Rapport de Seconde soutenance

A2C

Team MALT
Promo 2018

Thibaud "zehir" Michaud
Lucien "Luciano" Boillod
Charles "Arys" Yaiche
Maxime "Kylox" Gaudron

6 avril 2015

Table des matières

Introduction

Le projet A2C est un compilateur du langage Algo vers le langage C, qui a pour but de faciliter leur enseignement dans les classes préparatoires d'EPITA. La team MALT s'est assez naturellement formée avant le début du quatrième semestre, étant tous redoublant dû au semestre international. Notre projet étant assez ambitieux pour des étudiants de SPE, la phase de documentation a été une étape importante et nécessaire avant toute implémentation. On a donc emprunté des ouvrages de référence dans le domaine comme "Modern Compiler Implementation" ou encore "Compiler : Principles, Techniques and Tools (aka Dragon Book) afin d'acquérir une compréhension générale sur les compilateurs et la théorie des langages. Deux d'entre nous ont d'ailleurs eu la chance d'assister aux cours de THL d'ING1 durant leur stage au LRDE du S3, ce qui leur a permis d'assimiler les premières notions importantes.

Lors de la première soutenance nous avons décidé d'adopter une technique efficace : travailler sur un sous langage du langage algo afin de finaliser les premières tâches nécessaires au bon avancement du projet (lexer, parser ...). Cette stratégie s'est révélée payante puisque nous avons pu avancer toutes les tâches en parallèle. Ainsi lors de la seconde soutenance il fallait aggrandir le langage pour arriver à compiler l'intégralité du langage Algorithmique. L'analyse syntaxique était également une grosse partie pour la seconde soutenance puisqu'il commencait a peine a emmerger lors de la S1.

Le présent rapport decrit le travail effectué depuis la soutenance précédente, ainsi que les difficultés recontrées et les rappels des objectifs.

Rappel des Objectifs

Grammaires

La grammaire devait avancer en même temps que le parseur, puisque que nous avions quelque chose de fonctionnel sur un sous langage, l'objectif était donc d'aggrandir ce langage.

Parseur

Il fallait aggrandir le langage, en ajoutant les boucles, les paramètres, les fonctions avec une valeur de retour, l'en tête de fonction, gestion de plusieurs fonctions dans un même fichier ...

Analyse sémantique

Etant donné que le parseur était presque fini, il restait surtout la phase d'analyse sémantique à compléter. Voici une liste des points qui devaient être implémenter pour cette soutenance :

- finaliser l'analyse de type pour les structures plus complexe que les expressions arithmétiques.
- conception de la table des symboles.
- lier la définition d'une variable à son utilisation : il faut vérifier que l'utilisation d'une variable correspond à sa définition (et vérifier qu'elle a bien été définie), que l'on ne déclare pas plusieurs fois la même variable, etc.
- vérifier que les expressions dans les conditions des structures de contrôle sont bien des expressions booléennes.

Génération de code

Pour la génération de code, pour l'instant nous laissons le choix à l'utilisateur quant à l'utilisation du code généré, mais bien sûr l'objectif est de

sortir proprement un fichier C qui peut être compilé à la main ou de manière automatique. De plus et tout comme les autres tâches, il devait suivre l'avancement du langage et générer le code équivalent en C, tout en assurant une indentation lisible et cohérente.

site web

Des news devaient être postées de manière régulières et le logo devait être créé et ajouté au site.

Organisation

2.1 Organisation du groupe

Nous avons gardé la même organisation depuis le début. Une réunion par semaine : un rappel de ce qui a été fait depuis la dernière réunion, un échange et une explication si nécessaire de nos parties, un point sur ce qui reste à faire avec l'écriture du TODO. Ce mode de fonctionnement était très efficace puisque à l'issu de la réunion chacun repartait avec un objectif pour la semaine d'après.

