Carré Matthias & Chantoiseau Sacha

PROJET DE COMPILATION

STRUCIT, UN MINI COMPILATEUR C

Strucit est un compilateur conçu pour traduire un dialecte du C vers un autre dialecte du C, plus proche du langage machine, en utilisant du code à trois adresses. Ce compilateur a été développé avec les outils Lex et Yacc dans le cadre du projet de l'UE Compilation de la troisième année de licence en Informatique. Le front-end du compilateur prend en charge l'analyse syntaxique et sémantique du code source, ainsi que la génération de code intermédiaire. Ensuite, le back-end s'assure que ce code intermédiaire, plus proche du langage machine, est valide.
Dans ce rapport, nous allons détailler notre approche, les différentes fonctionnalités et spécificités du compilateur, ainsi que ses limites.

I) FRONTEND

Nous allons vous détailler comment nous nous y sommes prit pour l'analyse lexicale, syntaxique et sémantique de la partie frontend.

1). MODIFICATIONS APPORTÉES AUX FICHIERS FOURNIS

Pour commencer, des fichiers initiaux étaient fournis avec le sujet, incluant les grammaires pour les parties frontend et backend (dans un descripteur Yacc), ainsi que les spécifications Lex pour le langage C (ANSI.l).

Premièrement, nous avons supprimé les éléments de ces fichiers qui n'étaient pas pertinents pour les dialectes, et ajouté la reconnaissance des commentaires dans la partie Lex. Nous avons également apporté quelques modifications au lex :

- Nous avons rajouté la gestion des commentaires par le biais de la ligne:

  "/*"([^*]|*+[^*/])**+"/"{printf("commentaire\n");}
- Nous avons rajouté la gestion des contantes et des identifiants
- L'ajout de nouvelles lignes a été géré pour incrémenter le numéro de la ligne
- Nous nous sommes occupés des gestions d'espaces et caractères non reconnus

Et pour le fichier yacc structfe.y aussi :

- Nous avons ajouté des actions pour les règles de grammaire pour la gestion des types et et vérification de la comptabilité
- Pour l'analyse syntaxique : Nous avons complété les règles de grammaire qui définissent les structures valides des programmes en vérifiant si la syntaxe est correcte
- Pour l'analyse sémantique : Pour chaque règle de grammaire nous faisons la vérification des types arithmétiques , logiques et de comparaison. La gestion des déclarations et des spécifications de types , avec insertion des symboles dans la table des symboles (dont nous parlerons plus tard) . Et la détection et le signalement des erreurs sémantiques.
- Pour la génération de code : Nous avons fait une génération de code 3 adresses sur les instructions de contrôles (FOR et IF) , "a compléter"
- Nous avons géré l'ambiguïté du IF...ELSE grâce à la règle de priorité Yacc %prec; pour enlever le problème de shift /reduce.

2). Fichiers Rajoutés

L'implémentation de notre compilateur a nécessité l'ajout de nouvelles structures de données, que nous avons définies et mises en œuvre dans deux fichiers distincts : table_symboles.h et table_symboles.c.

L'un des éléments essentiels au sein d'un compilateur est la table des symboles, qui permet de garder en mémoire les identifiants utilisés dans le fichier source. Pour cela, nous avons implémenté la table des symboles en utilisant une table de hachage avec une taille définie par la constante SIZE. Nous avons également géré la portée des symboles à l'aide d'une pile de listes chaînées, appelée Tas. Chaque élément de la pile représente un bloc de code imbriqué, et la table des symboles au sommet de la pile correspond au bloc le plus imbriqué.

Les symboles dans notre compilateur représentent les variables du programme source et sont constitués d'un nom et d'un type. Nous avons défini plusieurs types de symboles : TYPE_INT, TYPE_VOID, TYPE_STRUCT, et TYPE_ERROR. Dans la partie frontend, les types acceptés sont donc int, void, et struct.

Détails d'Implémentation

Fichier table_symboles.h

Le fichier table_symboles.h contient les définitions des structures et des fonctions nécessaires à la gestion de la table des symboles. Nous avons défini un type énuméré SymboleType pour représenter les différents types de symboles. Les symboles eux-mêmes sont représentés par la structure Symbole, qui contient un nom et un type. Pour stocker ces symboles, nous avons utilisé une liste chaînée (Node).

