chapitre 6. Liberation de la mémoire.

On utilise l'outil **Valgrind** afin de vérifier la libération de la mémoire allouée. Dès qu'on compile avec succès et on exécute les programmes argv[0], argv[1] et argv[2] comme détaillé, on peut exécuter des commandes spécifiques de Valgrind afin de visualiser -plus au moins en détail- la gestion de la mémoire pour les parties (blocks) des programmes du projet « accessible » par cet outil.

```
valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./[exec-name] test.c
```

Observation empirique: Il est important d'exécuter la commande Valgrind avec l'exécutable suivi par le nom du fichier test du langage de départ. Sinon, cet outil ne va pas accéder aux allocations mémoire dans le fichier yacc ou lex, même si c'est dans la partie main que la fonction yyparse() est appelée.

Observation 2: Pour un programme *simple* (c.à.d. pas d'allocation de mémoire faite par le programmateur), l'outil trouve environ 17 482 octets alloués intrinsèquement dans des programmes adjacents, comme y.tab.c ou lex.yy.c. D'ailleurs, ces octets alloués sont de type *encore accessible* donc ils ne présentent pas notre préoccupation.
Dans le cas où nous effectuons une allocation mémoire sauf libération, le nombre d'octets va apparaitre dans la ligne *definitely lost*.

Les fonctions chargees de la liberation mémoire sont `void freeSymTable()` et `void freeAST(struct Ast_node *p)`.

```
==5867== HEAP SUMMARY:
==5867==     in use at exit: 17,482 bytes in 4 blocks
==5867==   total heap usage: 4 allocs, 0 frees, 17,482
bytes allocated
```

Octets alloues pas Lex et Yacc.

```
==5867== LEAK SUMMARY:
==5867==    definitely lost: 0 bytes in 0 blocks
==5867==    indirectly lost: 0 bytes in 0 blocks
==5867==      possibly lost: 0 bytes in 0 blocks
==5867==    still reachable: 17,482 bytes in 4 blocks
==5867==         suppressed: 0 bytes in 0 blocks
```

Pour notre programme main.c, l'outil trouve environ 26 000 octets perdus définitivement.

Peurs et inquiétudes: Segmentation fault.

References:

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman-Compilers - Principles, Techniques, and Tools - Pearson Addison Wesley (2006)

https://www.ibm.com/docs/en/xl-c-and-cpp-aix/13.1.2?topic=operators-lvalues-rvalues

https://valgrind.org/

https://www.javatpoint.com/structure-padding-in-c

http://c-faq.com/decl/spiral.anderson.html

https://norasandler.com/2018/04/10/Write-a-Compiler-8.html

https://docs.microsoft.com/en-us/cpp/c-language/indirection-and-address-of-operators?view=msvc-170