RAPPORT DU PROJET DE COMPILATION

Réalisé par :

- Fakhfakh Zied
- Mansouri Neila Camélia

Introduction :

Notre projet a pour but de créer un compilateur qui a la capacité de traduire un sous-langage du langage C appelé miniC en une représentation intermédiaire générée en langage DOT.

En s'appuyant sur les cours et les travaux dirigés qu'on a pu faire au cours du semestre, nous avons pu compléter les deux fichiers que les professeurs nous ont donnés à savoir « miniC.l » et « miniC.y ».

Pour l'analyse lexicale de notre langage, nous avons supprimé, comme indiqué, les tokens inutiles à notre langage. Nous avons également rajouté une expression régulière qui gère les commentaires.

Pour l'analyse syntaxique et l'analyse sémantique, nous nous sommes appuyés sur le fichier pdf qui détaille les différentes règles du langage miniC et ce qu'il contient, nous avons choisi d'implémenter deux tables de symboles dans notre compilateur, l'une pour les variables et l'autre pour les fonctions, en utilisant une table de hachage comme structure de données pour chacune.

Chaque règle de grammaire est accompagnée d'actions pour la génération de notre arbre. Lorsque nous parcourons le code miniC, nous analysons d'abord les tokens du programme source et construisons un AST approprié en utilisant notre grammaire et les fonctions qu'on a déclarées dans le fichier « node.c » dont la base est inspirée par les fonctions vues en TD.

Dans notre fichier « table_symboles.c »:

- La fonction 'hash calcule' le hash correspondant à notre variable
- La fonction 'init_table' initialise la table de symbole
- La fonction 'inserer' sert à inserer les symboles dans notre table de symbole
- La fonction 'existe' vérifie si le la variable existe déjà dans la table avant de l'insérer
- La fonction 'delete' permet de supprimer notre table à chaque fin de compilation

Dans notre fichier « table_symboles_fonction.c » :

- La structure param_t correspond aux paramètres des fonctions de notre langage
- La structure liste_t correspond à la liste des paramètres
- La structure fonction_t correspond à la structure des fonctions de notre langage
- La fonction init_table_symbole initialise la table des symboles des fonctions
- La fonction creer_liste crée la liste des parametres
- La fonction compter_elements compte le nombre de paramètres dans notre liste_t
- La fonction concatener_listes concatène deux listes de paramètres.
- La fonction ajouter_fonction ajoute la fonction à la table des symboles
- La fonction existe_fonction vérifie si la fonction existe au préalable avant de l'ajouter
- La fonction table_fonction_delete supprime la table

Dans notre fichier « node.c » :

- La structure node correspond aux nœuds de notre arbre
- La structure list_node correspond à une liste de nœuds
- La fonction create_empty_list crée une liste de nœuds vide
- La fonction create node crée un nœud
- La fonction append va concatener des chaines de caractères
- La fonction create_list crée une liste de nœuds non vide
- La fonction add_to_list ajoute des nœuds à la liste
- La fonction append_lists concatène deux listes de nœuds
- La fonction liste_length renvoie le nombre de nœuds dans une liste
- La fonction convert_to_array transforme en table une liste de nœuds

L’idée et la logique de notre projet :

Notre projet est composé de deux parties fondamentales :

Le stockage des variables et des fonctions dans la table des symboles en mémoire :

C’est un peu le backend de notre projet, nous avons pensé à utiliser des piles mais nous avons finalement opté pour les tables de hachages car celles-ci permettent non seulement un accès rapide aux éléments si la table devient trop grande, mais aussi une gestion moins couteuse en termes de performance de l’analyseur syntaxique qui peut être faite à mesure que l’analyse progresse.

