université de BORDEAUX

Analyse Syntaxique et Projet de Programmation

Jimmy GOURAUD, Yordan KIROV, Nicolas PALARD, Sébastien POUTEAU, Etienne ROBERT

30 avril 2016, Année 2015-2016

Sommaire

1 Introduction

Le but final de ce projet est d'implémenter un compilateur de site web en créant un langage de programmation qui nous permettra de décrire des pages web.

Pour ce faire, nous allons utiliser tout au long de ce projet **Lex/Yacc (Bison)** et le langage **C**.

Lex nous servira d'analyseur lexical pour reconnaître les mots-clés de notre langage de programmation.

Yacc (Bison), quant à lui, nous servira d'analyseur grammatical, nous permettant ainsi d'établir la grammaire de notre langage de programmation, ou autrement dit, il va nous permettre de déceler les expressions/formes valides de celles qui ne le sont pas.

Lex et **Yacc (Bison)** s'utilisent avec le langage **C**, celui-ci nous sera utile pour toutes les fonctions annexes permettant la liaison et l'interprétation de la grammaire générée.

Ce langage que nous créons doit être capable de manipuler deux types de données :

- Le type C : **int** appelé type numérique.
- Le type **document** qui sera représenté par des arbres, ou plutôt des forêts d'arbres.

2 L'analyse lexicale, le "lexeur".

La partie **analyse lexicale** du projet consiste, comme dit précédemment, à reconnaître les "mots-clés" de notre langage de programmation. Ces "mots-clés" sont appelés **lexèmes**. Une suite de lexèmes est appelé un mot, et ce mot sera traité plus tard par l'analyseur grammatical.

Nous avons réalisé **diverses implémentations** de notre lexeur.

Le problème principal de cette partie était de pouvoir gérer les espaces licites et illicites.

2.1 Implémentation de base

Notre idée de base a été de séparer l'analyse lexicale en plusieurs modes différents de façon à la simplifier :

- ◇ Un mode utilisé pour nous indiquer si nous étions dans un arbre dit **TREE MODE**
- ◇ Un mode utilisé à l'intérieur du **TREE MODE** permettant de récupérer le texte brut écrit par l'utilisateur dit **TEXT MODE**
- ◇ Un mode utilisé pour reconnaître les déclarations de variables/fonctions dit **LET MODE**
- ◇ Un mode spécial dit **EMIT MODE** qui sert uniquement pour l'appel de la fonction emit.
- ◇ Un mode **INITIAL** le mode de base de **Lex**.

Ainsi grâce à ces 4 modes nous pouvions, *via* des piles d'états, gérer des cas complexes, comme savoir de quel état nous venions et ainsi gérer "facilement" les différents cas et savoir quand autoriser les espaces.

Nous nous sommes vite rendu compte que certains états étaient inutiles et qu'avec un peu plus de réflexion nous pouvions réduire ce nombre. Nous sommes donc partis vers une autre forme de lexeur.

2.2 Implémentation après refonte

Cette implémentation ne contient qu'un seul état.

◇ **TEXT MODE**. Cet état reste **nécessaire** car n'importe quelle chaîne de caractère est autorisée (si elle est correctement mise entre " " et que les caractères pouvant contredire cette affirmation sont échappés). Il nous faut donc un état **extrêmement permissif**.

Notre lexeur doit aussi permettre d'identifier tous les mots-clés du langage, nous avons donc choisi d'envoyer un token à l'analyseur grammatical à chaque fois que nous en trouvions un.

2.3 Les choix effectués

Nous avons choisi de traiter certains cas simples directement dans l'analyseur lexical de manière à alléger l'analyseur grammatical :

◇ A la lecture d'un "mot" ou d'un nombre dans le lexeur, nous construisons instantanément les nœuds associés.

- mot : On renvoie un nœud de type **TREE**, contenant un fils de type **WORD**.
- nombre : On renvoie un nœud de type **INTEGER**.

◇ Pour la **gestion de la syntaxe de navigation entre les répertoires**, nous avons choisi de remplir la structure *path* directement dans le lexeur. En revanche nous effectuons la vérification du chemin dans la partie exécution et plus précisément dans la machine virtuelle.

3 L'analyse grammaticale, le "parser".

3.1 La grammaire

Tout comme l'analyseur lexical, l'analyseur grammatical a lui aussi beaucoup évolué tout au long du projet. En effet, le projet étant en constante évolution, nous avons été contraint de revoir entièrement notre grammaire de départ : au début nous rajoutions sans cesse de nouvelles règles au fur et à mesure de l'avancement du projet presque indépendantes des précédentes, sans tenir compte de l'importance de l'ordre de nos règles mais ceci est vite devenu impossible à gérer. Schématiquement notre grammaire ressemblait à quelque chose comme :

S → SA
| SB
| SC
| ...
A → ...
B → ...
C → ...