2.2 Organisation du projet

2.2.1 Chronologie

La chronologie définie dans le cahier des charges a du être revue au cours de la première période. Nous avions segmenté le projet en trop gros blocs : parseur, génération de code, etc. et nous nous sommes rendu compte que nous ne pourrions pas travailler comme ça car tant qu'une tâche n'est pas finie, les autres sont bloquées et ne peuvent pas avancer.

Nous sommes donc repartis dans l'idée de commencer par un langage très simple constitué uniquement d'expressions arithmétiques, et de travailler sur toute les parties du projet en même temps à partir de ce langage. L'idée étant ensuite d'étendre le langage au fur et à mesure jusqu'à atteindre le langage algo.

Nous avons ainsi pu étoffer le langage très vite et arriver à l'intégralité du langage algo lors de cette deuxième soutenance en avançant toutes les tâches en même temps.

Travail Effectué

3.1 Lexeur/Parseur

3.1.1 Nouvelles features

Le parseur est maintenant fini. Voici une liste des éléments du langage qui ont étés rajoutés depuis la dernière soutenance :

— structure conditionnelle "selon ... fin selon"
— gestion des erreurs syntaxiques
— déclaration de constantes
— déclaration de types
— déclaration de paramètres locaux et globaux
— déclaration de plusieurs algorithmes dans le même fichier

Le lexeur a aussi été complété sans problème pour suivre l'avancement du parseur.

3.1.2 Conflits shift/reduce

Tous les conflits shift/reduce ont étés corrigés, via l'ajout de règles de précédence ou en modifiant un peu la grammaire. Nous avons eu un peu de mal pour les déclarations, car celles-ci peuvent apparaître dans n'importe quel ordre. Et comme chaque type de déclaration (constantes, variables, ...) est potentiellement vide, cela créé un grand nombre de possibilités avec de nombreuses ambiguités. Nous n'avons pour l'instant pas trouvé d'autre moyen que de lister tous les cas possibles (16 cas en tout).

Un autre problème a été la présence de plusieurs ambiguités qui semblaient inhérentes au langage. Pour lever ces ambiguités, il a fallu rajouter explicitement les retours à la ligne dans la grammaire du langage.

3.1.3 Gestion des erreurs syntaxique

La gestion des erreurs est une phase importante dans l'ecriture d'un compilateur. C'est ce qui permet à l'utilisateur de bien écrire son code. Dans A2C, la gestions des erreurs est pratiquement terminée. dans Bison, la gestion des erreurs syntaxiques se fait comme ceci : lors du parsing, Bison repère qu'il y a une erreur de syntaxe, un "alors" alors que il attend une ")" par exemple ; à ce moment là, Bison va détecter l'erreur et arreter l'analyse. Notre objectif premier était de pouvoir détecter plusieurs erreurs. Pour cela nous avons utilisé le token "error" de bison, qui permet, lors de la detection d'une erreur sur un token, d'arreter le parsing pour aller au token suivant.

Dans un fichier algo plein d'erreurs, lors de la compilation, le parseur va gérer toutes les erreurs, les afficher et arreter la génération du code C :

nom_du_fichier.algo :(ligne de l'erreur) : message d'erreurs, où le token à fail
nom_du_fichier.algo :(ligne de l'erreur2) : message d'erreurs2, où le token à fail ...

3.1.4 Numéros de ligne

Nous avons aussi aujouté les numéros de ligne dans les noeuds de l'arbre syntaxique abstrait. Nous sommes donc capable d'afficher des messages d'erreur plus explicites, notamment pour la phase d'analyse sémantique. Nous avons pour cela utilisé l'option yylineno de flex.

Avec cette option, flex créé une variable globale appelée yylineno de type entier. Cette variable est mise à jour au fur et à mesure de la phase d'analyse lexicale. Ainsi, lorsque bison est capable de réduire une règle, il suffit de récupérer yylineno et de le mettre dans le noeud correspondant à la règle (appelé $$ dans bison).

