Rapport séminaire traduction-compilation, 10-15 décembre 2012 par Laurent Broto

Messi Nguélé Thomas
Supervisé par Pr Maurice Tchuente, Pr Jean-François Méhaut

21 avril 2014

Résumé

Ce rapport présente tout ce qui a été fait pendant le séminaire traduction-compilation, séminaire organisé du 10 au 15 décembre 2012 par le LIRIMA et animé par M. Laurent Broto. L'objectif de ce séminaire était de permettre aux étudiants de master et de doctorat de s'imprégner des concepts de base de contruction d'un compilateur. Pour atteindre cet objectif, les étudiants ont d'abord construit un analyseur lexical sans outil, puis avec l'aide de l'outil flex. Ils ont ensuite construit un analyseur syntaxique sans outil, puis avec l'aide de l'outil de bison. Ils ont enfin modifier l'analyseur pour qu'il génère du code assembleur x86.

Table des matières

Chapitre 1

Introduction

Le séminaire faisant l'objet de ce rapport s'est déroulé à l'UY1 du 10 au 15 décembre 2012. Il a été organisé par le LIRIMA à l'intension des étudiants de thèse et de master pour que ceux-ci s'imprègnent des principes de base permettant de construire les compilateurs. Le séminaire s'est déroulé en deux partie : - cours/tp ; - projet avec une forte insistance sur la partie tp. Dans ce rapport, nous présentons les différents tp effectués en rappelant chaque fois que cela est nécessaire, les concepts théoriques les soutendant.

Dans ce chapitre introductif, nous présentons d'abord les objectifs du séminaire, nous présentons ensuite les outils installés pour le séminaire, et enfin, nous présentons les différents chapitres qui apparaîtrons dans ce rapport.

1.1 Objectif

L'objectif global du séminaire était de savoir construire un compilateur et savoir se servir de certains outils pour construire un compilateur. Cet objectif devait être atteint à travers les sous objectifs suivants :

1. savoir construire un analyseur lexical ;
2. se servir d'un outil pour faciliter cette construction ;
3. connaître les bases de l'analyseur syntaxique ;
4. se servir d'outils pour effectuer un analyse syntaxique ;
5. générer du code x86 à partir de cette analyse syntaxique.

Plus précisement, à la fin de ce séminaire, on devait avoir un compilateur pouvant générer du code assembleur x86 à partir des fichiers sources de la forme :

```
a = 0;
read b;
read c;
while(a < c)
do
  a = a + b;
  if(a!=2) then
    print a;
  fi
done
```

1.2 Présentation du dossier des TPs

Le dossier des TPs fourni avec ce rapport se nomme "sources". Dans ce dossier, il y a les dossiers tp1, tp2, tp3 qui correspondent aux différents tps faits pendant le séminaire. Le dossier tp1 contient les exercices exo1, exo2, exo3, exo4, exo5 et exo6. Le dossier tp2 contient les exercices exo7, exo8_9 (regroupement des exercices 8 à 9), exo10, exo11, exo12, exo13 et exo14. Le dossier tp3 quant à lui contient les exercices exo15, exo16, exo17, exo18, exo19, et exo20-23 (regroupement des exercices 20 à 23).

Lorsque nous donnerons les solutions des différents exercices, nous supposerons qu'on est dans le dossier "sources". Chaque dossier d'exercice comporte un fichier Makefile (lorsque l'exercice correspond à un programme c). Ainsi, si le lecteur modifie le code que nous avons écrit, il n'a qu'à retaper la commande make et la compilation est refaite.

1.3 Outils utilisés

Les outils utilisés dans ce séminaire sont :
- nasm : pour la compilation les programmes en assembleur ;
- flex : outil facilitant la construction d'analyseurs lexicaux ;
- bison : outil facilitant la construction d'analyseurs syntaxiques ;
- gcc-multilib : version portable de gcc (architectures 32 et 64 bit) ;
- kdevelop : pour l'écriture des programmes ;
- ubuntu 12.04 : comme système d'exploitation.

1.4 Plan du rapport

Dans la suite, nous présenterons d'abord l'analyse lexicale, ensuite, nous présenterons l'analyse syntaxique et enfin la géneration de code.

Chapitre 2

Analyse lexicale

Dans ce chapitre, nous présentons à travers le tp1 et une partie du tp2 (dont les fichiers sources se trouvent dans le dossier fourni avec ce rapport) :

- les concepts de base de construction d'un analyseur lexical : les automates finis, les expressions régulières ;
- un outil facilitant cette construction : flex.

2.1 Généralités

L'analyse lexicale est la première phase de la compilation. Dans le texte source qui se présente comme un flot de caractères, l'analyse lexicale reconnait des unités lexicales qui sont les mots avec lesquels les phrases sont formées et les présente à la phase suivante l'analyse syntaxique. Une unité lexicale est généralement représentée par un code conventionnel. La chaine de caractères correspondante est appelé lexème.

L'analyse lexicale doit reconnaître des mots composés de lettres issues d'un alphabet (exemple : les lettres de a à z, les chiffres de 0 à 9).

En français, l'analyse lexicale du flot : "le chat est bleu" donnera en sortie

```
"<déterminant><nom><verbe><adjectif>"
```

En c, l'analyse du flot :"int a, b ; a=1 ; b=a+1 ;" donnera

```
<type><var><sepvar><var><sepstatement>
<var><operateur><ctse><sepstatement>
<var><operateur><var><operateur><cste><sepstatement>
```

2.2 TP1 : exercice 1, exercice 2

Nous commençons avec deux exercices qui nous permettent de reconnaître un mot particulier dans un fichier.

2.2.1 exercice 1

Enoncé : Écrire un programme qui compte le nombre d'occurence de la chaine "mur" dans le fichier fourni

- il serait préférable de le faire en une seule passe.

Solution proposée : La solution se trouve dans le dossier exo1 du dossier tp1. Nous l'avons testée avec le fichier lire.txt du dossier tp1.

```
cd tp1/exo1
./exo1 ../lire.txt
```

2.2.2 exercice 2

Enoncé : Écrire sa généralisation (le mot dont les occurences doivent être compté est passé en ligne de commande).

Solution proposée : La solution se trouve dans le dossier exo2 du dossier tp1.

```
cd tp1/exo2
./exo2 ../lire.txt mur
```

Les deux exercices de tout à l'heure pouvaient plus facilement être traités avec l'aide des automates finis.

