École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise

Projet : Analyseurs syntaxique et sémantique pour un sous-langage dAda

Constant Gayet et Barnabé Geffroy

5 janvier 2022

Sommaire

1 Introduction

Le but de ce projet est décrire un exécutable qui prend en entrée un programme écrit en Little Ada, un sous-langage dAda, qui construit son arbre de syntaxe abstraite et qui fait quelques analyses sémantiques dessus.

Pour vous renseigner sur la compilation du projet et l'exécution des tests, référez-vous au fichier `README.adoc` qui détaille notamment le fonctionnement du fichier `Makefile`. La compilation utilise **flex/bison**, car sous lex/yacc certaines commandes ne passent pas (i.e. `%empty`).

2 Structure du code

Le projet est composé de trois fichiers :
— `little_ada.c`
— `little_ada.l`
— `little_ada.y`

2.1 `little_ada.c`

Ce fichier contient une structure du langage Little Ada pour la question 1 du projet.

2.2 `little_ada.l`

Ce fichier contient l'analyse lexicale du langage Little Ada. Dans celui-ci on utilise les expressions régulières pour renvoyer le bon token associé.

2.3 `little_ada.y`

Ce fichier contient l'analyse syntaxique du langage Little Ada. Il contient également les fonctions des questions 3 et 4 du sujet (les questions 5 et 6 n'ont pas été traitées).

3 Réponses aux questions du projet

3.1 Question 1

Pour la première question nous avons édité un fichier `little_ada.c` dans lequel nous avons défini la structure du langage Little Ada. Par la suite ce fichier n'a pas été utilisé. Nous n'avions alors pas réellement compris comment utiliser lex/yacc pour la définition des types.

3.2 Question 2

Cette question est celle qui nous a demandé le plus de temps. Elle s'est décomposé en plusieurs phase. La rédaction des expressions régulières. Celle-ci a été fastidieuse notamment l'identification des constantes. Ensuite nous nous sommes assurés que les tokens renvoyés des expressions régulières étaient bien associés dans l'analyse syntaxique.

Nous avons utilisés les fichiers de tests pour s'assurer que cela fonctionnait correctement. Une fois que l'ensemble des fichiers corrects fonctionnait nous avons considéré que notre syntaxe était correcte.

Enfin nous avons créé l'arbre de syntaxe abstraite grâce à cette structure :

```
1 struct node {
2   struct node *left;
3   struct node *right;
4   char *token;
5 };
```

Cette structure est une structure classique d'arbre binaire. Nous ne savions pas comment faire dans les règles où il y a plus de deux non-terminaux. Nous avons donc pris une certaine liberté sur la construction de l'arbre pour que celui-ci puisse lire la syntaxe de Little Ada. Par exemple la boucle `for` s'écrit : Une

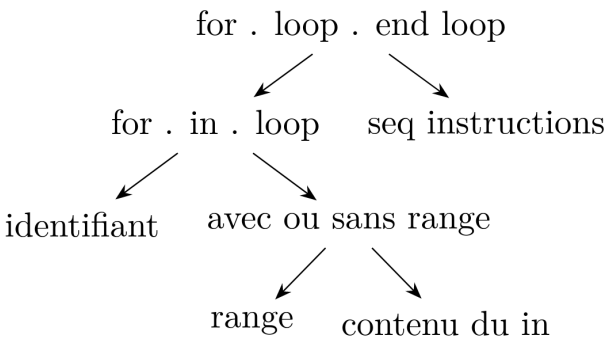

FIGURE 1 – AST pour la boucle for sans identifiant

fois l'arbre défini nous pouvions implanté les fonctions dans la troisième partie du fichier `little_ada.y` en prenant comme argument la racine de l'arbre.

3.3 Question 3

Pour afficher toutes les constantes, il nous suffisait de parcourir récursivement l'arbre et dès que l'on rencontrait le mot clé *"constante"* (nos noeuds étant nommés ainsi), il nous fallait print sa valeur, que l'on pouvait retrouver grâce

à `printf("\%s, ", tree->left->token);` puisque l'on stocke la valeur d'une constante dans le fils gauche du noeud par défaut.

3.4 Question 4

L'idée pour cette question était d'ajouter les constantes et les paramètres vus dans une liste chaînée au fur et à mesure, directement dans les actions de little_ada.y. Il suffisait alors seulement de compter les occurrences de chaque identifiant de constante ou de paramètre *in* dans cette liste : si un identifiant apparaissait strictement plus d'une fois dans la liste, c'était qu'un Warning devait être soulevé. Toutes les fonctions nécessaires à l'utilisation des listes ainsi qu'à la recherche des éléments sont définies dans little_ada.y. Une ébauche de cette fonction a été réalisée mais le soucis était que nous n'arrivions pas à récupérer toutes les constantes et tous les paramètres dans cette liste pour une raison que nous n'avons pas eu le temps de trouver.

4 Limites du projet

4.1 Délai

Nous n'avons pas réussi à terminer le projet. Nous avons perdu du temps sur l'implantation du projet en C. Nous avions fait le TP en Ocaml. Il aurait donc sûrement été judicieux de réaliser le projet en OCaml. Notre connaissance de flex/bison (quoique proche de ocamllex/ocamlyacc) était restreinte. La lecture de la documentation nous pris beaucoup de temps.

4.2 Syntaxe

Lors de la compilation nous avons eu 84 conflits par décalage/réduction. Ceux-ci sont notamment liés aux non-terminaux récursifs et ceux utilisant `%empty` et une autre règle. Par ailleurs, nous obtenons des `segmentation fault` lors de l'execution de notre programme sur les tests KO puisque nous essayons d'afficher les constantes alors que le fichier comporte une erreur syntaxique : l'AST sortant de notre programme n'est donc pas valide pour la fonction `print_consts`. Il faudrait ici tester la validité d'un AST avant d'effectuer des opérations dessus.