Lorsqu'une erreur est détectée au cours de l'analyse sémantique, on peut donc afficher le numéro de ligne contenant l'erreur, ainsi que la ligne elle même.

Pour la prochaine soutenance, nous avons prévu de gérer une fourchette de lignes (lorsqu'une erreur n'est pas localisée sur une ligne précise) ainsi qu'une précision au caractère près.

3.2 Type

Avant l'analyse de type il nous a fallu pouvoir decrire ces types, qui ils etaient et comment ils évoluaient. Pour cela nous somme parties des types que nous avions déjà à la soutenance 1. On les nommera les types primaires et qui sont donc composés des types.

- entier
- caractere
- chaine de caractere
- booleen
- reel
- nul

Ces types nous ont donc ensuite servis pour decrire les types plus composés que nous allons vous présenter :

3.2.1 Les enums

Les enums ne sont qu'une liste d'identifiant, rien de bien compliquer a creer. On les stockera directement dans la table des types associer à l'enum.

3.2.2 Les pointeurs

Le type pointeur n'est pas non plus bien compliquer a representer. Il s'agit d'un identifiant associé à un type pointeur.

3.2.3 Les enregistrements

Les enregistrements sont légèrements plus complexes. Nous les avons definis par une liste de field qui associes a leurs tour un identifiant et un type.

3.2.4 Les tableaux

Les tableaux sont composés d'un type déjâ existant et une liste d'entier pour les tailles de tableau. dans le cas d'un matrice d'entier de 2 par 2 nous aurons le type : entier(2,2).

3.3 Tables des symboles

Chaque table des symboles ne travail pas directement avec les éléments de l'ast mais par une représentation annexe plus facile d'utilisation. Nous utilisons une table de hachage pour representer ces symboles. Nous avons donc decidé de hascher nos elements par rapport a leur identifiant qui sont unique. pour chaque table nous avons donc en clef un identifiant et comme valeur le symbole associé à la table.

3.3.1 Table des functions

La table des fonctions utilise comme symbole des fonctions qui sont representé par un type de retour, une liste de type représentant les arguments et d'un identifiant qui est le nom de la fonction.

3.3.2 Table des variables

La table des variables correspond à toutes les déclarations (en dehors des types) que l'on aurait pu faire dans l'entête des algorithmes, chaque algorithme possède sa propre table de symbole. Les variables global sont présente dans toute les tables de variables.

3.3.3 Tables des types

La table des types références tout les types qui ont était créés pour l'algorithme, les types primaires sont directement intégrés dans la table sans avoir besoin de les créer et de les ajouter.

3.3.4 Evolution

Pour l'instant toute nos données sont représenté dans 3 tableaux mais avant la troisième soutenance un gros travail de refactorisation de code va etre éxécuter sur l'ensemble du projet. Les tables ne seront alors plus représenter que par une seule structure qui suivant ce que l'on cherche utilisera une seule des trois tables ca évitera de passer en paramètre les trois tables dans la plus part de nos fonctions.

3.4 L'analyse de type

3.4.1 Introduction

L'analyse de type est la partie du compilateur qui a le plus avancer. Elle se traite de maniere descendante c'est a dire que l'on traite le programme on entier en parcourant l'ensemble des algorithmes afin de recuperer leurs caracteristiques. Puis diverse verification sont effectuees sur ces algorithme que nous allons detailler dans cette partie.

3.4.2 Le programme

Pour le programme comme dit precedemment on parcour la liste des algos du programme pour en recuperer leurs noms, leurs arguments et leurs type de retour. Toute ces informations sont ensuite stocker dans la table des fonctions en fonctions de leur nom.

3.4.3 Les algo

Une fois que tout les algos sont enregistrer on parcour les algo plus en detaille en commencant par leurs declarations.

Les types

Dans la declaration de type on remplis prealablement la table des types avec les types primaire (cf section type). Le reste des types ne sera que des compositions de type deja definis et stocker dans la table des types.

