Rapport Projet Programmation 2

Menini Quentin
Brebant Alexandre
Randriamalazavola Andrea Ruffino

10 mai 2014

Table des matières

Introduction

Durant ce projet, nous avons créé un analyseur syntaxique et modifié une machine permettant de créer un mini-langage permettant la constructionn la transformation et l'affichage d'objets tels que des entiers, des listes, des formes géométriques et des musiques.

Première partie

Construction de base

0.1 Calculatrice

Nous avons commencé par reprendre une partie du code de notre calculatrice vue en TP pour tout ce qui est plus moins ID etc... Et nous avons modifié notre parser.y pour utiliser les fonctions de machine.c et les structures de expr.h.
Notre lex fonctionne comme ceci :
Si on voit des nombres, on passe le nombre dans yylval.num pour le récupérer dans parser.y et on renvoie le token T_NUM,
Si on voit un ID (suite de caractère commençant par une lettre suivi par une suite de chiffres lettres et/ou underscore) on le sauvegarde dans yylval.id et on renvoie le token T_ID. Pour les symboles, ('+', '-', '*', '=', '(', ')', '[', ']', '%', '{', '}') nous renvoyons directement le symbole.
Pour chaque mot clé (let par exemple) ou les symboles de comparaison, nous avons créé un token.

Du côté de notre grammaire, nous avons choisi un label 'e' pour tous nos objets. Nous utilisons le label 's' pour une expression complète avec un ';'.
Les règles de s à cette étape du projet sont : s : s EOE (on ne fait rien, EOE est un ;, cela veut dire que nous avons un ';' tout seul après une expression. s : s e EOE (pour évaluer et afficher e) s : s T_LET T_ID '=' e EOE (que nous allons voir dans la prochaine partie).

0.2 Let

Pour utiliser le let, nous utilisons les variables globales suivantes :
- struct env *environment = NULL ;
- struct configuration conf_concrete ;
- struct configuration*conf = &conf_concrete ;
Nous avons au début utilisé seulement la fonction push_rec_env comme ceci :

```
environment = push_rec_env($var,$expr,environment);
struct closure * cl = mk_closure($expr,environment);
conf->closure = cl;
conf->stack = NULL;
```

Avec $var notre ID et $expr notre 'e'.

0.3 If

Le if ne nous à pas posé de problèmes, avec la simple règle "e : T_IF e T_THEN e T_ELSE e" et en mettant T_ELSE associatif à droite avec une faible priorité, cela ne nous a créé aucun conflit et nous n'avons jamais mis de parenthèses autour d'un if.

0.4 Fonctions

0.4.1 Fonctions avec un paramètre

Pour créer une fonction avec un paramètre, la règle "e : T_FUN T_ID T_ARROW e" est suffisante avec T_ARROW associatif à droite avec la même priorité que T_ELSE. On peut donc écrire une fonction comme ceci : "fun x -> x+4". On appelle mk_fun avec comme premier paramètre la chaine de caractères correspondant à l'ID et l'expression après les flèche. Pour appliquer une telle fonction il suffit d'écrire "(f 3)" et la fonction f sera appliquée à 3.

0.4.2 Fonctions avec plusieurs paramètres

Avant d'enlever les parenthèses, nous n'avons pas eu besoin de créer de nouvelles règles pour les fonctions de plusieurs paramètres, il suffisait d'écrire une fonction comme ceci : "fun x ->(fun y -> x + y)", nous pouvions également écrire "fun x -> fun y -> x + y" car nous avions déjà choisi de ne pas mettre le parenthèses autour de la déclaration d'une fonction. Pour appliquer une fonction à plusieurs paramètres avant d'enlever les parenthèses, il fallait écrire "((f 1) 2)" pour appliquer f(1,2).