2.3 Rappels sur les automates finis

2.3.1 Automate Fini Déterministe (AFD)

Un AFD est un quintuplet $A = (\Sigma, Q, q_0, F, \delta)$ où
- Σ est ensemble fini de symboles (alphabet) ;
- Q est un ensemble fini d'états ;
- q_0 est l'état initial ;
- $F \subseteq Q$ est l'ensemble des états finaux ;
- $\delta : Q \times \Sigma \to Q$ est appelée fonction de transition, $\delta \subseteq Q \times \Sigma \times Q$.

Un langage est dit reconnaissable s'il existe un automate fini qui le reconnait.

Soit A et A' deux automates avec $L(A)$ et $L(A')$ les langages qu'ils reconnaissent respectivement. A et A' sont dits équivalents si $L(A) = L(A')$.

2.3.2 Automate Fini Non-déterministe (AFN)

Un AFN est un automate pour lequel il existe plusieurs transitions sortant d'un même état sur le même symbole. Formellement, un AFN est un quintuplet $A = (\Sigma, Q, I, F, \delta)$ où
- Σ est ensemble fini de symboles (alphabet) ;
- Q est un ensemble fini d'états ;
- I est l'état initial ;
- $F \subseteq Q$ est l'ensemble des états finaux ;
- $\delta : Q \times \Sigma \to 2^Q$ est appelée fonction de transition, $\delta \subseteq Q \times \Sigma \times 2^Q$.

Théorème Pour tout AFN A défini sur Σ, il existe un AFD A' équivalent à A. Si A a n états, alors A' a au plus 2^n états. A' est souvent appelé le déterminisé de A.

2.3.3 AFN à transition spontanée (ϵ-AFN)

Un Automate Fini Non-déterministe à transition spontanée (ϵ-AFN) est un quintuplet $A = (\Sigma, Q, I, F, \delta)$ où
- Σ est ensemble fini de symboles (alphabet) ;

- Q est un ensemble fini d'états ;
- I est l'état initial ;
- $F \subseteq Q$ est l'ensemble des états finaux ;
- $\delta : Q\text{x}\Sigma \cup \{\epsilon\} \rightarrow Q$ est appelée fonction de transition, $\delta \subseteq Q\text{x}\Sigma \cup \{\epsilon\}\text{x}Q$. Ici, il est possible de changer d'état sans consommer de symbole en empruntant une transition étiquetée par le mot vide (ϵ).

Théorème Pour tout $\epsilon - AFN\ A$, il existe un AFN A' équivalent à A.

2.4 TP1 : exercice 3, exercice 4

2.4.1 Exercice 3

Ici, il s'agit de se servir de la définition d'un automate pour refaire les exercices 1 et 2 ;

Enoncé : Re-écrire si ce n'est déjà fait les analyseurs précédents avec un automate.

Solution proposée : La solution se trouve dans le dossier exo3 du dossier tp1.

```
cd tp1/exo3
./exo3-1 ../lire.txt mur
./exo3-2 ../lire.txt mur
```

2.4.2 Exercice 4

Dans cet exercice, nous écrivons un premier analyseur lexical des expressions arithmétiques.

Enoncé : Écrire un analyseur qui permet de renvoyer les lexèmes d'une chaine arithmétique permettant de faire les additions et les multiplications.

Exemple : L'analyse 21+12*45 devra renvoyer :

```
<int:21><operateur:+><int:12><operateur:*><int:45>
```

Votre programme devra détecter les erreurs de syntaxe (1.2+12*45 → parse error)

Hypothèses :

- les entiers seront sur deux caractères ;
- les opérateurs seront + et *.

Solution proposée : Nous avons encore utilisé un automate pour faire cet exercice. La solution se trouve dans le dossier exo4 du dossier tp1. Dans cette solution, nous ne respectons pas la première hypothèse : les entiers ont une taille quelconque. Les expressions arithmétiques que nous utilisons pour tester notre solution sont saisie dans le fichier expression.txt. Le lecteur est invité à le modifier à sa guise.

Pour exécuter cette solution, nous saisissons les commandes suivantes :

```
cd tp1/exo4
./exo4 expression.txt
```

2.5 Les expressions régulières

2.5.1 Généralités

Une expression régulière est une chaine de caractères qui décrit un ensemble d'autres chaînes de caractères. Les expressions régulières sont écrites à partir des symboles suivants :

$\{\phi, x \in \Sigma, +, ., *, (,)\}$

Ainsi, l'ensemble $EReg(\Sigma)$ des expressions régulières définies sur Σ est l'ensemble des expressions définies par application des clauses inductives suivantes :

1. $x \in EReg(\Sigma), \forall x \in \Sigma$
2. $\phi \in EReg(\Sigma)$
3. si $\alpha, \beta \in EReg(\Sigma)$, alors $(\alpha + \beta) \in EReg(\Sigma)$
4. si $\alpha, \beta \in EReg(\Sigma)$, alors $(\alpha.\beta) \in EReg(\Sigma)$
5. si $\alpha \in EReg(\Sigma)$, alors $(\alpha^*) \in EReg(\Sigma)$

Une interprétation I d'une expression regulière ($I : EReg(\Sigma) \rightarrow P(\Sigma^*)$) est définie par les clauses suivantes :

1. $I(x) = x, \forall x \in \Sigma$
2. $I(\phi) = \phi$
3. $I(\alpha + \beta) = I(\alpha) + I(\beta)$
4. $I(\alpha.\beta) = I(\alpha).I(\beta)$
5. $I(\alpha^*) = (I(\alpha))^*$

$L \subseteq \Sigma^*$ est un **langage régulier** sur Σ si et seulement si il existe une expression régulière $\alpha \in EReg(\Sigma)$ telle que $L = I(\alpha)$. On dira que α est une expression du langage L.

Les propriétes de la somme, de la concaténation et de l'itération montrent que chaque langage régulier admet une infinité d'expressions régulières.

2.5.2 Expressions régulières sur linux

Sur linux, les expressions régulières sont définis dans le manuel. Pour y accéder, on saisit la commande :

```
man 7 regex
```

Le tableau suivant présente les opérateurs utilisés pour construire ces expressions régulières.