Il a donc fallu factoriser et "optimiser" cette grammaire.

Notre objectif principal était d'avoir une grammaire **sans conflit** Nous avons donc effectué une grammaire descendante où chaque règle se réduit en une règle plus basse dans l'arborescence. De cette façon nous pouvons très facilement rajouter une règle en fonction de sa priorité : Plus la règle est **haute** dans la grammaire, plus sa priorité sera **grande**. Notre grammaire ressemble donc à :

S → AS
A → ...
| B
B → ...
| C
C → ...

3.2 Les choix effectués

Comme pour l'analyseur lexical, il nous a fallu faire des choix qui nous semblaient **pertinents et importants** ou qui amélioraient la lisibilité et la compréhension de notre langage. Voici une liste des principaux choix que nous avons effectué :

- ◇ Un "let"de la forme : *let f a b = fun args -> ...* implique que "args" soit non vide. Il est totalement illogique d'utiliser le mot clef "fun" si c'est pour ne rien définir après.

- ◇ À l'intérieur des "{}", nous obligeons l'utilisateur à parenthésées les expressions ou plus précisément les opérations, les blocks "in", "where", ... Les fonctions ne sont pas sujettes à cette modification (exemple : *{ f a b, "..."}* ou *{ f a b}* sont des formes valides.)

- ◇ Pour la fonction emit : le premier argument de cette fonction est forcément un TEXT. De plus si nous voulons lui passer plusieurs expressions en même temps comme par exemple ceci :

```
let google = a[href="http://www.google.fr"]{"google"};
let labri = a[href="http://www.labri.fr"]{"labri"};
emit "index.html" google labri; // invalde
```

  nous devons utiliser :

```
emit "index.html" {google, labri};
```

4 Rendu et conversion HTML

Avant que la machine virtuelle nous soit fournie, il nous fallait un moyen de **tester nos productions** pour ne pas avancer à l'aveugle, nous avons donc écrit deux fonctions :

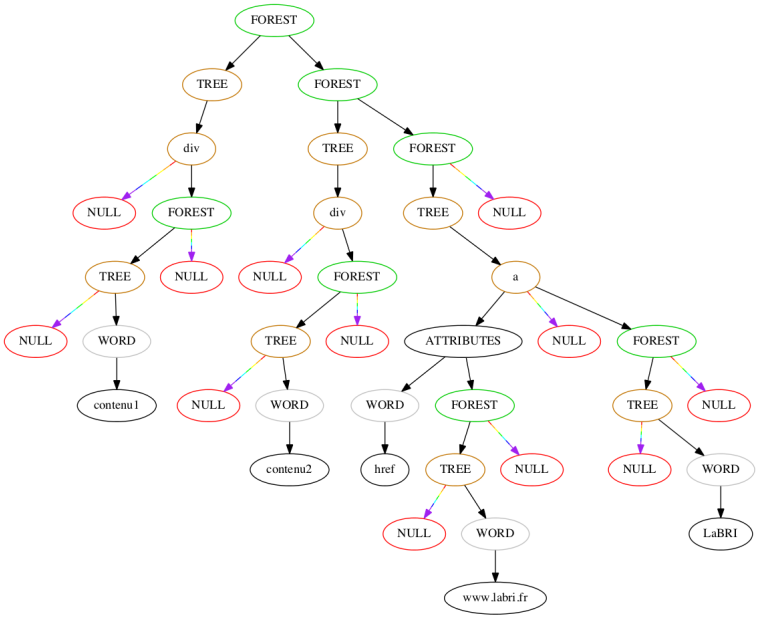
- Générateur graphviz
- Générateur HTML

4.1 Générateur graphviz

Nous avons mis en place une série de fonction permettant de dessiner une *struct ast* grâce à **Graphviz**. Ceci nous a permis de vérifier si nos constructions étaient valides et avaient le comportement souhaité.

Voici un exemple de résultats que nous obtenons :

Arbre généré par graphviz



Rendu HTML

contenu1
contenu2
LaBRI

4.2 Générateur HTML

A côté de cela nous avons aussi réalisé un convertisseur arbre → HTML.

De cette façon nous sommes capables de générer des pages web HTML simples mais fonctionnelles et nous pouvons donc évaluer en permanence le travail que nous produisons. Il subsiste tout de même encore quelques problèmes dont nous parlerons dans la partie : **Les difficultés rencontrées**. Voici un exemple de code généré :

```
<!DOCTYPE html>
<div>
  contenu1
</div>
<div>
  contenu2
</div>
<a href="www&#46;labri&#46;fr">
  LaBRI
</a>
```

5 Difficultés rencontrés

Durant notre projet, nous avons eu certaines difficultés, en voici une liste non exhaustive :

- Le principal problème auquel nous avons fait face était de réaliser une grammaire sans conflit. Cela nous a d'ailleurs poussé à refaire à plusieurs reprises notre grammaire. Nous avons donc opté pour une grammaire descendante ce qui nous a permis d'arriver à notre objectif.