Nous avons également défini une structure LinkedList pour gérer les listes chaînées et une structure Tas pour gérer la pile de listes chaînées. Les principales fonctions définies dans ce fichier incluent la création et la suppression de nœuds, l'insertion et la recherche de symboles, ainsi que la mise à jour de symboles.

Fichier table_symboles.c

Le fichier table_symboles.c contient les implémentations des fonctions déclarées dans table_symboles.h. Voici quelques points clés :

- Création et Suppression de Nœuds : La fonction create_node alloue de la mémoire pour un nouveau nœud, initialise son nom et son type, et le retourne. La fonction delete_node libère la mémoire allouée pour un nœud.
- Fonction de Hachage : La fonction fonction_hash calcule l'index d'un symbole dans la table

- Fonction de Hachage : La fonction fonction_hash calcule l'index d'un symbole dans la table de hachage en utilisant la somme des valeurs ASCII des caractères du nom du symbole.
- Gestion de la Pile : Les fonctions initialize_tas, addinTas, popTas, et getTopTas gèrent la pile de listes chaînées. Ces fonctions permettent d'initialiser la pile, d'ajouter et de retirer des listes de symboles, et d'accéder à la liste de symboles au sommet de la pile.
- Insertion et Recherche de Symboles : Les fonctions insert_symbol, find_symbol, et update_symbol permettent d'insérer de nouveaux symboles, de rechercher des symboles existants, et de mettre à jour les types de symboles dans la table de hachage.
- Impression de la Table des Symboles : La fonction print_symbol_table affiche le contenu de la table des symboles, ce qui est utile pour le débogage et la vérification de la table des symboles pendant le développement.

Nous avons mis en œuvre ces structures et fonctions pour assurer une gestion efficace des symboles dans le compilateur, en garantissant que chaque symbole est correctement déclaré, typé, et accessible dans le bon contexte. Cette approche nous a permis de gérer les différents blocs de code et les portées des variables de manière claire et structurée.

3) Les squelettes

```
if(C) D1 ;else D2
goto cond X:
corp X:
D1
goto fin X
cond X:
if (C) goto corpX
D2
fin X:
```

```
while(A){B}
goto cond:
corps:
B
cond:
if(A) goto corps
```

```
for(A,B,C){D}
A
goto cond:
corps:
D
c
cond:
if B goto corps
```

Pour nous aider a la génération 3 adresses du code intermédiaire nous nous sommes aidés de ces squelettes notamment les squelettes de IF FOR et WHILe

II) BACKEND

Pour les modifications apportées : l'inclusion des bibliothèques nécessaires a été effectuée en ajoutant stdio.h, stdlib.h, et string.h pour gérer les entrées/sorties, la gestion de la mémoire, et les manipulations de chaînes de caractères. Une fonction yyerror a été définie pour afficher des messages d'erreur de syntaxe, ce qui facilite le débogage. Les tokens spécifiques ont été ajoutés, notamment IDENTIFIER, CONSTANT, LE_OP, GE_OP, EQ_OP, NE_OP, EXTERN, INT, VOID, IF, RETURN, GOTO, et COMMENT, représentant les éléments syntaxiques fondamentaux du langage STRUCIT-backend. La structure de départ de la grammaire a été déclarée, spécifiant que l'analyse syntaxique commence par external_declaration, correspondant à une déclaration externe (fonction ou variable). Les expressions primaires et postfixées ont été définies pour gérer les identificateurs, les constantes, et les appels de fonctions, permettant de construire les expressions de base du langage. Les expressions unaires et multiplicatives ont été ajoutées pour permettre des calculs arithmétiques et des manipulations de pointeurs, avec des opérateurs unaires (&, , -) et les opérations de multiplication et division (, /). Les expressions additives et relationnelles ont été ajoutées pour compléter les opérations arithmétiques et logiques avec les opérateurs (+, -, <, >, <=, >=). Les expressions d'égalité et générales ont été ajoutées pour permettre la comparaison de valeurs et l'affectation de résultats avec les opérateurs d'égalité (==, !=). Les déclarations et fonctions ont été définies pour limiter les types à int et void, garantissant la conformité avec les types permis dans STRUCIT-backend, et ajoutant les règles pour les déclarations de variables et de fonctions. Les instructions de contrôle et de retour ont été ajoutées pour gérer le flux de contrôle du programme, avec les instructions de contrôle (if ... goto, goto) et de retour (return). Enfin, la fonction main a été définie pour appeler le parseur (yyparse) afin d'analyser le programme source et vérifier sa conformité avec la grammaire définie.