.	n'importe quel caractère	classique
*	répétition 0 ou n fois du caractère qui précède	classique
+	répétition 1 ou n fois du caractère qui précède	étendu
?	répétition 0 ou 1 fois du caractère qui précède	étendu
[ ]	classe de caractères	classique
[^]	tous les caractères qui ne sont pas après ^	classique
^	début de ligne	classique
$	fin de ligne	classique
|	alternative entre deux expressions	étendu
( )	délimitation d'une sous expression	étendu

Exemples

- .ac représente les mots de trois lettres qui se terminent par "ac".
- [a - b] correspond à n'importe quelle lettre minuscule (non-accentuée).
- [^ a - b] correspond à n'importe quel caractère qui n'est pas une lettre minuscule non-accentuée.
- [st]ac représente entre autres "sac" et "tac".
- [^ f]ac représente les mots de trois lettres qui se terminent par "ac" et ne commencent pas par "f".
- ^ [st]ac représente les mots "sac" et "tac" en début de ligne.
- [st]ac$ représente les mots "sac" et "tac" en fin de ligne.
- ^ trac$ représente le mot "trac" seul sur une ligne.

Théorème de Kleene Les expressions régulières et les automates définissent les mêmes langages.

2.6 TP1 : exercice 5, exercice 6

Dans ces deux exercices, on applique le théorème de Kleene. D'abord à l'exercice 5, on utilise une expression régulière avec la commande grep. Ensuite à l'exercice 6, on construit un automate qui fait de même.

2.6.1 Exercice 5

Enoncé : comptez le nombre de ligne qui commencent par "if" et qui se terminent par 'n' du fichier /etc/bash.bashrc Commande : grep -c -E '<regexp>' /etc/bash.bashrc

Solution proposée : La solution proposée se trouve dans le dossier exo5. Pour obtenir cette solution, nous avons simplement remplacé <regex> de la commande de l'énoncé par ^ if.*n$

Pour exécuter cette solution, nous saisissons les commandes suivantes :

```
cd tp1/exo5
./exo5.sh
```

2.6.2 Exercice 6

Ici, on simule la commande grep.

Enoncé : Écrivez un programme permettant de parser un texte à l'aide d'une regex simple passée en paramètre :

1. pour les regex de type : mur → c'est déjà fait :)
2. travaillez sur la reconnaissance de ^ if.*n$: vous construisez juste l'automate qui reconnait cette regex
3. puis sur la reconnaissance de regex de type [abc]*,vous construisez un automate générique permettant de reconnaître les regex de ce type.

Solution proposée : La solution proposée se trouve dans le dossier exo6. Nous nous sommes contentés de ne faire que le deuxième point de l'énoncé. Le troisième point et même le programme générale (qui simule la commande grep) sont laissés aux soins du lecteur.

Pour exécuter cette solution, nous saisissons les commandes suivantes :

```
cd tp1/exo6-1
./exo6-1 /etc/bash.bashrc
```

2.7 Analyseur lexical

2.7.1 Définition

Un analyseur lexical est un automate fini pour la réunion de toutes les regex définissant les lexèmes

- il doit décomposer un mot (le programme source) en une suite mots reconnus
- il doit lever les ambiguités
- il doit construire les lexèmes : les états finaux contiennent des actions.

Une ambiguité survient lorsque l'analyseur lexical ne sait pas quel mot reconnaitre. Par exemple, il peux y avoir ambiguité pour reconnaître les mots fun et funx définis respectivement par les regex fun et funx. Pour lever les ambiguités, on peut soit définir une priorité entre les lexèmes, soit décider de reconnaître le lexème le plus long.

2.7.2 Outils de génération des analyseurs lexicaux

Il existe plusieurs outils de construction des analyseurs lexicaux :

- lex : lexical analyser generator (pour le langage c),
- flex : fast lexical analyser generator (version évoluée de lex),
- jlex : lexical analyser generator for java (j est mis pour java),
- jflex : fast lexical analyser generator for java (j est mis pour java),
- ocamllex : générateur d'analyseur lexical pour le langage ocaml, ...

Dans ce séminaire, nous avons utilisé flex.

2.7.3 L'outil flex

Généralités

Un fichier flex peut être vu comme un fichier de description des regex. À chaque regex, on associe des actions. Ainsi, chaque fois qu'une regex est reconnue, l'action est exprimée. Flex génère un fichier c dont les fonctions peuvent être utilisées depuis un autre programme. Un fichier flex a pour extension .lex

Pour générer l'analyseur lexical, on saisit les commandes suivantes :

```
1. flex -o sortie.yy.c fichier-entree.lex
2. gcc -o sortie sortie.yy.c -lfl
```

Format d'un fichier flex

Un fichier .lex a trois sections (voir figure 2.1) :

1. définitions
2. regex
3. code additionnel

Ces sections sont séparées par %%

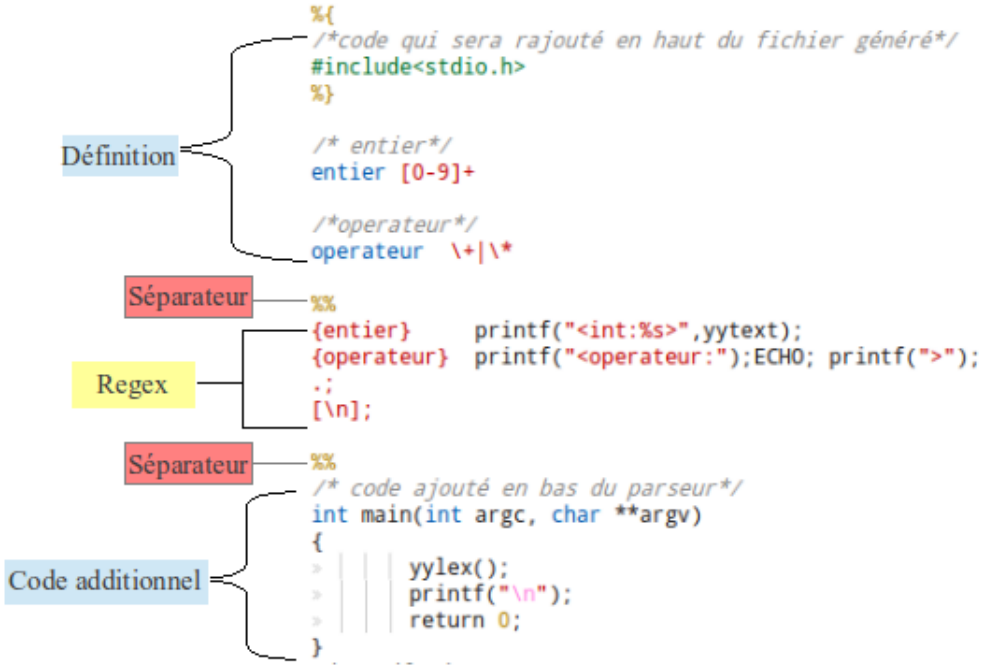


FIGURE 2.1 – Format d'un fichier flex

Section définition

La section définition
- permet d'inclure du code c ;
- permet de définir des alias pour les regex.

