Analyse Syntaxique

RAKOTONIERA Hoby, VER VALEN PAIVA Willian et LAULAN Antoine

9 mai 2014

Table des matières

1 Présentation des fonctionnalités implémentées

1.1 Première partie du projet

Question 1

Le but de la première partie du projet était d'écrire un analyseur syntaxique. Nous avions comme source une machine virtuelle ***(machine.c)*** qui permettait de construire les arbres syntaxiques correspondant à des expressions, en utilisant les constructeurs fournis dans le fichier ***expr.c***. Nous avons donc implémenté un analyseur lexical flex ***(parser.l)*** et un analyseur syntaxique bison ***(parser.y)***. Tout d'abord,nous avons commencé par nous inspirer de l'exemple d'un TD pour implémenter une calculatrice. Nous avons, à l'aide du lexique reconnu par le fichier *flex*, codé les opérations de base dans le fichier *bison* en utilisant les fonctions données pour construire les arbres ***mk_**. Il nous a fallu aussi créer des règles de priorités entre les opérations pour éviter les conflits. Notre calculatrice une fois terminée était capable de faire toutes les opérations de base : ***+, -, *, /***, ainsi que toutes les opérations de ***comparaisons/égalité*** et pour évaluer de telles expressions, il suffisait de taper l'opération dans le ***parser.out*** suivis d'un ***« ; »***. Exemples d'expressions évaluées par notre calculatrice de base :

- - 3 + 5 ; // cela imprimait dans le terminal « » 8 »
- - 3 < 4 ; // cela imprimait dans le terminal « » 1 »

Ensuite, nous avons ajouté comme fonctionnalité la déclaration de variables du type :

```
let x = expression ;
```

ainsi que le if exp then exp else exp ;

```
T_LET T_ID[x] T_EQUAL e[expr] {$$ = push_rec_env($x,$expr,env);}
```

Pour ce faire, nous avons utilisé la fonction donnée ***push_rec_env*** afin de sauvegarder l'environnement lors de la déclaration de la variable.

Une fois ceci fait, nous étions capables de faire des opérations entre variables du style :

```
let a = 2 ;
let b = 3 ;
a+b ;
>>> 5
```

Grâce à cela nous avons pu implémenter à l'aide des fonctions fournies la déclaration des fonctions :

```
let f = fun x -> x+1 ;
```

Question 2

Il nous était demandé ici de retirer les parenthèses afin d'écrire des choses comme :

```
(fun x1 ... xn -> expr)
```

à la place de

```
(fun x1 -> ... (fun xn -> expr)...)
```

et

```
(f g1 ... gn)
```

à la place de

```
(... (f  g1)... gn)
```

Nous y sommes parvenus et nous avons même retiré toutes les parenthèses mais cela nous a posé quelques problèmes que nous expliquerons dans la partie du rapport « ***difficultés rencontrées***».

Question 3

Nous avons implémenté le *let in* et le *where* afin de pouvoir écrire des expressions comme :

```
let x = 5 in x < 10 ;
x < 10 where x=5 ;
```

Question 4

Cette partie était notée « optionnelle ». Cependant, comme nous étions en avance, nous l'avons commencée et de plus, nous savions que les listes étaient demandées dans la deuxième partie du projet.

1.2 Deuxième partie du projet

Question 1 : Ajout des listes

Nous avons implémenté, comme il était demandé, la gestion des listes par la machine. Pour ce faire, nous avons créé un nouveau type de données ***CELL*** . Exemple :

```
struct cell{
struct expr *ex ;
struct expr *next ;
} ;
```

Il a fallu modifier type *union node* en rajoutant le type ***CELL***. Il a fallu aussi créer la fonction ***mk_cell*** pour créer les cellules. Nous

avons aussi élaboré la fonction ***mk_nil*** afin de créer la liste vide, ou des cellules vides.

Il nous était demandé de réaliser les 3 opérations de base sur les listes, à savoir : *push*, *pop*, *top* : avec comme nom possible pour pop « head, hd, pop » , pour next « tail, next, tl » et pour push « ***: :*** » .