Les autres

Le reste des declarations et des parametres sera stocker dans une seule et meme table des symboles : la table des variables. Ils suffit donc de parcourer l'ensemble de ces declarations et verifier que le type existe bien et que l'element n'est pas deja existant dans la table pour eviter les multi-declarations.

3.4.4 Les instructions

Une fois que les entetes des algos sont bien verifier et que tout est stocker dans une table de symbole il faut verifier que les algos sont syntaxiquement correct pour cela on parcours l'ensemble des instructions de l'algorithme en verifiant toutes les expressions de ces instructions.

les instructions peuvent etre de different type :

Les assignations

pour les assignement il nous suffit de verifier que les exrepssions a droite et a gauche sont bien correcte puis de verifier que leur type est bien egale

```
faire
        [insts]
tant que [expr]
```

Boucle pour

Pour les boucles pour on verifie que expr1 et exrp2 dans : sont correcte et sont de meme type ils suffits ensuite de verifier chaque instruction dans la boucle pour.

```
pour [expr1] jusqu'a [expr2] faire
 [insts]
fin pour
```

Selon

Dans le cas du Selon il faut verifier que toutes les [expr] soit du meme type et valide. Une fois verifier nous verifions que l'ensemble des instructions dans chaque case sont valident aussi.

```
selon [expr] faire
        [expr]: [instrs]
        autrement: [instrs]
fin selon
```

Faire tant que et tant que faire

La gestion de ces instructions est plutot semblable on verifie que l'expression dans le tant que est bien de type booleen puis on verifie ensuite l'ensemble des instructions.

```
faire
        [insts]
tant que [expr]
```

```
tant que [expr] faire
        [inst]
fin tant que
```

Appel de fonction

On verifie uniquement que la fonction est bien presente dans la table des fonctions, que le nombre d'argument est bien le meme que celui dans la table des fonctions, et que le type des arguments correspond egalement.

Condition Si

```
si [expr] alors
        [insts]
sinon
        [insts]
fin si
```

Retourne de fonction

On verifie que l'expression dans le retourne est bien de meme type que le type de la fonction dans laquelle nous nous trouvons.

```
retourne [expr]
```

Lorsque nous avons detecter l'instruction la verification des expression composant cette instructions peut etre lancer.

3.4.5 les expressions

valtype

Pour les expressions directe nous n'avons rien besoin de verifier l'expression est de base correcte

binop

Les binops n'ayant pas etait modifier depuis la soutenance 1. leur verification est toujours semblable. On verifie que l'operation a droite et a gauche de l'operateur sont bien de meme type. Une seul modification a etait apporter dans le cas d'operateur booleen on force le type de l'operateur au type booleen.

array expr

Pour les expression de type tableau il y a plusieurs verification a faire. Il faut verifier que si l'on tente d'acceder a une case du tableau celle-ci est bien dans la taille du tableau mais que l'expression avec la quelque on tente d'atteindre la case du tableau est bien de type entier. tab[1] est ok tab[a+b] est ok avec a et b entier tab[c] avec c de type booleen est impossible.

Une fois ceci verifier il faut aller recuperer le type des elements stocker dans le tableau dans le cas d'une matrice d'entier noter tab on aura : tab est de type tableau de tableau d'entier, tab[1] qui doit renvoyer le type tableau d'entier tandis que tab[1][2] renverra un type entier.

identtype

Pour les identypes ils nous suffit uniquement d'aller chercher dans la table des variables voir si il est present de retourner son type.

struct elt

Les structures forment en quelque sorte un nouveau type comme les tableau si on utilise directement la structure il suffit de retourner son type qui est dans la table des types de meme si l'on utilise un des champs de la structure il faut verifier que le champ existe et qu'il a bien un type correspondant dans la table des types.

dereftype

les expressions de dereferencement s'utilise avec les pointeurs qui ne sont, comme nous l'avons vu dans la partie sur les types que des pointeur vers une type. Il nous suffit donc d'uniquement verifier si le type de la variable pointer est bien present dans la table des symbole et de le retourner.