Exemple

```
%{
#include<stdio.h>
%}
entier [0-9]+
operateur  \+|\*
```

Section regex

La section regex
- permet de définir les regex détectées par l'automate ;
- permet d'associer du code à chaque regex.

Exemple

```
{entier}     printf("<int:%s>",yytext);
{operateur}  printf("<operateur:");ECHO; printf(">");
.;
[\n];
```

Dans cet exemple yytext et ECHO sont des variables :
- yytext est la valeur du lexème détecté ;
- ECHO est équivalent à printf("%s",yytext).

Section code

La section code permet de rajouter du code utilisateur. Elle est recopiée telle quelle à la fin du fichier.

Exemple

```
int main(int argc, char **argv)
{
yylex();
printf("\n");
return 0;
}
```

2.8 TP2 : exercice 7, exercice 8 et exercice 9

2.8.1 Exercice 7

Il s'agit ici d'un exemple d'application sur flex.

Enoncé : Reprenez avec flex l'exercice 4.

Solution proposée : Pour exécuter cette solution, nous saisissons les commandes suivantes :

```
cd tp2/exo7
echo "12+4+56" | ./exo7
```

2.8.2 Exercice 8_9

Nous avons jumelé les exercices 8 et 9. Dans ces exercices, nous programmons l'évalution des expressions de la gauche vers la droite.

Enoncé :

1. Évaluer les expressions de la gauche vers la droite sans aucune priorité.
2. Évaluer les expressions de la gauche vers la droite en respectant la priorité des opérateurs (* plus prioritaire que +) :
   (a) vous pourrez utiliser l'algorithme de Shunting-yard pour passer d'une expression infixée vers une postfixée ;
   (b) vous évaluerez ensuite l'expression postfixée.

   Hypothèses : le nombre d'opérateurs sera limité à 5 et le nombre d'opérandes à 6.

Solution proposée : La solution proposée se trouve dans le fichier exo8_9.c du dossier exo8_9. Pour obtenir cette solution, nous avons utilisé les structures file et pile définies dans les fichiers utilitaire.c et utilitaire.h. La procédure *exp_post_fixe_sans_prio()* resoud le premier point de l'énoncé. La procédure *exp_post_fixe_avec_prio()* quant à elle implémente l'algorithme de Shunting-yard. Nous n'avons pas respecté les hypothèses car les structures pile et file sont implémentées avec les listes chainées.

Pour exécuter cette solution, nous saisissons les commandes suivantes :

```
cd tp1/exo8_9
./exo8_9 ../../tp1/exo4/expression.txt
```

2.9 Synthèse de l'analyse lexicale

En définitive, les regex sont à l'origine de l'analyse lexicale. À partir de ces regex, des automates sont construits. Ces automates permettent de reconnaître les regex. Ce travail est grandement facilité grâce à des outils comme flex.

L'analyse lexicale apparaît pour l'analyse syntaxique comme une sorte de fonction de lecture améliorée qui fournit un mot lors de chaque appel. Dans le prochain chapitre nous reverons ce qui a été fait pendant le séminaire concernant l'analyse syntaxique.

Chapitre 3

Analyse syntaxique

Dans ce chapitre, nous présentons à travers la suite du tp2 et une partie du tp3 (dont les fichiers sources se trouvent dans le dossier fourni avec ce rapport) :

- les concepts de base de construction d'un analyseur syntaxique : quelques concepts grammaires ;
- un outil facilitant cette construction : bison.

3.1 Généralités

L'analyse syntaxique est la deuxième phase de la compilation. Cette phase a pour but de reconnaître les phrases du langage : vérifier que **l'assemblage** des mots provenant de l'analyse lexical est **syntaxiquement correct**.

En entrée, nous avons :

- les lexèmes issus de l'analyseur lexical ;
- et une **grammaire** à respecter.

En sortie, nous avons :

- un arbre abstrait : la succession des règles utilisées pour construire le programme ;
- les erreurs de syntaxe avec leur localisation et decription le plus précis possible.

En d'autres termes, cette phase a pour rôle principal de dire si le texte source appartient au langage considéré, c'est à dire s'il est correct relativement à la grammaire de ce dernier.

Dans la section qui suivre, nous allons faire quelques rappels sur les grammaires.

3.2 Rappels sur les grammaires

3.2.1 Définition

Une grammaire est un quadruplet $G = (\Sigma, V, S, R)$ où

- Σ est un alphabet fini(de symboles terminaux) ;
- V est l'ensemble des symboles non terminaux ;
- S est l'axiome ;
- R est l'ensemble des règles $\alpha \to \beta$ avec $\alpha \in (\Sigma \cup V)^+$ et $\beta \in (\Sigma \cup V)^*$.

3.2.2 Hierarchie de CHOMSKY

D'après l'hierarchie de CHOMSKY, les grammaires sont classées en quatre types :

Type	Description	Machine de reconnaissance
0	Pas de restriction sur les règles $\alpha \in (\Sigma \cup V)^+, \beta \in (\Sigma \cup V)^*$	Machine de Turing
1	$S \to \epsilon$, les autres règles sont de la forme $\alpha \to \beta$, $\|\alpha\| \leqslant \|\beta\|$	Machine de Turing à mémoire linéairement bornée
2	Les règles sont de la forme : $A \to \gamma$ $A \in V$ et $\gamma \in (\Sigma \cup V)^*$	Automate à pile
3	$A \to \epsilon, A \to a, A \to aB$	Automate fini

3.2.3 Les avantages des grammaires

La syntaxe de construction d'un langage de programmation peut être décrite par les grammaires de type 2. Ces grammaires sont aussi appelées les grammaires non contextuelles. Elles offrent des avantages significatifs à la fois aux concepteurs des langages et aux écrivains des compilateurs :

- Une grammaire donne une spécification syntaxique précise et cependant facile à comprendre d'un langage de compilation.
- Pour certaines classes de grammaire, il est possible de construire un analyseur syntaxique efficace qui détermine si un programme source est syntaxiquement bien formé. De plus, le processsus de construction des analyseurs syntaxiques peut relever des ambiguités syntaxiques et d'autres constructions difficiles à analyser qui pourraient, sinon, rester non détectées lors de la construction d'un langage et son compilateur.
- Une grammaire conçue proprement impose au langage de programmation une structure qui est utile pour la traduction du programme source en du code objet correcte et pour la détection d'erreures.
- Les langages évoluent au cours du temps acquérant de nouvelles constructions et effectuant des tâches additionnelles. Ces constructions nouvelles peuvent être ajoutées à un langage plus facilement quand il en existe une implémentation fondée sur une description grammaticale du langage.