Pour l'utilisation des listes, une grosse modification du fichier ***machine.c*** a été nécessaire, ainsi que l'implémentation de quelques fonctions subalternes pour nous aider dans la compréhension du code. Il a fallu modifier le code fourni dans les opérations de la machine pour rajouter des « ***case*** » nécessaires à la gestion des listes, ainsi qu'une fonction d'evaluation des arguments pour chaque case. Une fois ceci fait nous pouvions utiliser des listes en les déclarant comme :

```
let l = [exp,...,exp]
```

et les opérations :

- - push : a : :l
- - pop l ou hd l
- - tail l ou next l
- - liste vide []

Question 2 : Ajout de la partie graphique

Partie 1 :Objets de base

Génération de dessins :

Pour cette partie, il s'agissait d'ajouter des types et fonctions de manipulation pour pouvoir définir des dessins et les afficher. Pour ce faire, nous avons d'abord défini des nouveaux types de données :

- Point : e1,e2 où e1 et e2 sont des expressions s'évaluant en NUM (int)

- Chemin (path) : p1- -p2- -...- -pn où pn sont des expressions s'évaluant en type POINT

- Cercle (circle) : Circle(e1,e2) où e1 et e2 sont des expressions et e1 s'évalue en type POINT alors que e2 s'évalue en type NUM

- Courbe de Bézier (bezier) : Bezier(e1,e2,e3,e4) où e1,e2,e3,e4 sont des expressions s'évaluant en type POINT.

  Pour chaque type de données il a fallu créer un nouveau « enum kind », modifier le « union node », faire des fonctions de construction du type ***mk_*****.

  Nous avons utilisé ces fonctions dans le ***parser.y*** pour que notre analyseur syntaxique puisse reconnaître et dessiner ces types de données sur une page HTML.

  Ensuite nous avons implémenté une fonction *map* permettant de générer une page html avec du code j*avascript/HTML5 canvas* pour dessiner les nouveaux types de données ci-dessus.

  Cette fonction est partagée en plusieurs sous fonctions :

- Deux fonctions, *html_head* et *html_tail* permettant d'écrire la partie commune du code à toute les pages html générées pour le dessin.

- Une fonction *draw_** pour chaque type de données que l'on peut dessiner.

- Une fonction *draw* générale qui appelle la bonne fonction *draw_** en fonction de la figure passée en paramètre.

- Une fonction *map* qui appelle *html_head()*, *draw()*,*html_tail()*

  Nous avons ensuite modifié cette fonction pour qu'elle puisse prendre en paramètre une liste d'objets dessinables. La fonction *map* est capable de dessiner sur une page ***HTML*** plusieurs figures.

Partie 2 : Transformation

Dans cette partie, le but était d'implémenter des transformations géométriques appliquées aux objets de base codés dans la partie précédente. Trois transformations étaient demandées :

- ***Translation (translate)*** : transformation séparée en deux fonctions. La première, *trans_point(p,v)* applique une translation au point p passé en paramètre par rapport au vecteur v. La seconde, *translate()* applique la fonction *trans_point()* à tous les points de la figure passés en paramètre, et renvoie la figure modifiée par la translation.

- ***Rotation (rotate)*** : transformation séparée en deux fonctions. La première, *rotate_point(p,v,a)* applique une rotation au point p passé en paramètre par rapport au vecteur v et l'angle a donné en degrés. La seconde, *rotate()* applique la fonction *rotate_point()* à tous les points de la figure passés en paramètre, et renvoie la figure modifiée par la rotation.

- ***Homothétie (scale)*** : transformation séparée en deux fonctions. La première, *scale_point(p,v,r)* applique une homothétie au point p passé en paramètre par rapport au vecteur v et au rapport r. La seconde, *scale()* applique la fonction *scale_point()* à tous les points de la figure passés en paramètre, et renvoie la figure modifiée par l'homothétie.

2 Valorisation du travail réalisé

2.1 Difficultés rencontrées

Au départ, il a été difficile de bien comprendre les fichiers fournis, car le code n'était pas très commenté. C'est pourquoi quand nous avons commencé à implémenter la calculatrice et la syntaxe de base, nous avons pas mal utilisé la méthode par tâtonnement, c'est à dire sans vraiment comprendre comment cela marchait. Nous avons cependant dû rattraper cela lorsque nous avons commencé à faire des choses plus compliquées comme le *let* et les déclarations de fonctions.