Les modifications apportées ont été guidées par les spécifications du langage STRUCIT-backend, qui impose un format strict de code à trois adresses. Les expressions et opérateurs ont été adaptés pour gérer les expressions arithmétiques et logiques simples, nécessaires pour produire des instructions de la forme x = y op z. Les déclarations de variables et fonctions sont limitées aux types int et void *, conformément aux spécifications du backend. Les fonctions peuvent prendre des arguments de ces types et retourner des int ou void. Les instructions de contrôle conditionnelles (if ... goto) et inconditionnelles (goto) sont essentielles pour le contrôle de flux dans le code à trois adresses. Les affectations sont restreintes à des formes simples pour correspondre au modèle de code à trois adresses

III) LIMITES ET DIFFICULTÉS

Reprise en Main du Langage C et Gestion de Pointeurs et de Mémoire

L'une des premières difficultés rencontrées a été la reprise en main du langage C, notamment en ce qui concerne la gestion des pointeurs et de la mémoire. Étant donné que le C offre un contrôle bas-niveau sur la mémoire, cela requiert une attention minutieuse pour éviter des erreurs de manipulation des pointeurs. Les erreurs de gestion de mémoire peuvent conduire à des comportements imprévisibles du programme, y compris des plantages ou des corruptions de données. Cela a nécessité une révision approfondie des concepts de base du C, ainsi qu'une pratique constante pour maîtriser ces aspects critiques du langage.

Segmentation Faults Répétitifs

Une autre difficulté majeure a été le nombre incalculable de segmentation faults que nous avons rencontrés, en particulier lors de la génération du code à trois adresses et de la gestion de la table des symboles. Les segmentation faults sont souvent dus à des accès mémoire illégaux, et ils se sont avérés particulièrement frustrants et chronophages à déboguer. Nous avons passé de nombreuses heures à insérer des instructions de print dans le code pour tracer l'exécution et identifier les points de défaillance. Cette méthode de débogage, bien que rudimentaire, s'est révélée indispensable pour localiser les erreurs et comprendre les causes sous-jacentes des plantages.

Les segmentation faults étaient souvent causés par des erreurs dans la manipulation des pointeurs, comme la déréférence de pointeurs non initialisés ou la gestion incorrecte de la mémoire allouée dynamiquement. À chaque étape, nous devions vérifier attentivement l'allocation et la libération de la mémoire, ainsi que l'intégrité des pointeurs utilisés dans le programme.

Gestion du Temps et Pression des Échéances

Enfin, la gestion du temps a été un défi constant tout au long du projet. Le projet STRUCIT-backend était complexe et exigeant, nécessitant une compréhension approfondie et une mise en œuvre soignée des concepts de compilation. Parallèlement à ce projet, nous avions d'autres projets académiques à mener à bien ainsi que des examens à préparer. Cette cohabitation entre plusieurs obligations a souvent conduit à des périodes de stress intense, surtout à l'approche des échéances.

La pression du temps nous a obligés à travailler de manière plus efficace et organisée, mais cela a également signifié que nous avons souvent travaillé dans des conditions de sprint, en essayant de finaliser les fonctionnalités et de corriger les bugs à la dernière minute. Cette situation a parfois compromis la qualité de notre code et a exigé des ajustements et des corrections supplémentaires après les phases initiales de développement.

IV) CONCLUSION

La répartition des tâches dans ce projet a été essentielle pour gérer la complexité et l'ampleur du travail à accomplir. Matthias s'est concentré sur la génération de code à trois adresses ainsi que sur la définition des règles de grammaire. La génération de code à trois adresses a nécessité une compréhension approfondie de la transformation des instructions en un format intermédiaire utilisable pour l'optimisation et l'exécution. Les règles de grammaire, quant à elles, ont défini la structure syntaxique du langage, un travail crucial pour s'assurer que les programmes respectent les normes de syntaxe définies par STRUCIT-backend.

Parallèlement, Sacha a pris en charge la création et la gestion des tables de symboles, ainsi que le développement des composants lex et yacc pour le backend. La gestion des tables de symboles est une tâche critique pour suivre les déclarations et les utilisations des variables et des fonctions, assurant ainsi l'intégrité et la cohérence du programme. Le développement de lex et yacc pour le backend a permis de définir les règles lexicales et syntaxiques nécessaires pour analyser le code STRUCIT-backend.