0.5 Enlever les parenthèses

0.5.1 Déclaration de fonctions

Pour la déclaration de fonctions, nous n'avions déjà pas de parenthèses, mais nous voulions pouvoir écrire "fun x y -> x + y" au lieu de "fun x -> fun y -> x + y". Pour cela, nous avons créé un label intermédiaire 'p' qui correspond à une fonction de plusieurs paramètres dont tous les paramètres ne sont pas encore déclarés. Les règles ajoutées sont les suivantes :
p : T_ID T_ARROW e $$ = mk_fun ($1,$3) ;
| T_ID p $$ = mk_fun ($1,$2) ;
e : T_FUN T_ID p $$ = mk_fun ($2,$3) ;
Voici un exemple de dérivation :
fun x y -> x + y – y -> x + y se dérive en p, nous avons donc fun x p qui se dérive en e.

0.5.2 Applications de fonctions

Pour l'application de fonctions, nous avons d'abord souhaité ne laisser que les parenthèses extérieures à l'application de la fonction par exemple écrire (f 1 2 3) pour (((f 1) 2) 3), car nous trouvions cela plus logique que d'enlever toutes les parenthèses, (si l'on avait "f 3 + 2", fallait-il faire (f 3) et ensuite ajouter 2 ou faire (f 5), cela aurait entrainé un choix à faire et nous avons préféré notre version qui était plus compréhensible par l'utilisateur sans avoir à expliciter le choix de la syntaxe. Pour appliquer une fonction à plusieurs paramètres, nous

avons procédé comme pour la déclaration de fonction mais cette fois en partant du début. Nous avons ajouté les règles suivantes :
e : f e ')' $$ = mk_app($1,$2) ;
f : '(' e e $$ = mk_app($2,$3) ;
| f e $$ = mk_app($1,$2) ;
Voici un exemple de dérivation :
(g 1 2) – (g 1 se dérive en f, nous avons donc f 2) qui se dérive en e.

Nous avons quand même essayé d'enlever toutes les parenthèses, pour cela nous avons essayé deux solutions :
- La première consistait à créer un type spécifique pour les fonctions, mais nous avions du coup un problème avec les ID, car un ID pouvait représenter un nombre ou une fonction, nous avons donc créé une fonction que nous utilisions dans le lex pour ne pas retourner le même token lorsqu'un id correspondait à une fonction. Le problème de cette solution est qu'elle ne fonctionnait pas pour les applications de fonctions à plusieurs paramètres, en effet après l'application du premier argument, nous ne pouvions pas savoir si cela devait donner une fonction à appliquer au prochain paramètre ou une variable.
- La deuxième solution était d'utiliser des atomes, mais cette solution n'a pas aboutie non plus, surement par manque de compréhension sur les atomes.

0.5.3 Moins unaire

Nous avons essayé de gérer le moins unaire avec une règle de priorité spéciale (en utilisant le mot clé prec) mais cela n'a pas fonctionné, nous avons donc décidé que tous les moins unaires seraient parenthésés (par exemple pour appliquer la fonction f à 2 et -3, il faut écrire "(f 2 (-3))").

0.6 Let rec

Après plusieurs tests avec notre version qui n'utilisait que la fonction push_rec_env, nous nous sommes rendu compte que nous ne pouvions pas modifier la valeur d'une variable en fonction de son ancienne valeur, nous avons donc décidé de créer un token T_REC, pour créer une variable récursivement, il faut donc écrire "let rec f = ...", ceci utilise la fonction push_rec_env de la façon décrite dans le paragraphe sur le let. Le let "normal" utilise donc désormais la fonction pushde la façon suivante :

```
struct closure * cl = mk_closure($expr,environment);
conf->closure = cl;
conf->stack = NULL;
environment = push_env($var,cl,environment);
```