3.5 Génération de code

3.5.1 Avancement

La génération de code est allée beaucoup plus vite que prévue. En effet, la syntaxe du langage algo étant assez proche du langage C, il suffit de parcourir l'arbre syntaxique abstrait et d'afficher le code C correspondant.

3.5.2 Types et fonctions prédéfinies

Il y a cependant une difficulté en ce qui concerne les types et fonctions prédéfinis. Pour les types, nous avons juste écrit une fonction qui convertit un type primaire du langage algo vers le C (e.g. entier -> int). Pour les fonctions, c'est un peu plus délicat puisque les fonctions algo et C ne s'appellent pas forcément de la même manière (e.g. allouer() et malloc(), ou bien écrire() et printf()).

3.5.3 Ce qui reste à faire

Il va donc falloir gérer les symboles prédéfinis au cas par cas et faire appel au travail de l'analyse de type pour déterminer la bonne traduction. Cela nécessitera un peu de restructuration car pour l'instant, les types des expressions ne sont enregistrées nulle part (sauf les identifiants qui sont enregistrés dans les tables de symboles).

Rappels

3.1 Respect du cahier des charges

Rappel du cahier des Charges

	deuxième soutenance
site web	done
structure de données	done
lexer	done
construction grammaire	done
parseur	done
analyse sémantique	74.99%
génération de code	90%

— site web : terminé et en ligne
— structure de données : toutes les structures nécessaires au projet ont été implémentés. Cependant si besoin est, d'autres pourront être ajoutéets.
— lexeur complètement terminé et fonctionnel
— construction de la grammaire : la grammaire abstraite est terminée (elle n'a pas d'autre intêret que de facilité la compréhension), la grammaire implémenté est également terminé en accordance avec le parseur.
— parseur : terminé et fonctionnel
— analyse sémantique : les 3/4 de l'analyse sémantique ont été terminé.
— génération de code : 90% de la génération de code a été modifié.

Les objectifs ont donc été parfaitement remplis, voir bien plus. Le projet est pratiquement terminé, il reste une partie de l'analyse sémantique ainsi que génération de code afin de finir le projet et de pouvoir s'amuser à implémenter divers bonus et surprises.

3.2 Avance par rapport au cahier des charges

conclusion des tâches en avance :
— generation de code (+)
— analyse sémantique (+++)

3.3 Ce qui reste à faire

3.3.1 soutenance finale

Everything has to be done

	dernière soutenance
site web	done
structure de données	done
lexer	done
construction grammaire	done
parseur	done
analyse sémantique	done
génération de code	done

Biensur tout doit être fini dans ce qui était prévu, mais grâce à l'avance acquise de par notre rigueur de travail, nous pouvons espèrer implémenter des bonus et des surprises en tout genre.

Conclusion

Nous avons fini le parseur, et la génération de code et l'analyse de type sont presque terminées. Il ne reste donc plus qu'à faire une gestion d'erreurs plus précise et plus complète.

Dans l'ensemble le projet avance bien et même bien plus que nos espèrances, nous en sommes très content. Il y a une bonne ambiance dans le groupe, les réunions hebdomadaires nous permettent d'avancer régulièrement et nous sommes confiant pour la dernière soutenance.

Nous espèrons tenir bon dans notre sérieu et notre rigueur pour la dernière ligne droite et étonner le Jury avec ce projet qui nous tient à coeur.

Sources

Livres

Compiler : Principles, Techniques and Tools (Dragon Book)
Modern Compiler Implementation in ML

Website

http ://www.google.com (meilleur ami)
http ://stackoverflow.com
http ://llvm.org

Autre

Tiger Project d'EPITA (ING1)
Interpréteur de Marwan