Les routines sémantiques, qui vérifient la cohérence des types et des opérations dans le programme, ont été partagées entre Matthias et Sacha. Cela a nécessité une collaboration étroite pour s'assurer que les vérifications sémantiques étaient cohérentes et complètes.

En termes d'organisation, nous avons cherché à diviser les tâches autant que possible pour maximiser l'efficacité. Cependant, dans de nombreux cas, la séparation des tâches n'était pas possible en raison de la complexité et de l'interdépendance des différentes parties du projet. Dans ces situations, nous avons travaillé ensemble sur les mêmes fonctions, réfléchissant et résolvant les problèmes en tandem. Cette collaboration a permis de combiner nos compétences et perspectives pour surmonter les défis les plus difficiles.

Quant à notre avis sur le projet, nous reconnaissons que nous aurions pu mieux nous organiser pour gérer le temps et les tâches. Cependant, nous sommes satisfaits de ce que nous avons accompli. Le projet a été très instructif et nous a permis d'approfondir notre compréhension des concepts de compilation et de gestion de mémoire en C. Néanmoins, nous avons trouvé le projet assez difficile et très long. Un peu plus de guidage aurait été bénéfique pour nous orienter et nous aider à surmonter certains des obstacles plus efficacement.

Avancée du projet :

Fonctionnement des Fichiers et Génération du Code Intermédiaire

Tout d'abord, il est important de souligner que plusieurs fichiers de notre projet fonctionnent comme prévu et sont capables de générer du code intermédiaire de manière efficace. Par exemple, les modules de gestion des listes et des fonctions semblent fonctionner correctement dans la plupart des cas. Cela signifie que notre système est capable de reconnaître et de traiter correctement certaines structures de données et instructions, générant ainsi le code intermédiaire nécessaire pour les étapes ultérieures de compilation ou d'exécution.

Cependant, malgré ces succès, il existe des cas où le système reconnaît des éléments qu'il ne devrait pas. Par exemple, certaines listes et fonctions sont reconnues par notre système, alors qu'elles devraient logiquement produire des erreurs selon les règles de grammaire définies. Ce comportement inattendu suggère qu'il y a des lacunes dans notre implémentation actuelle des règles de grammaire ou des mécanismes de validation, ce qui permet à des structures incorrectes de passer sans être détectées.

Problèmes de Backend et Reconnaissance Incorrecte des Types

Le problème le plus crucial que nous avons identifié réside dans le backend du système. Plus précisément, nous rencontrons une difficulté persistante avec la reconnaissance des types de données. Un exemple concret de ce problème est la reconnaissance incorrecte d'un type INT. Dans notre système actuel, il semble qu'un INT soit reconnu même lorsqu'il n'a jamais été explicitement défini dans les règles de grammaire. Cette anomalie est particulièrement préoccupante car elle compromet l'intégrité de notre système de vérification et peut potentiellement conduire à des erreurs graves lors de l'exécution du code généré.

Nous avons effectué des recherches approfondies pour identifier la source de ce problème. Cela a impliqué une révision minutieuse de nos règles de grammaire, ainsi qu'une inspection détaillée des mécanismes de parsing et de vérification du type. Malgré nos efforts, nous n'avons pas encore réussi à localiser la cause exacte de cette reconnaissance incorrecte. Ce problème persistant suggère qu'il pourrait y avoir des failles plus profondes dans la façon dont nous avons conçu ou implémenté certaines parties du backend.

Impact sur la Vérification du Code

L'impact de ces problèmes sur la vérification du code est significatif. Étant donné que notre système est actuellement incapable de détecter et de signaler certaines erreurs correctement, la vérification reste incomplète. Cela signifie que des erreurs peuvent passer inaperçues jusqu'à ce qu'elles provoquent des échecs lors de l'exécution du code, ce qui est loin d'être idéal. Une vérification incomplète nuit non seulement à la fiabilité de notre système mais pose également des risques en termes de qualité et de sécurité du code produit.

En conclusion, malgré les défis rencontrés et la complexité du projet, nous avons réussi à atteindre nos objectifs et à produire un backend fonctionnel pour STRUCIT. Le projet nous a offert une précieuse expérience pratique et a renforcé notre capacité à travailler en équipe sur des tâches techniques complexes.