3.2.4 Les méthodes d'analyse syntaxique

Il existe trois méthodes d'analyse syntaxique :

1. Les méthodes universelles comme celles de Cocke-Younger-Kasami et d'Early qui permettent d'analyser une grammaire quelconque. Cependant ces méthodes sont trop inefficaces pour être utilisées dans les compilateurs industriels.
2. Les méthodes ascendantes : on part du mot pour retrouver l'axiome.
3. Les méthodes descendantes : ici, on part de l'axiome pour déduire le mot.

Les méthodes descendantes ou ascendantes les plus efficaces travaillent uniquement sur les sous classes de grammaire, mais plusieurs de ces sous classes comme les grammaires LL et LR sont suffisamment expressives pour décrire la plupart des constructions syntaxiques des langages de programmation.

3.2.5 Dérivation

Considérons la grammaire de la figure 3.1. La phrase $int + int * int$ peut se dériver de deux manières différentes (voir figure).

1 E→E+E
2 E→E*E
3 E → int

FIGURE 3.1 – Grammaire des expressions arithmetiques

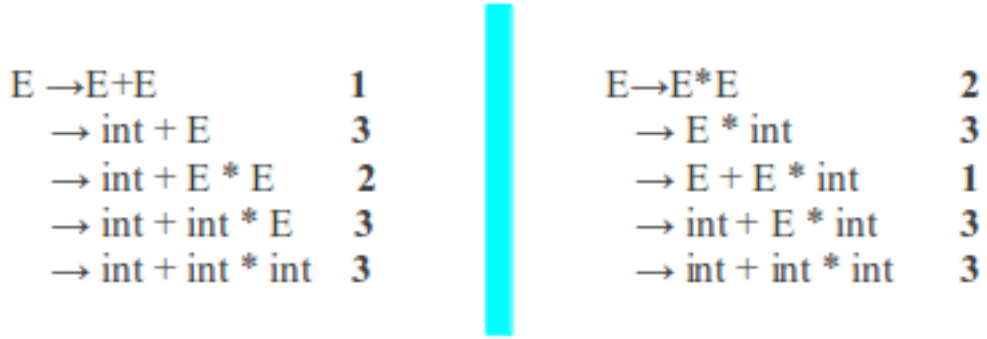


FIGURE 3.2 – dérivation

3.2.6 Arbre d'analyse

Un arbre d'analyse peut être considéré comme une représentation graphique d'une dérivation dans laquelle les choix concernant l'ordre de remplacement ont disparus. La figure donne les arbres d'analyse

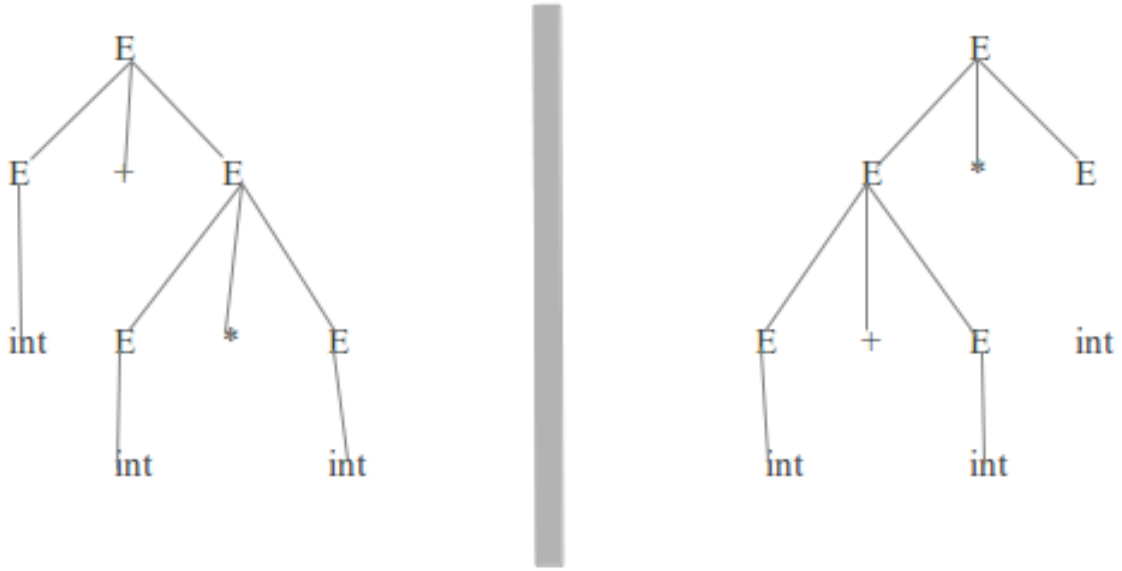


FIGURE 3.3 – Arbres d'analyse de la phrase $int + int * int$

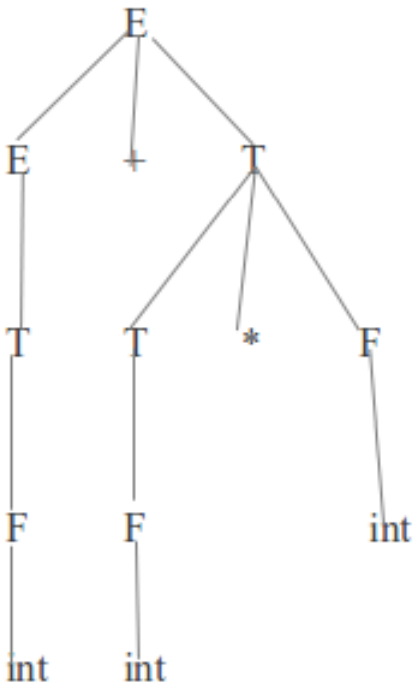
3.2.7 Grammaire ambiguë

Une grammaire qui produit plus d'un arbre d'analyse pour une phrase donnée est dite ambiguë.