- La fonction on_import nous a posée un réel problème. En effet, pour pouvoir utiliser un fichier ou une fonction de ce fichier, nous devons générer son arbre, cela dit nous ne savons pas exactement comment procéder pour récupérer cette arbre. Nous avions penser à appeler yyparse() sur le dit fichier et récupérer le résultat de l'évaluation mais cela impliquerait que yyparse() stocke cette valeur dans un pointeur pour que la machine y ait accès. Faute de temps nous n'avons pas pu tester cette solution.

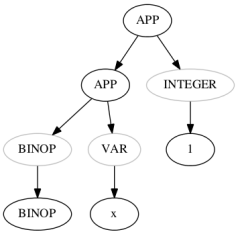
- La transformation des caractères spéciaux en caractères *HTML* : Nous arrivons à traiter tous les caractères spéciaux *ASCII*, mais nous n'avons pas réussi à gérer les caractères é, è, ... car ces caractères ne font pas partie de la table ASCII. Ils font partie de la table extended ASCII que le C ne gère pas. Nous affichons donc le caractère normal sans le convertir. Nous avions aussi pensé à utiliser "wchar_t" mais par manque de temps, nous n'avons pas pu implémenter cette solution.

- Pour les fonctions récursives, dans le cas où nous voulions écrire cela :

```
let f x =
     if x <= 0 then
         {}    else
         {x, f (x-1)};
```

  l'appel à la fonction f (x-1) n'est pas supporté car quand on utilise une expression entre parenthèses cela forme une forêt à part et n'est pas évalué tout de suite. Donc la fonction *f* va se rappeler avec un arbre et non une valeur. Nous avons essayé d'évaluer le *(x-1)* mais ici le problème est qu'il ne connaît pas la variable x, car x n'est pas globale.

Arbre généré pour *(x-1)*


APP
APP
INTEGER
BINOP
VAR
1
BINOP
x

6 Conclusion

6.1 Répartition des taches

La répartition des tâches a été assez naturelle, chacun d'entre nous apportait son aide où il le pouvait en fonction des besoins.

Sébastien et Nicolas ont beaucoup travaillé ensemble, sur le principe du **pair programming**. Ils se sont concentrés sur l'analyseur lexical, et l'analyseur grammatical (auxquels **Jimmy** a apporté une grande aide lors des nombreuses refontes de celle-ci) ainsi que sur les fonctions d'évaluations et la construction des arbres qui va avec.

Jimmy lorsqu'il n'aidait pas sur la partie analyse lexicale et grammaticale,il a réalisé l'utilitaire nous permettant de construire les graphes de nos arbres. Il a aussi implémenté la fonction qui transforme un arbre évalué en page web HTML.

Yordan s'est occupé de tester la robustesse de notre grammaire, et il nous a aidé quand nous le désirions sur des problèmes divers et variés. Il a écrit une batterie de test que nous pouvons faire passer à notre grammaire après chaque modification pour vérifier que celle-ci est toujours fonctionnelle. Il a aussi réaliser des pages HTML de test pour notre projet.

Sébastien a aussi réalisé toute la structure du projet (Arborescence, Makefile(s), ...). De plus il a corrigé la plupart des erreurs (par exemple il a débuggé tout le fichier machine.c avant que les correctifs ne sortent, etc.), et il a grandement amélioré la qualité de notre projet. Il a aussi réaliser beaucoup de fonctions intermédiaires que nous utilisons pour construire nos arbres etc.. Il a également repris les tests pour les adapter au mieux à notre code.

Nicolas avait aussi réalisé toute l'interface des arbres (avant que celle-ci ne soit fourni). Il s'est occupé de traiter les caractères spéciaux du langage HTML et autres fonctions utilitaires. De plus il a écrit une bonne partie du rapport.

Etienne quant à lui a essayé de créer une fonction "evaluate", mais très vite suspendue du fait de l'arrivée de la machine virtuelle.

6.2 Conclusion Générale

Ce projet permet de générer des fichiers web à partir d'un nouveau langage de programmation que nous venons de créer. Cela nous montre également que créer un langage de programmation de toutes pièces est accessible.

Ce projet nous a permis de comprendre comment fonctionnait un compilateur, par quelles étapes il doit passer pour fonctionner. Nous avons trouvé dommage que la machine virtuelle soit donnée au dernier moment et sur laquelle nous n'avons pas pu nous pencher correctement.