La création de notre forme intermédiaire sous la forme d’un fichier DOT :

Prenons l’exemple de la construction du nœud SWITCH, d'abord, l'expression dans l'instruction SWITCH est créée comme un nœud. Ensuite, la liste de nœuds des CASE et DEFAULT de switch_cases est ajoutée à ce nœud à l'aide de la fonction ‘append_lists’. Cette liste ajoutée est ensuite convertie en tableau. Enfin, un nouveau nœud est créé pour toute l'instruction SWITCH, et le tableau de cas est passé à cette fonction. Cela construit un arbre de nœuds avec le nœud SWITCH à la racine, et le nœud d'expression et les nœuds des CASE et DEFAULT comme ses enfants. Cette base de fonctionnement est la même à travers tout le parsing.

Ce qui a été réalisé :

Nous avons pu stocker nos variables déclarées dans notre table des symboles et idem pour les fonctions.
Notre analyse lexicale ne renvoie aucune erreur selon le compilateur.
Nous avons également pu régler les problèmes syntaxiques et sémantiques, lors du test de notre compilateur nous n’avons aucune erreur générée.
Néanmoins, la génération de code ne se produit pas comme nous l’avions espéré, les erreurs sont détectées et affichées dans notre console, mais le fichier DOT créé ne contient malheureusement pas la totalité des nœuds comme souhaité.

```
mansourineila@MacBook-Pro-de-mansouri Projet-Compilation % lex miniC.l
mansourineila@MacBook-Pro-de-mansouri Projet-Compilation % yacc -d miniC.y
mansourineila@MacBook-Pro-de-mansouri Projet-Compilation % gcc  -ll lex.yy.c y.tab.c
mansourineila@MacBook-Pro-de-mansouri Projet-Compilation % ./a.out < Tests/variables.c variables
* Erreur "la variable existe deja", à la ligne 5:
variables declarees : int a;
```

Le test que nous avons effectué sur le fichier ‘variables.c’ a donné ceci :

mon_programme.dot

```
1  digraph mon_programme {
2    node_7 [label="programme" shape=ellipse];
3    node_7 -> node_6;
4    node_6 [label="main,int" shape=invtrapezium];
5    node_6 -> node_1080060848;
6    node_1080060848 [label="j`@" shape=int];
7  }
8
```

Problèmes rencontrés :

Nous avions du mal à commencer le projet, à visualiser ce qu’on devait faire et comment procéder à l’analyse sémantique de notre langage.
Nous avons essayé de rendre un projet qui soit complet et correct du mieux que vous pouvions malgré les difficultés qu’on a rencontrées.
Les erreurs affichées à la compilation ont pu être corrigées, en voici quelques exemples :

```
miniC.y:17:1: error: unknown type name 'List_Node'
List_Node* noeuds;
^
```

```
In file included from miniC.y:66:
./table_symboles_fonctions.c:4:10: error: conflicting types for 'concatener_listes'
liste_t *concatener_listes( liste_t *l1,  liste_t *l2 );
         ^
./table_symboles_fonctions.h:38:10: note: previous declaration is here
liste_t *concatener_listes(struct liste_t *l1,  liste_t *l2 );
         ^
```

Partage des tâches :

En ce qui concerne le partage des tâches, nous avons fait le projet ensemble en présentiel du début à la fin. On se voyait tous les deux et on avançait comme on le pouvait, on se partageait l'écriture des fonctions, on s'entraidait en cas de problèmes, et on réfléchissait à deux afin d'être le plus efficace possible.

Nous avons aussi demandé de l'aide à nos camarades qui ont eu la gentillesse de nous aider à comprendre ce qui doit être représenté ou pas dans l'AST.

Conclusion :

À l'issue de ce projet, nous avons acquis une connaissance approfondie des fondements de la création d'un compilateur. Cette expérience nous a non seulement fourni une nouvelle compétence technique, mais nous a aussi appris la valeur de l'écoute active et de la communication précise.
Les erreurs rencontrées étaient souvent difficiles à résoudre, et bien que nous ne sommes pas parvenus à avoir le résultat escompté, nous avons fait de notre mieux pour présenter un travail aussi complet et précis que possible jusqu'au bout.
Nous sommes reconnaissants pour cette expérience, malgré sa difficulté, et remercions nos professeurs pour leur patience tout au long du semestre et les différentes ressources qu'ils nous ont fournies.

MERCI BEAUCOUP !