Université Côte d'Azur

Département d'informatique

Compilation

Rapport projet de compilation

Élèves :
Arnaud DUMANOIS
Laurène FEA

Enseignant
Sid Touati
Cinzia Di Giusto

Table des matières

Table des figures

1 Avant-propos

Avant de commencer le cœur de ce rapport, une petite précision : la majeure partie de notre inspiration nous est venu durant les différentes session de td où nous avons été assidus durant toute la durée du semestre.

2 Frontend

Cette partie du compilateur strucit traite du langage en entrée STRUCIT-frontend. Nous y détaillerons ici la manière dont nous avons traité l'analyse lexico-syntaxique et sémantique.

2.1 Lex

La première étape de ce projet a été la création d'un parseur qui nous permette de reconnaitre si un fichier en entrée était lexicalement du STRUCIT-frontend. Pour se faire, nous avons utilisé le fichier fournis ANSI que nous avons modifié pour permettre la reconnaissance des commentaires et des mots-clés spécifiques du langage en entrée.

2.2 Modification apportée à la grammaire de structfe.y

Une fois le parseur réalisé et fonctionnel, nous avons commencé à nous imprégner de la grammaire proposée par le fichier fournis dans l'archive structfe.y. Cette dernière est une grammaire ascendante avec des conflits au niveau de la règle de grammaire IF ... ELSE. Donc nous avons d'une part géré ce conflit grâce à une règle de priorité %prec et un nouveau token ELSE de Yacc puis avons posé des priorités sur les opérateurs logiques et binaires avec les règles %right et pour finir ajouter un token MOINS afin de gérer le problème du « - » unaire.

2.3 Structure de données

Pour mener à bien ce projet, nous avons dû dans un premier temps réfléchir aux structures de données à utiliser et de quelle manière nous allions gérer la gestion des identifiants, la génération de code et la gestion de type. Ainsi pour commencer, nous avons fabriquer une table de hachage, basée sur le modèle vu en TD, qui nous servira de structure de donnée pour la table des symboles. Cette table de symbole permettra de garder en mémoire les identifiants du fichier en entrée. Les symboles sont représentés par un type défini, toujours inspiré par le TD, nommé « symbole_t ». Il contient 2 champs : nom, une chaine de caractères et type, un « type_t » définit ci-dessous.

Dans le même temps, nous avons définis les types que nous allions utiliser pour la gestion dans un autre type énuméré défini, cette fois nommé « type_t » contenant les valeurs « ENTIER ; VIDE ; POINTEUR ; EXTERN ; STRUCTURE » contenu dans le même fichier que la table de hachage.

2.4 Génération de code

2.4.1 Introduction et généralités

Une fois la partie des types et des identifiants terminés, nous nous sommes penchés sur la plus importante partie du projet qui nous a pris la majeure partie de notre temps : la génération de code. Pendant de longues semaines, nous avons réfléchis sous quel angle attaquer le problème mais sans jamais être sûr de notre implémentation. De ce fait, afin d'être plus productif, nous avons décidé d'attendre la séance de TD traitant du sujet pour avoir une idée d'implémentation et en attendant, nous avons au papier, et sous forme de schéma, élaborés les différents squelettes de traduction de la grammaire.
Les squelettes sont disponibles en annexe.

Une fois les séances de TD passées, nous avons décidé de reprendre les idées de ces séances pour notre projet. Ainsi dans un fichier nommé « generation_code.c », nous avons implémenté les fonctions vues au TD ; c'est-à-dire :

- « init_code() » qui initialise une chaine de caractère à vide
- « inserer_code(c1,c2) » qui concatène les chaines de code c1 et c2 et nous permet d'ajouter du code au fichier de sortie
- « new_label() » qui génère des étiquettes pour créer les branchements de certain squelette
- « new_var() » qui permet de créer des variables temporaires pour respecter le format 3 adresses de STRUCIT-backend

Nous avons aussi créé un fichier « type.h » qui contient un type défini « all_types » qui sera le type utilisé pour les routines sémantiques de toutes les règles de grammaire du frontend et que nous manipulerons pour la génération de code que nous allons expliquer dans la partie suivante.

Afin de générer du code nous avons défini dans « all_type » une chaîne de caractères nommée « code ». Cette dernière sera utilisée de pair avec les fonctions « inserer_code » et « init_code » afin de former une grande chaîne de caractères contenant tout le code généré par le compilateur.

Après de longue réflexion et de nombreux tests, nous avons pour chaque règle de grammaire du frontend inséré le code correspondant à sa traduction en backend puis avons écrit cette chaîne de caractère dans un fichier de sortie pour y afficher le code généré.

