université de BORDEAUX

PROJET DE COMPILATION – LICENCE 3

Mardi 10 Avril 2018

AUDRAIN Alexis, GUTIERREZ Pablo, LARTIGUE Ghislain, LÉVÊQUE Maxence

Année Universitaire 2017-2018

Table des matières

1. Contexte

Le but de ce projet était de réaliser un compilateur en LEA pour le langage C3A. Pour cela, il nous fallait donc réaliser un analyseur syntaxique, un analyseur lexical, un analyseur sémantique, un interpréteur du langage, un compilateur Léa vers C3A et enfin un émulateur C3A. Nous devions réaliser ce travail par groupe de 4 personnes, avec un délai d'environ un mois.

2. Travail réalisé

a. Les fichiers Lex et Yacc

Tout d'abord, nous avons commencé par travailler sur le yacc et le lex. Pablo s'occupait du fichier yacc tout seul : il a réécrit le yacc donné dans la documentation du projet, puis a ajouté des règles de priorité sur les opérateurs. Comme il a rapidement terminé cette tâche, il a ensuite aidé Ghislain à écrire le fichier lex.

Le lex et le yacc sont utilisés afin de faire une analyse lexicale (entre autres). A eux deux, ils permettent de vérifier la syntaxe d'un code fourni. Le lex va détecter l'apparition de certains symboles ou certaines chaînes de caractères présents dans le fichier à analyser, et renvoyer au yacc le type auquel cette chaîne correspond.

Les règles de priorité

Le yacc va ensuite effectuer les règles de priorités que nous avons ajoutées : ces règles sont identiques à celles vues en TD. Les règles des conditions sont les moins prioritaires et ne sont pas associatives, puis le " OU " et le " ET " sont prioritaires à gauche, vient ensuite la négation (non associative), puis au même niveau de priorité les égalités, inégalités et différences, non associatives elles aussi. En dernier lieu se trouvent les multiplications / divisions et l'addition / soustraction toutes deux prioritaires à gauche. La dernière ligne s'occupe des valeurs négatives.

La capture d'écran ci-contre illustre les règles de priorités que nous avons réalisées.

```
18
19  %nonassoc THEN
20  %nonassoc ELSE
21
22  %left OR
23  %left AND
24  %nonassoc NOT
25
26  %nonassoc LT LE GT GE EQ DIFF
27
28  %left PLUS MINUS
29  %left TIMES DIV
30  %nonassoc UNARYMINUS
```

Les expressions régulières

Il nous a ensuite fallu trouver des expressions régulières nous permettant de détecter les commentaires, qu'ils soient faits avec un double slash ou bien en utilisant les symboles suivants : /* et */.

La première fonction est simple : lorsque l'on détecte un // sur une ligne, le reste de la ligne ne doit pas être analysé.

La seconde cependant est un petit plus délicate à détecter, car il faut commenter tout ce qui se situe entre la balise ouvrante, soit /* et la balise fermante qui est */. Ceci permet donc de commenter plusieurs lignes d'un coup.

Nos expressions finales sont celles ci-dessous : la première représente le commentaire avec un // et la seconde celui avec /* et */.

```
\/\/.*                                  {;}
\/\*(\n.*)*\*\/                         {;}
```

Pendant ce temps, Maxence et Alexis travaillaient sur l'arbre de types.

b. L'arbre des types

Les fichiers de types (respectivement nommés Type.c et Type.h) ont pour but de générer les types, de les afficher, mais aussi de vérifier l'égalité entre deux types (ceci est également applicable pour les arbres de types).

La première fonction, createType(), permet de créer un type grâce au tag donné en paramètre ainsi que ses fils gauche et droit s'il en a un. S'il n'en a pas, ces paramètres vaudront null.

Les différents types possibles sont : integer, boolean, character, pointer, array ainsi que des fonctions contenant en paramètre un certain nombre d'arguments composés des types cités précédemment.

```
Type* createType(Tag tag, Type* left, Type* right){
  Type* tmp = malloc(sizeof(Type));

  tmp->tag = tag;
  tmp->left = left;
  tmp->right = right;

  return tmp;
}
```

Cette seconde fonction, comme son nom l'indique, convertit un Tag en un String.

```
char* convertEnumToString(Tag tag) {
  switch(tag) {
    case INT:
      return "INT";
    break;
    case BOOL:
      return "BOOL";
    break;
    case CHAR:
      return "CHAR";
    break;
    case PTR:
      return "PTR";
    break;
    case ARR:
      return "ARR";
    break;
    case FUNC:
      return "FUNC";
    break;
    case PROD:
      return "PROD";
    break;
  }
}
```

La fonction suivante, displayType(), permet d'afficher l'arbre. Elle utilise la fonction précédente afin de récupérer le type dans un String.

displayType() est une fonction récursive ; cela est pratique pour afficher les fonctions qui sont représentées comme des arbres binaires.

```c
void displayType(Type* type){

  // TAG(lhs, rhs)
  printf("%s(", convertEnumToString(type->tag));
  if (type->left) {
    displayType(type->left);
  }

  printf(",");

  if (type->right) {
    displayType(type->right);
  }

  printf(")");
}

int EqualTo(Type* type1, Type* type2){
  if (type1->tag != type2->tag)
    return 0;

  if (type1->left && type2->left){
    if (!EqualTo(type1->left, type2->left))
      return 0;
    }
  else if(type1->left || type2->left)
    return 0;

  if (type1->right && type2->right){
    if (!EqualTo(type1->right, type2->right))
      return 0;
    }
  else if(type1->right || type2->right)
    return 0;

  return 1;
}
```

La dernière fonction, EqualTo (voir la capture d'écran ci-dessus), vérifie si deux types sont égaux. Cette fonction est également récursive si l'un des deux types à vérifier s'avère être un arbre.

c. Arbre de syntaxe abstrait

Le fichier des arbres abstraits permet de gérer facilement les groupements d'opérandes. Il est utilisé pour faire une analyse syntaxique en association avec le fichier yacc.