La grammaire de la figure 3.1 est ambiguë car la phrase $int + int * int$ admet deux arbres d'analyse (confère figure 3.3). Mais, on peut la rendre non ambiguë voir figure 3.4. La phrase $int + int * int$ a à présent une unique dérivation et un unique arbre d'analyse (voir figure 3.5).

1 E→E+T
2 E→T
3 T→T*F
4 T→F
5 F → int

FIGURE 3.4 – Grammaire des expressions arithmetiques non ambigue



E → E + T **1**
→ T + T **2**
→ F + T **4**
→ int + T **5**
→ int + T * F **3**
→ int + F * F **4**
→ int + int * F **5**
→ int + int * int **5**

La multilplication est prioritaire. On a renforcé l'associativité à gauche.

FIGURE 3.5 – nouvelle dérivation et nouveau arbre de $int + int * int$

L'ambiguité des grammaires n'est pas un problème décidable. Il existe des classes de grammaire non ambiguë : LL(0), LR(0), LR(1), SLR(1), LALR(1), ... On sait construire un automate à pile correspondant pour :

- analyse descendante LL(0) : top-down on part de l'axiome pour retrouver l'arbre de dérivation.
- analyse ascendante LR(0), LR(1), LALR(1), SLR(1), ... : bottom-up on part du mot pour retrouver l'arbre de dérivation.

Lors du séminaire, nous nous sommes concentrés sur l'analyse ascendante $LR(0)$. Dans la suite de ce rapport, nous présenterons des exercices sur la connaissance de l'analyse ascendante $LR(0)$. Nous invitons donc le lecteur à revoir la documentation concernant l'analyse $LR(0)$.

3.3 TP2 : exercice 11, exercice 12, exercice 13 et exercice 14

Les exercices 11, 12, 13 permettent de reviser la construction des tables $LR(0)$ et l'algorithme de reconnaissance d'un mot lorsque la table est déjà construite. L'exercice 14 quant à lui permet

de programmer un automate à pile pour la reconnaissance d'un mot (une sorte de construction manuelle d' un analyseur syntaxique).

3.3.1 Exercice 11

Enoncé : Construire la table de la grammaire :

1. $E \rightarrow Ac$
2. $A \rightarrow AaAb$
3. $A \rightarrow d$

À l'aide de la table construite, analysez le mot dcadcb.

Solution proposée : La solution se trouve dans le dossier exo11 du dossier tp2. Le fichier tableGrammaireExo11.pdf contient la table d'analyse et le fichier exo11.txt contient la reconnaissance du mot dcadcb.

3.3.2 Exercice 12

Enoncé : Construisez la table action/goto pour la grammaire de la figure 3.1.

Solution proposée : La solution se trouve dans le fichier exo12.pdf contenu dans le dossier exo12 du dossier tp2.

3.3.3 Exercice 13

Enoncé : Avec la table de l'exercice 12 , analysez le mot x + y * z
- en privilégiant le shift puis le reduce,
- qu'est ce que ça change sur le mot reconnu ?

Solution proposée : La solution se trouve dans le fichier exo13.pdf contenu dans le dossier exo13 du dossier tp2.

3.3.4 Exercice 14

Enoncé Construisez un automate à pile permettant d'analyser les mots de la grammaire de la figure 3.1. Cet automate prendra en entrée des lexèmes renvoyés par flex et sera inclus dans sa partie code.

Solution proposée : La solution proposée se trouve dans le dossier exo14 du dossier tp2. Pour exécuter cette solution nous saisissons les commandes :

```
cd tp2/exo14
echo 12*12+14$ | ./exo14
```

3.4 Générateur d'analyseur syntaxique

Il existe plusieurs générateurs d'analyseurs syntaxiques :
- yacc,
- bison,
- ocamlyacc,
- cups,
- menhir,
- ...

Pendant le séminaire, nous avons utilisé bison.

3.4.1 L'outil bison

Généralités

- L'outil bison prend en entrées
  - les lexèmes renvoyés par flex,
  - un fichier de règles de grammaire semblable celui de flex.
- L'outil bison génère en sortie un programme c qui effectue l'analyse ascendante et lance des actions sémantiques.
- L'outil bison résoud les conflits de la manière suivante (par défaut) :
  - shift/reduce : priorité au shift,
  - reduce/reduce : ordre d'appartion des règles
- La génération de l'anlyseur syntaxique se fait avec les commandes suivantes :
  1. `bison --defines=simple.h -v -o s-bison.tab.c s-bison.y`
  2. `flex -o s-flex.yy.c s-flex.lex`
  3. `gcc s-flex.yy.c s-bison.tab.c -o simple -lfl`

Format d'un fichier bison

Un fichier bison (d'extension .y) a aussi trois sections (voir figure 3.6) comme un fichier flex :

1. section définition
2. section règle
3. section code additionnel

Ces sections sont séparées par %%

Section définition

La section définition
- permet de déclarer les fonctions issues de flex ;
- permet de déclarer les token : nom des lexèmes renvoyés par flex.