Les premières grosses difficultés rencontrées ont été d'enlever les parenthèses dans les applications de fonctions multiples et les fonctions avec plusieurs arguments. Dans un premier temps nous avons retiré les parenthèses pour les fonctions à argument multiples par exemple :

```
(fun x y z -> x+y+z)
```

Ce problème a été résolu grâce à l'utilisation d'un arg_list :

```
arg_list:T_ARROW e           {$$=$2;}
        |T_ID[var] arg_list {$$=mk_fun ($1, $2); }
```

La seconde difficulté rencontrée l'a été pour enlever les parenthèses dans les applications de fonctions, parce que nous avions affaire à une précédence pour des règles sans terminals, uniquement des non-terminals. Par exemple :

```
| e[fun] e[arg]              { $$ = mk_app($fun,$arg);}
```

Puisqu'il n'y a pas de token impliqué dans l'application de fonction, les règles normales de précédence *yacc/bison* ne fonctionneront pas sans aide supplémentaire. Pour résoudre cela, nous avons dû déclarer un token fictif, comme par exemple :

```
%left FUNCTION_APPLICATION
```

Ce token n'est créé que parce qu'on en a besoin pour forcer la précédence de la règle d'application de fonction, comme nous l'avons fait :

```
| e[fun] e[arg] %prec FUNCTION_APPLICATION { $$ = mk_app($fun,$arg); }
```

On a utilisé une directive spéciale ***%prec*** qui permet d'appliquer une précédence manuellement.

Lorsque nous somme parvenus, dans la partie gestion graphique, au moment où la fonction *map* (qui dessine plusieurs figures sur une page html) devait prendre une liste d'objets en paramètre, nous nous sommes rendu compte d'une erreur sur nos listes.. C'est à dire que lorsque nous avons créé nos listes, nous avons testé celles-ci seulement avec des listes d'entiers et cela fonctionnait correctement. Cependant lorsque nous avons du faire des listes avec autre chose que des entiers, des variables par exemple, nos listes ne marchaient plus.. Le problème était que lors des déclarations de listes comme *let l = [a,b];* on obtenait une *syntax error*. Cela nous a pris pas mal de temps pour trouver d'où le problème venait... Au départ, nous nous sommes concentré sur notre fichier *bison*, pensant que notre syntaxe des listes était mauvaise. Il s'est avéré que le problème venait de notre analyseur lexical..

A l'aide de la fonction ***yydebugg*** nous nous sommes rendu compte que la règle :

```
[a-zA-Z]+ {yylval.id = strdup(yytext) ; return T_ID;}
```

renvoyait *" [a "* au lieu de deux tokens différents. En remplaçant cette règle par :

```
[[:alpha:]_]+  {yylval.id = strdup(yytext) ; return T_ID;}
```

cela a resolu notre problème, et nous avons pu faire des listes contenant aussi des variables et *a fortiori* des objets.

aller voir : `http://stackoverflow.com/questions/23368667/syntax-not-being-recognized-by-23407424#23407424`

2.2 Explications des choix effectués

Par rapport aux choix effectués, il n'y pas de grandes modifications par rapport à ce qui était demandé dans le sujet. Nous avons cependant choisi de faire toute la partie graphique plutôt que de faire un peu de la partie image, et un peu de la partie sons.

Concernant les divergences vis-à-vis du sujet, nous avons fait le choix de mettre toutes nos déclarations de fonctions et autre token en anglais pour avoir une homogénéité au niveau de la langue.

Nous avons aussi choisi, sur les conseils de notre chargé de TD, de retirer toutes les parenthèses concernant les fonctions à arguments multiples et les applications de fonctions multiples. C'est-à-dire que notre analyseur syntaxique est capable d'évaluer des expressions comme :

```
f x y z -> x+y-z
```

à la place de

```
(f x y z -> x+y-z)
```

et

```
f g h i
```

à la place de

```
(f g h i)
```

Le reste de nos choix a été fait selon ce qui nous semblait le plus naturel, car plusieurs questions nous laissaient libre de nos choix en nous demandant simplement le résultat attendu, sans aucune indication concernant la marche à suivre pour y parvenir.

3 Annexes et code du projet

L'ensemble des codes et des annexes est consultable en suivant le lien ci-dessous :

`https://github.com/WillianPaiva/project_AS`