Université Gustave Eiffel

License Informatique

Rapport Technique

Analyse Syntaxique

3ème Année

Décembre 2020

Auteurs

- Jimmy TEILLARD
- Nicolas Atrax

Contents

1 Introduction

1.1 Français

Ce rapport technique a pour but principal de décrire le projet suivant ; c'est à dire le mode d'emploi du programme fourni, nos choix d'implémentation ainsi que les difficultés que nous avons pu rencontrer au cours du développement.

Le but de ce projet est de créer un analyseur syntaxique capable d'analyser un subset (sous-groupe) du langage de programmation C, et d'en retourner un rapport d'erreur syntaxique complet (numéro de ligne d'erreur, et pointage de ladite erreur).

1.2 Anglais

This technical report's goal is to describe the following project ; namely, a guide on how to use the given program, as well as our implementation choices and the diffulties we had to face during the development.

The goal of this project was to create a syntax parser (syntax lexer) that is able to analyse a subset of the C programming language, and return a complete report of the encountered syntax errors (line number, actual error on the line).

2 Mode d'emploi

2.1 Structure

- Les sources du programme se situent dans le dossier 'src' :
  - as.l contient la syntaxe des différents lexèmes de la grammaire
  - as.y contient les différentes règles définies par la grammaire
  - make est un executable permettant de compiler le programme
  - buildandtest.sh est un executable permettant d'effectuer une série de tests automatiquement
- Les tests sont contenus dans les dossiers 'test/valid' et 'test/invalid'
- Ce document est contenu dans le dossier 'doc'
- Un README.md est présent à la racine du projet, étant un résumé très succin de ce document

2.2 Compilation et Lancement

Après vous être placé dans le dossier 'src', les commandes suivantes vous permettront de manipuler le programme :

```
# pour compiler l'analyseur syntaxique
make
# pour recevoir un rapport d'erreur sur un fichier
./as < test.tpc
# pour nettoyer le dossier
make clean
```

2.3 Tests automatiques

Il est **nécessaire et impératif** de placer les fichiers à tester dans les dossiers 'test/valid' et 'test/invalid' en conséquence – c'est à dire que les programmes valides vont dans valid, et les programmes volontairement invalides vont dans invalid –, et doivent avoir pour extention '.tpc'. Si ces conditions ne sont pas respectées pour un fichier, alors il ne sera pas analysé.

Une fois dans le dossier 'src', lancer la commande `./buildandtest.sh` lancera les tests automatiques en plus de stocker le rapport complet dans un fichier 'testoutputs.txt'. Un test valide bien valide, ou un test invalide bien invalide, apporte 1 point (0 dans le cas contraire) sur le score de test final.

3 Développement

3.1 Choix

Le premier choix important qui nous a été donné de faire concernait la façon dont nous allions gérer les commentaires dans le code. Différentes possibilités s'offraient à nous, mais nous avons choisi d'utiliser les condition de démarrage de lex. De cette façon, nous étions sur que, pour le futur du projet, aucune inconsistance ne surviendrait puisque la gestion des commentaires est complètement séparée des autres lexèmes.

Afin de compter les lignes parcourues, nous avions le choix de créer notre propre variable globale dans le lex pour la passer au yacc... mais ce n'est pas ce que nous avons fait car la librairie offre déjà une option qui effectue précisément cette action : yylineno. Cependant le comptage des caractères jusqu'a l'erreur s'est fait au moyen d'une variable globale (errcharno) incrémentée à la lecture de chaque lexème, et réinitialisée à chaque retour à la ligne (cf. Difficultés).

Pour récupérer le contenu de la ligne où se situe l'erreur, nous avons choisi d'utiliser la commande REJECT, en plus de copier dans une variable globale le contenu de yytext grâce à l'expression régulière suivante : `^.*/` qui signifie que l'on cherche la prochaine fin de ligne sans récupérer le retour lui même (afin de ne pas incrémenter yylineno sur le rejet).

L'un des conflits présents dans la grammaire fournie est souvent appelé "Dangling ELSE" (parfois traduit par "Problème du sinon-pendant"). La résolution de ce conflit s'est donc faite en suivant un algorithme de factorisation simple (cf. `https://en.wikipedia.org/wiki/Dangling_else`).

3.2 Difficultés

La deuxième partie nous a pris beaucoup plus de temps. En effet, même si l'ajout à la grammaire du support des types structurés (struct) était relativement simple, régler les conflits engendrés l'était beaucoup moins. Boucher un conflit en créait rapidement un autre ; ce qui nous a amené à revoir la structure globale de la grammaire, notamment sur la gestion des types (avec TypeName) par exemple, et une factorisation des différentes règles afin de faire remonter le token STRUCT au plus haut possible par rapport au point d'entrée de la grammaire.

Une autre difficulté mineure rencontrée a été la gestion de la position exacte de l'erreur (position du lexème incorrect). En effet il nous a fallu beaucoup d'essais et de tests afin de comprendre pourquoi la flèche était toujours décalée de quelques caractères. Nous nous sommes finalement rendus compte qu'il fallait aussi soustraire le mauvais lexème au compte, en plus de compter les espaces avant celui-ci. Il reste cependant encore quelques imprécisions pour lesquelles nous n'avons pas pu trouver de solution.

4 Conclusion

Ce projet a pour nous été un excellent moyen de nous familiariser avec les outils lex (flex) et yacc (bison), en plus de mettre en pratique ce que nous avons appris sur les grammaires. En effet, malgré les difficultés rencontrées, nous sommes parvenus à trouver des solutions au fur et à mesure, tout en nous rendent compte des choses que nous n'avions pas comprises sans le savoir.