Exemple

```
%{
#include <stdio.h>
#include "simple.h"
#define nbMax
```

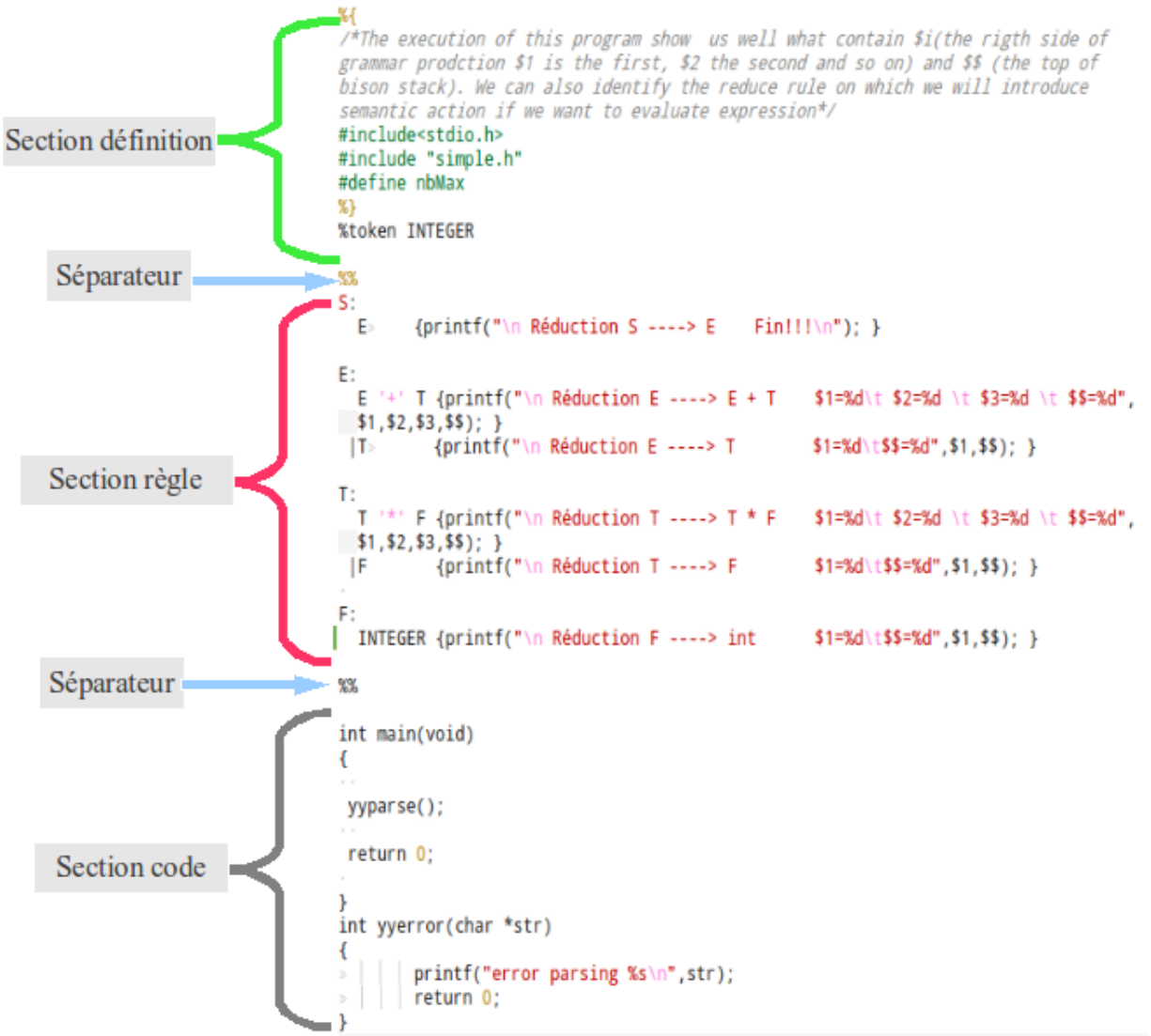


FIGURE 3.6 – Format d'un fichier bison

```
/* les fonctions issues de flex */
int yylex(void); /* un peu comme getNextToken */
void yyerror(const char *);
%}
/* les symboles terminaux */
%token INTEGER
```

Section règle

La section règle
- permet de définir la grammaire ;
- permet d'associer des actions sémantiques à certaines règles de la grammaire.

Exemple

```
S:
  E {printf("\n Réduction S ----> E    Fin!!!\n"); }
```

```
E:
  E '+' T {printf("\n Réduction E ----> E + T    "); }
|T   {printf("\n Réduction E ----> T        $1=%d\t$$=%d",$1,$$); }

T:
  T '*' F {printf("\n Réduction T ----> T * F    "); }
|F       {printf("\n Réduction T ----> F        $1=%d\t$$=%d",$1,$$); }

F:
  INTEGER {printf("\n Réduction F ----> int      $1=%d\t$$=%d",$1,$$); }
```

Dans cet exemple, on voit deux types de variables : $$ et les $i ($1, $2, $3)
- $$ représente la pile ;
- les $i représentent les valeurs des parties droites des règles.

Section code

La section code permet de rajouter du code utilisateur. Dans cette section on écrit le code de certaines fonctions déclarées dans la section définition. On peut également lancer l'analyse.

Exemple

```
int main(void)
{
  yyparse();
  return 0;
}
int yyerror(char *str)
{
  printf("error parsing %s\n",str);
  return 0;
}
```

Nous pouvons aussi utiliser les variables comme yyin ou yyout qui sont du type FILE*.

3.4.2 Modifications conséquentes dans le fichier flex

- Dans la section définition, le fichier .h permet d'échanger des lexèmes entre flex et bison. Il est généré automatiquement par bison.
  ```
  %{
    #include<stdio.h>
    #include "simple.h"
  %}
  ```

- Dans la section regex, yyval permet à bison de retourner la valeur du lexème lu.
  ```
  {entier} { yylval=atoi(yytext);return INTEGER;}
  {operateur} {return *yytext;}
  ```

- La section code sera souvent vide (on utilisera celle de bison).

3.5 TP3 : exercice 15, exercice 16, exercice 17, exercice 18

3.5.1 Exercice 15

Enoncé Construisez l'analyseur syntaxique permettant de suivre les actions de l'automate pour l'évaluation des expressions arithmétiques

- vous définirez la grammaire dans le fichier de règles de bison (celle utilisée à la figure 3.4).
- vous effectuerez des printf des variables intéressantes lors des réductions.

Solution proposée : La solution proposée se trouve dans le dossier exo15 du dossier tp3. Pour exécuter cette solution nous saisissons les commandes :

```
cd tp3/exo15
cat source.txt | ./exo15
```

3.5.2 Exercice 16

Enoncé Examinez la sortie de l'exercice 15 et modifiez votre analyseur pour qu'il évalue des expressions arithmétiques.

Solution proposée : La solution proposée se trouve dans le dossier exo16 du dossier tp3. Pour exécuter cette solution nous saisissons les commandes :

```
cd tp3/exo16
cat ../exo15/source.txt  | ./exo16
```

3.5.3 Exercice 17

Enoncé Modifiez votre grammaire pour pouvoir écrire une séquence d'instructions telle que celle-ci :

```
a=3; b=4+3*a;
print a; print b;
```

où A,B,C,D seront 4 variables prédéfinies

Solution proposée : La solution proposée se trouve dans le dossier exo17 du dossier tp3. Pour exécuter cette solution nous saisissons les commandes :

```
cd tp3/exo17
cat commande.mes | ./exo17
```

3.5.4 Exercice 18 : interpreteur

Enoncé Modifiez le fichier de règles de bison pour interprêter les programmes issus de votre grammaire.