Deuxième partie

Ajout des listes

Pour la création des listes, nous avons pour la première fois touché à la structure et à machine.c.
Nous avons tout d'abord créé une structure cell, qui contient un "struct expr *" et un "struct cell *", le premier (le car) contenant le premier élément de la liste et le second (le cdr) contenant la liste commençant par le 2e élément. Puis nous avons modifié la fonction step dans machine.c pour pouvoir utiliser la fonction cons qui ajoute un élément en tête d'une liste, puis plus tard head et tail pour afficher le premier élément ou la liste ne contenant pas le premier élément.
Après avoir modifié la fonction step et créé les fonctions adéquates pour la création et la modification de la liste, nous avons modifié notre fichier bison pour créer les listes.
Nous créons les listes récursivement de la même façon que pour les fonctions en commençant par créer la liste vide et nous ajoutons en tête les éléments un par un.
Les règles sont les suivantes :
list : '[' l $$ = $2 ;
l : e ']' $$ = mk_app(mk$_a pp(mk_o p(CONS), \$1), mk_c ell(NULL, NULL))$;
$|e', l\$\$ = mk\_app(mk_a pp(mk_o p(CONS), \$1), \$3)$;
$|']'\$\$ = mk\_cell(NULL, NULL)$;

$Lorsque nous avons testles listes, nous nous sommes rendu compte que nous ne pouvions pas crer des listes de plus$
$.Pour rgler ce problme, nous avons mis dans le cdr un struct expr * contenant la liste.(Cette solution a trouve par$

$Nous avons aussi cr 3 fonctions que nous avons dcid de ne pas laisser dans le parser.La premire est la fonction hea$
$imelment d'une liste, la seconde est la fonction append qui concatne deux listes, qui le lemme utilise la fonction r$

Troisième partie

Génération de dessins

0.6.1 Dans la structure

Pour les dessins, dans expr.h nous avons créé les types point (contenant deux int), circle (contenant un point en struct expr * et un int) et bezier (contenant 4 points en struct expr *). Pour le type path, nous n'avons pas créé de structure car nous avons utilisés celle du type liste, un path étant une liste de points.
Pour être sur que les expressions passés dans un point s'évaluent bien en int, nous avons choisi de d'abord créer un point initialisé à 0,0 et ensuite de modifier les valeurs des points grâce aux fonctions setabs et setord dans le step (on utilise le step pour évaluer les expressions en paramètres du point).
Nous avons choisi de faire de même pour les cercles et les courbes de bezier. Pour les path, nous avons procédé comme pour les listes mais en partant du début au lieu de partir de la fin.

0.6.2 Transformations

Une fois les structures au point, nous avons écrit les transformations, comme toujours en modifiant machine.c et en créant les fonctions adéquates.
Les fonctions de transformation sont la translation, la rotation et l'homotethie.
Dans le bison nous avons utilisé les règles suivantes :
e : $T_{T}RANSLAT'('e',' e')'\$\$ = mk_{a}pp(mk_{a}pp(mk_{o}p(TRANSLATION), \$3), \$5);$
$|T_{R}OTAT'('e',' e',' e')'\$\$ = mk_{a}pp(mk_{a}pp(mk_{a}pp(mk_{o}p(ROTATION), \$3), \$5), \$7);$
$|T_{H}OMOT'('e',' e',' e')'\$\$ = mk_{a}pp(mk_{a}pp(mk_{a}pp(mk_{o}p(HOMOTHETIE), \$3), \$5), \$7);$

$Nousavonscritunefonctionpartransformation, quitested'abordletype, eteffectueensuitelestransformat$

0.6.3 Affichage

Pour afficher les dessins, il fallait créer un fichier .js pour creer le canvas en javascript, nous avons donc créé des fonctions dans un fichier $js_{w}riter.cpourcriredansunfichier"html_{c}ode.htn$
$Parappeldelafonctiondraw, lesfichierssontcrsetlesdessinssontaffichsdanslapagehtml.$