Nous allons maintenant vous décrire certain point intéressant de notre génération de code.

2.4.2 Format 3 adresses et variables temporaires

Commençons par expliquer la gestion des variables temporaires et des déclarations pour transformer le code frontend en 3 adresses.

A chaque règle pouvant nécessiter de la création de nouvelles variables, comme pour les règles du « + », « * », « / », « - », nous avons créé une variable grâce à la fonction

« new_var() ». A chaque création d'une variable, nous concaténons le mot-clé « void » et le nom de la variable nouvellement crée dans une chaine de caractère du nom de « decla_tmp » présent dans « all_types ». Cette chaîne de caractère sert à concaténer toutes les déclarations de variables afin de les insérer dans le code au début de la fonction concernée.

Pour les affectations de ces variables nouvelles, elles sont concaténées dans une chaine de caractère sur le même modèle que pour les déclarations nommées « temp ». Cette chaine de caractère sera ajoutée au code après les déclarations de variables. Au niveau de la règle sémantique, on ajoute alors seulement le nom de la nouvelle variable dans le code, qui représente alors le calcul déclaré au-dessus.

2.4.3 Gestion des boucles et condition : cas du IF-ELSE

Nous allons maintenant expliquer comment fonctionne le IF-ELSE. Pour ce faire, nous allons nous appuyer sur les squelettes tracés ci-dessous.

Dans le cas où le statement2 contient un return

```
IF (expression) statement1 ELSE statement2
expression goto etiq1 ;
statement2
etiq1 :
  statement1
```

Sinon

```
IF (expression) statement1 ELSE statement2
expression goto etiq1 ;
statement2
goto etiq2 ;
etiq1 :
  statement1
etiq2 :
  ...
```

Tout d'abord, nous avons remarqué deux comportements différents.

Nous avons un cas où le branchement else contient un return ; dans ce cas-là, la fonction s'arrête, sans soucis. Dans le deuxième cas, le branchement else ne contient pas de return ; il faut alors sauter la partie du if grâce à un goto.

Détaillons le fonctionnement de ce deuxième cas. Si la condition est juste alors nous allons à l'étiquette 1, qui contient l'action à faire dans ce cas-là. Si la condition est fausse, nous lisons la suite du code, et effectuons donc la partie du else. Une fois le else fini, il faut alors aller à l'étiquette 2 qui se trouve après l'action effectué par la branche du if. Cette étiquette nous amène à la fin de cette partie et nous permet de ne pas lire la partie du if.

Pour les autres branchements, boucles et conditions, nous ne détaillerons pas le fonctionnement, très proche de celui décrit ci-dessus, mais nous fournissons des squelettes explicites.

3 Backend

Cette seconde partie du compilateur traite cette fois du langage de sortie STRUCIT-backend. Cette fois-ci nous nous étalerons moins sur les détails car peu de chose y ont été

effectuer.

3.1 Lex

Comme dit en 2-1 cela a été la première partie de ce projet. Après lecture et compréhension de la grammaire de STRUCIT-backend, qui est légèrement moins naturelle que ce que nous avons eu l'habitude d'étudier, nous avons ensuite modifié en fonction le fichier ANSI fourni et vérifié sa fiabilité.

Une petite subtilité est a noté dans notre compilateur à ce niveau. Ce dernier reconnaitra des commentaires du type /*toto */ mais ne les écrira pas dans le fichier de sortie. Ainsi les commentaires ne génèreront pas d'erreur lexicale si on test notre parseur de backend sur un fichier en contenant mais notre compilateur ne généra pas de commentaire.

3.2 Modification apportée à la grammaire de structbe.y

Nous avons remarqué en testant nos fichiers que le parseur du backend ne reconnaissait pas les signes « < », « > ». Après vérification de notre lex, nous nous sommes rendu compte que le problème venait de la grammaire. En effet dans expression, nous avions la règle : « primary_expression = additive_expression », ce qui fait que les signes « < » et « > » se trouvant plus loin dans la grammaire, au niveau de relational_expression, il n'était pas pris en compte. Nous avons donc changé pour « primary_expression = relational_expression ».

4 Répartition du travail

Pour ce projet, nous jugeons personnellement la répartition du travail correcte et équitable.

Pour Laurène, sa partie consistait notamment en l'implémentation des structures de données, du parseur pour le frontend, du changement dans la grammaire du backend.