Lorsque tout le monde avait fini sa partie, Pablo a revu le code présent dans le fichier des types, tandis que Ghislain modifiait le yacc pour le préparer à l'arbre de syntaxe abstrait et aux expressions, respectivement réalisés par Alexis et Maxence.

d. L'environnement

L'environnement permet de répertorier les différentes variables du programme et leur type.

Il est rédigé en un fichier environment.c qui permet deux choses : ajouter une variable à l'environnement via son nome et son type, et, de trouver une variable à partir de son nom afin de connaitre son type. Pour y arriver, nous avons décider d'utiliser une liste chainée qui répertorie les variables crées.

3. Capitalisation d’expérience

Ce projet nous a permis une meilleure compréhension des fichiers yacc et flex ainsi que de leur utilisation.

4. Difficultés rencontrées

Au début du projet nous ne connaissions pas le langage Léa, et en comprendre les mécaniques nous a demandé beaucoup de temps. En plus de cela, les documents fournis pour nous aider (comme l'addenda) contiennent des exemples de code en Léa et non en C, ce qui nous a empêché de comprendre certaines portions de code ou d'être capable de les implémenter en C.

Nous avions continuellement besoin de Mr Lionel Clément pour la réalisation du projet car nous n'avions pas les connaissances et les compétences nécessaires. Ceci a été vraiment pénalisant pour avancer en dehors des cours. Il est pour nous impensable d'avoir besoin en continu d'un professeur pour nous seconder.

Nous avons eu aussi beaucoup de difficultés pour comprendre les étapes du projet et pour les réaliser par exemple l'arbre de syntaxe abstrait.

Nous avions bien évidemment compris que le projet avait pour but d'être complexe, mais nous pensons qu'il devrait y avoir un juste milieu. Ce qui nous amène à la dernière partie la conclusion.

5. Tests

Nous n’avons malheureusement pas pu réaliser de test car nous ne sommes pas parvenus à compiler notre projet via notre Makefile.

6. Ce qu'il reste à faire

Nous n'avons pas terminé la représentation d'un arbre de syntaxe abstrait. Nous connaissions son principe, mais sa réalisation s'est avérée bien plus difficile que prévu. Avec cela nous aurions pu finir l'analyse syntaxique. Pour cette raison, par rapport à l'énoncé du projet, il nous reste à réaliser tout ce qui se trouve à partir de la question 2.

7. Conclusion

Nous pensons que le problème ne provient pas du projet en lui-même mais de l'UE Compilation en général. Si vous demandez à la promotion leur avis sur ce projet et l'UE la majorité vous répondra qu'elle comprend peu de choses, voire carrément rien. Certes, quelques personnes avaient déjà certaines notions apprises de leur côté en autodidacte, mais cela reste une minorité.

Cette matière aurait pu être intéressante et nous apporter des connaissances sur tout le domaine de la compilation. Mais entre les cours et ce projet, le seul sentiment que nous avons ressenti est de pas être fait pour ce domaine et de ne pas le comprendre.

Nous pensons aussi qu'il devrait y avoir des changements dans l'organisation des cours et des TD. Pour seul exemple, nous avons vu pendant environ 1h la syntaxe des Makefile et comment en écrire un plutôt sommaire. On nous a aussi indiqué que presque plus personne ne les réalisait à la main. Nous avons acquiescé et nous nous sommes dit que c'était intéressant pour notre culture générale. Cependant, nous avons plusieurs fois eu la remarque de compiler les fichiers en lignes de commandes et de ne pas utiliser de Makefile. Nous n'avions alors pas assez de connaissances pour en créer un de nous même car il nous avait été indiqué que plus personne ne faisait ça. Ce genre d'incohérence rend d'autant plus difficile à comprendre la finalité des TD.

Nous comprenons la démarche d'un projet complexe, mais il nous a été répété qu'en groupe les premières étapes étaient rapides. Seulement même à quatre, si personne ne comprend, on ne peut pas avancer. Nous trouvons que vous cherchez à placer des élèves là où ils ne sont pas en termes de compétences techniques. Pour nous, un projet d'envergure devrait être fait sur un semestre entier et au cours duquel l'enseignant serait là pour nous former et nous donner des idées, et non travailler à notre place pour compenser une absence de connaissances de notre part.

Pour parler plus en détails du thème du projet, nous n'avons pas saisi l'intérêt d'écrire un compilateur pour un langage que nous connaissions pas. Le faire pour un langage connu comme le C aurait été bien plus intéressant.

Pour conclure, nous avons été attristés de notre incapacité à réaliser ce projet dans un domaine qui comme nous l'avons dit, aurait pu être intéressant et nous apprendre beaucoup.