Solution proposée : La solution proposée se trouve dans le dossier exo18 du dossier tp3. Pour exécuter cette solution nous saisissons les commandes :

```
cd tp3/exo18
cat commande.mes | ./exo18
```

3.6 Synthèse sur l'analyse syntaxique

Dans ce chapitre, nous avons vu que la tâche principale de l'analyse syntaxique est **de vérifier que l'assemblage des mots** (constituant le programme source) **provenant de l'analyse lexicale est synaxiquement correcte**. Le programme source est écrit en respetant une grammaire de type 2 (une grammaire hors contexte). Les analyseurs sont réalisés en programmant les automates à pile. Cette tâche est grandement simplifiée par des outils comme bison (qui travaille en collaboration avec flex).

Pendant le séminaire, après l'analyse syntaxique, nous sommes directement passé à la phase de génération du code cible. Le chapitre qui suivra détaillera ce qui a été fait.

Chapitre 4

Génération de code

Dans ce chapitre, nous présentons à travers le reste du tp3 la dernière étape de la compilation étudiée lors du séminaire : la génération de code assembleur x86.

4.1 Généralités

- Les compilateurs utilisent généralement un langage intermédiaire :
  - pour travailler sur la structure du code,
  - pour optimiser le code
  - pour faciliter le portage vers des langages cibles différents
- Le code sera généré dans les actions sémantiques.
- Dans ce séminaire, nous avons choisi l'assembleur x86 comme langage cible : nous simulerons une machine à pile.
- Règle générale :
  - à chaque réduction nous effectuerons les actions requises,
  - les opérandes seront issues du sommet de la pile,
  - le résultat sera empilé.

Dans la suite, nous ferons une séquence d'exercices menant à la génération de code assembleur x86. Le lecteur devra reviser le cours sur l'assembleur pour être à l'aise dans cette partie.

4.2 TP3 : exercice 19, exercice 20-23

4.2.1 Exercice 19

Enoncé Assemblez le fichier hello-world.asm, exécutez le

```
1 nasm -f elf -o hello-world.o hello-world.asm
2 ld -s -o hello-world hello-world.o -melf_i386 -I/lib/ld-linux.so.2 -lc
```

Solution proposée La solution proposée se trouve dans le dossier exo19 du dossier tp3. On lance le hello world avec les commandes suivantes :

```
cd tp3/exo19
./hello-world
```

4.2.2 Exercice 20-23

Enoncé

Exercice 20 : En vous aidant de la réponse à l'exercice 15, écrivez manuellement les instructions assembleur permettant de réaliser l'évaluation de l'expression arithmétique 1+4*5

Exercice 21 : Modifiez les actions sémantiques du fichier de règle de bison pour générer automatiquement les instructions assembleur.

Exercice 22 : Modifiez les actions sémantiques du fichier de règle bison pour générer automatiquement les instructions assembleur correspondant au langage définit à l'exercice 17.

Exercice 23 : • La séquence assembleur suivante permet de lire un chiffre au clavier :

```
mov eax, 3 ;sys_read(3)
mov ebx, 0 ;stdin (0)
mov ecx, car ;string to store to
mov edx, 1 ;number of bytes to read
int 0x80h ;interrupt for linux
```

Modifiez la grammaire, le fichier de règles de bison et les actions sémantiques pour pouvoir compiler en assembleur des séquences de code telle que :

```
read a;
b=4+3*a;
print b;
```

- Faites les modifications nécessaires pour pouvoir compiler en assembleur des séquences de code telle que :

```
c = (2==3);
d = (2<3);
a = (2!=3);
b = (2>3);
print a;
print b;
print c;
print d;
```

Les variables a,b,c,d devraient contenir respectivement 1,0,0,1.

- Faites les modifications nécessaires pour pouvoir compiler en assembleur des séquences de code telle que :

```
read a
if(a!=2) then
  print a;
fi
```

- Faites les modifications nécessaires pour pouvoir compiler en assembleur des séquences de code telle que :

```
a = 0;
read b;
read c;
while(a < c)
do
  a = a + b;

  if(a!=2) then
    print a;
  fi
done
```

Solution proposée Nous avons jumelé les solutions de ces exercices en une solution qui se trouve dans le dossier exo20-23 du dossier tp3. Pour exécuter cette solution, on execute les commandes suivantes :

```
cd tp3/exo20-23
./compilateur.sh
./test
```

La commande ./compilateur.sh compile notre programme (commande.mes) et génère l'exécutable test.

Chapitre 5

Conclusion

Dans ce dernier chapitre, nous présentons le bilan du séminaire ainsi que l'apport de ce séminaire par rapport à nos lectures sur les dsl (domaine specific language) ou langages dédiés.

5.1 Bilan du séminaire

Ce séminaire nous a permis de reviser par la pratique (presque tout a été codé en code c) les concepts de base d'un compilateur. Les objectifs du séminaire ont donc été atteints :

1. nous avons construit à la main (sans l'aide d'un outil) un analyseur lexical (voir TP1 : exercice 4) ;
2. nous nous sommes servis de l'outil flex pour faciliter cette construction (voir TP2 : exercice 7) ;
3. nous avons revisé les concepts de bases de l'analyse syntaxique (les grammaires, la construction d'un automate à pile au TP2 : exercice 14) ;
4. nous nous sommes servis de l'outil bison pour effectuer un analyse syntaxique ;
5. nous avons généré du code x86 à partir de cette analyse syntaxique (voir TP3 :exo20-23).

5.2 Apport sur les lectures effectuées

Ce séminaire a renforcé l'idée qui est souvent exposée sur les dsl (langages dédiés) : un dsl fourni un code qui s'exécute de façon optimal pour le domaine pour lequel ce dsl existe. En effet, si l'on considère le langage développé pendant le séminaire. Nous avons comparé les codes assembleurs générés d'un côté par le compilateur c, de l'autre côté par notre compilateur. Nous avons vu que notre code générait beaucoup moins de lignes assembleurs que le code généré par gcc. Ce qui est bien justifié car notre compilateur ne fait que l'essentiel correspondant à notre langage. Alors que gcc génère bien d'autres choses allant au delà de notre langage.

Les dsl que nous avons étudiés jusqu'ici s'appuient essentiellement sur l'optimisation de code et la représentation intermédiaire qui malheureusement ont été abordée de manière très sommaire dans ce séminaire.

Toute fois nous pouvons dire que nous nous sommes imprégnés des techniques de base pour la construction des compilateurs. Ce qui nous sera grandement bénéfique lors de la réalisation de notre dsl pour la simulation des systèmes complexes.