Parallèlement, Arnaud a géré le conflit IF-ELSE de la grammaire du frontend, le parseur du backend et à commencer à poser les bases de la génération de code.

Par la suite, notre binôme s'est partagé équitablement les différentes routines sémantiques.
Nous sommes très satisfaits de notre collaboration.

5 Critique de notre projet

5.1 Gestion de type

Cette partie de notre compilateur est à notre avis celle qui laisse le plus le droit à l'erreur. En effet nous n'avons pas utilisé une méthode judicieuse pour vérifier le typage de programme contenant des fonctions. Notre gestionnaire de type remonte les branches

de l'arbre de dérivation des types pour les vérifier mais n'arrive pas à prendre en compte des situations complexes telles que des pointeurs sur des fonctions ou alors le type de retour attendu a une instruction return.
Avec du recul il aurait sans doute été plus efficace et avantageux d'utiliser une structure de donnée comme un arbre.

5.2 Cas des pointeurs sur fonction

La génération de code a été la partie la plus longue et celle qui nous a demander le plus de réflexion, mais elle n'est pas parfaite. En effet sur tous les tests proposés, nous avons des résultats cohérents pour la plupart, sauf deux, qui contiennent des pointeurs sur des fonctions

Nous avons essayé de trouver solution à ce problème durant les dernières semaines mais sans résultat conséquent. Nous pensons que cela est peut-être due à une mauvaise compréhension de l'énoncé sur ce point ou alors d'une mauvaise implémentation a un endroit de notre code.

5.3 Problème lié au déclaration de variables

Pour déclarer des variables, nous utilisons comme vue en cours et dit en 2-3 une table de symbole. Cette table est fonctionnelle et marche dans notre projet mais nous avons détecter certains problèmes. En langage C, on peut déclarer des variables de même nom de façon global (en dehors de fonction) et local (paramètre de fonction ou bloc interne à la fonction). Le problème est que notre implémentation, basée sur une unique table de symbole, ne parvient pas à prendre en compte 2 variable du même nom déclaré de manière indépendante.

Sur ce point nous pensons qu'il aurait été peut-être plus astucieux de créer une pile de table de symbole ou alors une table de symbole pour les fonctions et leurs paramètres.

6 Conclusion

Notre compilateur strucit parvient à générer du code 3 adresses qui respecte globalement les consignes données par le projet et à vérifier s'il est syntaxiquement et lexicalement correct. Nous n'avons néanmoins pas généré du code parfait pour tous les tests et ne parvenons pas à gérer certain cas de figure.

Ce projet de Compilation a été pour nous deux un exercice assez long et complexe où nous avons dû utiliser de nombreuses connaissances acquises durant notre licence.

Pour nous, l'une des parties les plus dur dans un premier temps a été de s'approprier les outils d'analyse lex et yacc avec leur particularité. Une fois cette étape passée, l'élaboration d'un plan d'implémentation a aussi été une partie délicate et vaste. Mais a force d'acharnement et de travail, nous avons réussi à produire un résultat qui nous satisfait et nous rend fier.

Ce projet nous a permis de consolider des acquis tel que l'implémentation de structures de données et la programmation C (et même LaTeX) mais aussi de mieux comprendre par la pratique les concepts et idées du domaine de la compilation. Pour nous, ce projet a en effet était très formateur jusqu'à la fin.

7 Remerciement

Nous tenons tous les deux à remercier nos enseignant Cinzia DI GIUSTO et Sid TOUATI pour les réponse à nos différentes questions durant toute la durée de ce semestre.

8 Annexe

logical_or_expression OR_OP logical_and_expression

if logical_or_expression goto etiqT ;
goto etiqS ;
etiqS :
  if logical_and_expression goto etiqT ;
  goto etiqF ;
etiqT :
  0
  goto etiqFin ;
etiqF :
  1
etiqFin :
  ...

logical_and_expression AND_OP equality_expression

if logical_expression goto etiq1 ;
goto etiqF ;
etiq1 :
  if equality_expression goto etiqT ;
  goto etiqF ;
etiqT :
  0
  goto etiqE ;
etiqF :
  1
etiqE :
  ...

FOR (expression_statement1 expression_statement2 expression) statement

goto etiq1 ;
etiq2 :
  statement
  expression
etiq1 :
  if expression_statement2 goto etiq2 ;

IF (expression) statement

expression goto etiq1 ;
goto etiq2 ;
etiq1 :
  statement
etiq2 :
  ...

WHILE (expression) statement

goto etiq1 ;
etiq2 :
  statement
etiq1 :
  if expression goto etiq2 ;