Quatrième partie

Génération de fichiers musicaux

0.6.4 Dans la structure

Pour la musique, nous avons créé une structure contenant une liste de notes (toujours struct expr *), une tonique (char *), et deux int pour la durée (numerateur et denominateur pour ne pas casser la fraction, celle ci est simplifiée au moment de l'affichage).
Nous avons aussi créé une structure pour les notes, contenant un int (la valeur) et deux char *, info1 et info2. Le premier contient soit b si la note est un bemol, soit si la note est un dièse et " sinon. Le second contient des '-' ou des '.' pour donner la longueur de la note. Nous avons utilisé le tiret normal et non le tiret semi-cadratin car le tiret semi-cadratin ne peut pas être parsé en char car c'est un double caractère.
Pour la création de musique, nous avons créé un token qui est renvoyé par le lex lorsque l'on écrit '(', pour savoir quand une musique commence. Ceci peut être génant lors de la création d'un point, si par exemple le premier élément d'un point est un appel de fonction (f 3) par exemple, on ne peut plus écrire (f 3),3 mais il faut écrire (f 3), 3 pour pas rentrer dans le mode musique dans le lex.
Il y a trois nouveaux modes dans le lex : le mode NOTES, qui est démarré après '(' et qui renvoie les notes de la liste ; le mode NOTESNEG, qui gère les notes en négatif (biensur mises entre parenthèses) ; et le mode MUSIQUE, qui renvoie le char * correspondant à la tonique et le ou les int correspondant à la durée d'une note et le caractère '/' si nécessaire.

0.6.5 Affichage

Cette partie a été une des parties les plus compliquées du projet, car il fallait comprendre la syntaxe lilypond et les gammes, puis pouvoir calculer la note en fonction de la gamme. Nous avons pour cela créé musique.c, qui avec une série de fonction trouve la note en fonction de la valeur de la note dans notre syntaxe et de la tonique.
Nous avions quelques notions de musique mais les recherches faites sur les gammes nous ont permis d'apprendre des choses, lors de l'implémentation de ces fonctions, nous avions une guitare avec nous pour bien visualiser les cases et pouvoir jouer les gammes.
Nous avons aussi calculé le nombre d'octaves de différences entre deux notes qui se suivent car lilypond met par défaut la note la plus proche de la note précédente, il fallait donc ajouter ou enlever des octaves suivant la position de la note.
Après tous ces calculs, tout ceci est concaténé dans une chaine de caractère que l'on écrit dans le fichier musique.ly qui une fois compilé nous donne la partition souhaitée.
Nous créons ce fichier en appelant la fonction print(a) avec a une liste de musiques.

0.6.6 Transformations

Voici les transformations que nous avons codées :
- La concaténation : Pour concaténer deux musiques, nous créons tout simplement une liste contenant les deux musiques et nous avons permis dans notre fonction print de lire des listes de listes tant que tous les éléments de ces listes sont des musiques.
- La transposition : Pour la transposition, nous testons d'abord si la première tonique en paramètre est la même que celle de la musique, dans ce cas on remplace la tonique de la musique par la deuxième tonique en paramètre, sinon on calcule la différence entre les deux toniques et on l'ajoute à la tonique de la musique.
- L'ajout : Lorsque l'on additione une musique avec un entier, les valeurs de toutes les notes de cette musique sont additionées avec l'entier.
- Le changement de vitesse : Lorsque l'on multiplie ou l'on divise une musique par un entier, le temps d'une note de cette musique est multiplié ou divisé par cet entier.
- L'inverse : Pour inverser une musique, nous avons créé une fonction invmusique, qui, si la musique est une liste de musique, inverse la liste et s'appelle pour chaque élément de la liste. Lors des tests, nous nous sommes rendus compte que cette fonction fonctionnait bien pour les musiques simples mais pas pour les musiques qui sont des listes de musiques, mais nous n'avons pas réussi à réparer ce problème.
Comme dans les exemples du sujet, les fonctions d'ajout et de changement de vitesse de musique changent directement la valeur des musiques modifier au lieu d'en créer une autre et de devoir utiliser un let pour la sauvegarder.

Conclusion

Avec ce projet, nous avons appris beaucoup, que ce soit en analyse syntaxique ou en musique, en syntaxe lilypond ou en canvas javascript. Nous avons trouvé interessant de créer un mini-langage et malgrès les quelques problèmes que nous avons eu, nous sommes assez content du résultat car nous avons créé quelque chose de fonctionnel et le fait de pouvoir visualiser le résultat sous forme de dessin ou d'image est encore plus interessant.