STRUCIT, mini Compilateur C

Dima Elisabeta Iulia | Diane Horly

SLUIN602: Compilation

Étudiants

Dima Elisabeta Iulia

Diane Horly

Professeur

Pr. Sid Touati

Licence 3 - Informatique

A picture containing text

Description automatically generated

Chapitre 1 - Analyse lexicale. L’outil Lex

Chapitre 2 - Analyse syntaxique. L’outil YACC

Conflits shift/reduce et associativité

Changement de la grammaire initiale

Chapitre 3 – Bibliothèque personnelle, definitions.h

Chapitre 4 – Analyse sémantique dans YACC.

Déclarations.

Évaluation des expressions.

Chapitre 5 – Génération de code.

Parcours de l’arbre abstrait syntaxique.

Chapitre 6 – Libération de la mémoire allouée. Valgrind.

Chapitre 1

Analyse lexicale. 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".

Le programme ANSI-C.l contient 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

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 le dernier s’il est bien fini.

**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 qur plusieurs lignes

Exemple des reconnaissances des lexèmes :

{L}({L}|{D})\* { 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 et 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).

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, ainsi que le fait que chacun intervient dans les dérivations de l’autre en créant 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).

Pour permettre quand même des opérations unaires, on a rajoute la dérivation suivante :

additive\_expression

: '-' multiplicative\_expression **%prec** UNARY\_MINUS

---------

Quelques exemples de code considéré erroné en compilant le programme avec le 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)

1. **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. (notre solution : changement de la grammaire).

1. **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. (notre solution: dès l’apparition d’une définition d’un fonction, on vérifie si elle est de type pointeur sur fonction. 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).

1. int a;

a = 10(2+3);

//C: **error: called object type 'int' is not a function or function pointer**

* cause : postfix\_expression -> postfix\_expression '(' ')' -> primary\_expression '(' ')' -> CONSTANT '(' ')' (une solution : actions sémantiques adéquates pour vérifier si la variable avant () n’est pas une constante.

Ou, étant donné que la grammaire initiale insiste par sa construction que le nom d’une fonction soit un IDENTIFICATEUR, on peut aussi chercher le nom de la fonction dans la table de méthodes, et dans ce cas c’est évident qu’on ne va pas la trouver. Mais cela implique une complexité algorithmique plus grande que la première et donc on ne s’intéresse pas à l’implémenter).

1. a(1+2)=b(2+3)=4;

// C : error: expression is not assignable

1. int a;

a=1=2=3;

//C: **error: expression is not assignable**

* expression -> unary\_expression ‘=’ expression -> unary\_expression ‘=’ unary\_expression ‘=’ expression -> … -> IDENTIFIER ‘=’ CONSTANT ‘=’ CONSTANT ‘=’ CONSTANT ( « est-ce que pour chaque production expression -> unary\_expression ‘=’ expression, le token à gauche n’est pas une valeur constante? »)

1. \*&\*&\*&id=-1+2;

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 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 facilement vérifiée par l’analyse sémantique.

((((Le fig. 1 et 2 montrent des schémas des dérivations possibles pour la grammaire initiale et celui changée concernant spécifiquement les fonctions.))))

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 declarations des variables ou définitions des fonctions (exemple : int ((((test(int a,int b)))));).

Cependant, elle acceptera évidemment quelques parties de code incorrectes mais qui vont être passées à l’analyse sémantiques.

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 une définition est une déclaration d’une variable suivi par une liste des paramètres entre parantheses.

En plus, une définition d’une fonction est une déclaration d’une 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.

**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 qui va contenir les définition des structures qu’on va utiliser, des constantes pour l’arbre syntaxique et des fonctions présentes en structfe.y et semantiques.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 represente.

Exemple :

#define astProgram 500

#define astIntConst 548

On définit aussi une énumération, une union et les structures qu’on utilise pour les tables de symboles, ainsi que pour la création de l’arbre syntaxique.

Chapitre 4

Analyse sémantique.

Actions sémantiques en structfe.y :

Avant de commencer les actions sémantiques, on etabli le type possible d’un terminal et non-terminal.Pou 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 reserve que pour le terminal CONSTANT et le type char\* que 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 node a les champs suivants :

**int** node\_type -  le type astType declare du facon explicit avec #define dans la biblioteque personelle

**struct** Symbol \*symbol\_node – un symbol ratache(ou non) representatif pour nœud, utilise pour faciliter l’analyse semantique et la generation du code.

**struct** Ast\_node \*child\_node[4]- au plus 4 nœuds-fils de type node, selon la derivation.

**enum** Type expression\_type – champ utilise que pour la verification du type-resultat d’une expression.

On commence l’analyse semantique a partir des non-terminaux qui se dérivent 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 derivation, donc on va créer un symbol qui porte son nom et on va lui donner le type UNDEFINED, qui va etre change selon le type déjà 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 que l’identifiant est peut etre un pointeur sur «type\_specifiers» ou une fonction qui retoune un pointeur sur un «type\_specifiers».

Cas 4: '\*' '(' '\*' IDENTIFIER ')' – ici l’identifiant est un POINTEUR\_SUR\_FONCTION qui va retourner un pointeur vers le type de «type\_specifiers».

La seule dérivation qui contient les deux non-terminaux mentionnés au-dessus est var\_declaration. On spécifie si un type est aussi externe, et on complète le type de l’identifiant.

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.

Specifications : dans la suite, le mot *symbole* signifie une variable de type struct Symbol\* qui représente un identificateur et qui continent des divers champs, initialises selon le cas (pour une variable ou une fonction).

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 a 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 simplement l’identifiant dans la table de symboles, en spécifiant le fait qu’il est une fonction, sa liste des paramètres, son type de retour, etc… . Dans ce cas, on actualise le nom de la fonction courtante a l’aide d’une variable globale curFunctionSym de type struct Symbol\* qui copie ce symbole et qui va nous servir a 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 (donne par curFunctionSym). En sachant le nom de la fonction courante, on peut facilement vérifier et signaler si le symbole courant se trouve déjà dans le bloc de la fonction courante (ou dans sa liste des paramètres) ou parmi les variables declarees 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’a copier la pile globale des variables dans la variable-champ fields\_list du symbole. Dans ce cas, le symbole n’est pas insère dans la tab des symboles, mais dans la pile des champs de la structure courante.

Gestion de la mémoire : malloc/free

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 détaille dans le Chapitre 1, 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: Cet outil ne peut pas analyser des allocations de la mémoire que si les fonctions qui les contiennent sont accessible à partir de la fonction main(). 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 l’une des auteurs du projet), 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*.

﻿==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

﻿==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

*References:*

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

[*https://valgrind.org/*](https://valgrind.org/)

[*https://www.javatpoint.com/structure-padding-in-c*](https://www.javatpoint.com/structure-padding-in-c)

[*http://c-faq.com/decl/spiral.anderson.html*](http://c-faq.com/decl/spiral.anderson.html)

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

[*https://stackoverflow.com/questions/12259101/why-is-the-size-of-a-function-in-c-always-1-byte*](https://stackoverflow.com/questions/12259101/why-is-the-size-of-a-function-in-c-always-1-byte)