Compilation

Projet Compilateur

KAOUANE Marwane - NEJI Wadi

Introduction :

L'objectif de ce projet est de poursuivre le développement de notre analyseur lexical pour aboutir à un compilateur complet en utilisant les outils Flex et Bison. Ce compilateur doit traduire un fichier source en langage tpc en un arbre syntaxique abstrait , capable de reconnaître les structures du langage et de détecter les erreurs syntaxiques.
Une fois les erreurs syntaxiques identifiées et corrigées, le compilateur procédera à la détection des erreurs sémantiques dans le code, conformément à la sémantique des expressions et instructions du langage C. En cas d'erreur sémantique, un message d'erreur approprié sera émis. Après avoir traité toutes les erreurs, notre compilateur convertira le code source tpc en code assembleur NASM en respectant les conventions d'appel AMD64. Le code assembleur généré pourra ensuite être exécuté pour vérifier le résultat de la compilation. Le projet consiste à réutiliser et potentiellement modifier le projet d'analyse syntaxique du premier semestre pour inclure les nouvelles fonctionnalités demandées. Pour valider notre travail, nous avons préparé plusieurs jeux d'essais incluant des programmes corrects et des programmes avec différentes sortes d'erreurs (lexicales, syntaxiques, et sémantiques), ainsi qu'un script de test pour automatiser la vérification des résultats.

Correction de l’arbre abstrait :

La première étape à été de corriger notre arbre abstrait celui n’étant pas au point pour nous simplifier grandement la tâche lors du parcours de celui-ci. Une des que lorsque nous traduisons dans l’arbre un appel de fonction les paramètres était sous la forme :

- premier param = père
- n -1 param = frère entre eux et fils du premier paramètre.

Très loin d’être pratique cela à été réglé. D’autres problématiques pratiques ont été réglées par la suite.

Tables de Symboles :

Pour créer une table de symbole nous avons décidé de procéder comme suit:

```
 7  typedef struct symbole {
 8      char *name;                    // nom de la variable
 9      int adresse;                   // adresse de la variable
10      char *type;                    // type de la variable
11      int indexTab;                  // index du tableau
12      struct symbole *suiv;
13  } Symbole;
```

Figure 1 : Contenu des symboles d’une table.

Pour représenter une liste de symboles dans la table, nous avons décidé d’opter pour une structure comprenant : un nom, une adresse, un type sous forme de chaîne de caractères,

et un index indexTab qui sera égal à la taille du tableau si c'en est un. Si c'est un tableau, indexTab > 0; sinon, c'est un symbole quelconque. Le champ *suiv est un pointeur vers l'élément suivant dans la liste, ou vaut NULL s'il n'y a pas d'élément suivant.
Nous avons donc décidé d'implémenter cette structure sous forme de liste chaînée, en raison de son aspect pratique pour la gestion dynamique des symboles.

```
16  typedef struct tableSymbole {
17      char *name;                     // nom de la fonction
18      char *type;                     // type de la fonction
19      int adresse_retour;             // adresse de retour
20      Symbole *symbole;               // tables des variables locales + params
21      int nb_param;                   // nombre param
22      struct tableSymbole *suiv;      // pointeur vers la tete de liste de symbole
23  } TableSymbole;
```

Figure 2 : Une Table de Symbole.

Une table de symbole dans notre projet concerne uniquement les tables pour les fonctions, la table globale est une suite de symboles seulement (voir figure 3). Notre table possède un nom qui sera le nom de la fonction, un type indiquant le type de retour de la fonction, l'adresse de retour, donc l'adresse à laquelle on ira après avoir quitté la fonction. Son nombre de paramètres qui nous sera utile pour éviter les erreurs de sémantique si la fonction est appelée avec trop d'arguments et un pointeur vers la tête de la liste de symbole.

```
typedef struct tables {
    TableSymbole *fonction;         // tables des variables locales
    TableSymbole *fonctCourante;    // Garde en mémoire l'adresse de la dernière fonction insérer
    Symbole *globale;               // tables des variables locales
} Tables;
```

Figure 3 : Les Tables de Symboles.

Nous avons décidé de chaîner les tables contenant les fonctions. *fonction est un pointeur vers la tête de la liste donc de la première fonction du code. *fonctCourante est l'adresse de la dernière fonction insérée dans la table des fonctions, utile pour éviter de parcourir toute la table des fonctions pour trouver les erreurs de sémantiques. Enfin *gloable est uniquement une liste de symboles, ne portant pas de nom ou autre, inutile cela prendrait plus d'espace que nécessaire et possèderait un aspect moins pratique en terme d'accès à celle-ci.

Pour remplir les tables de symboles nous utilisons deux fonctions principales et d'autres auxiliaires :

- -construitListeSymbole
  Cette fonction parcourt l'arbre abstrait représentant une structure de programme et vérifie les déclarations de fonctions. Elle extrait le nom et le type de retour de chaque fonction, vérifie s'il existe des conflits de noms avec des variables globales ou des fonctions prédéfinies, et gère les paramètres de fonction. Enfin, elle ajoute les informations sur la fonction analysée à une table de symboles.

- -rempliTableSymbole
  Cette fonction analyse les déclarations de variables dans un arbre syntaxique représentant une structure de programme. Elle parcourt les nœuds de l'arbre pour extraire le type de chaque variable et vérifier les éventuelles redéclarations. Elle gère également les déclarations de tableaux, vérifiant leur taille et évitant les redéclarations. Enfin, elle ajoute les informations sur les variables analysées à une liste de symboles.

Sémantique :

Nous organisons le contrôle de la sémantique en deux phases:

- Premièrement, nous vérifions les éléments lors du remplissage des tables de symboles, par exemple en vérifiant si une nouvelle fonction **getint()** est déclarée correctement. Dans notre grammaire, les tableaux peuvent être déclarés avec des expressions et pas forcément avec des valeurs fixes, mais ils doivent toujours être déclarés avec des valeurs positives.
- Puis une autre phase où nous allons calculer les expressions dès que nous arrivons au nœud suiteinstruction. À partir d'ici, on vérifie plutôt si les variables sont de bons types ou encore si les fonctions utilisées dans les expressions ne sont pas de type void. Les tableaux, dans cette phase, sont utilisés à l'aide d'index toujours positifs et chaque tableau peut être utilisé uniquement avec des opérateurs valides, comme l'accès aux éléments via des crochets (**[]**) ou le passage en argument dans un appel de fonction.

Traduction:

La fonction principale, **parcoursMain**, traverse l'arbre abstrait et génère les instructions assembleur appropriées pour chaque nœud rencontré. Les types de nœuds gérés incluent :

- Les opérations arithmétiques et logiques.
- Les affectations et les constantes.
- Les structures de contrôle comme les conditions et les boucles.
- Les appels de fonctions et les retours.

La fonction **fonctionsASM** ajoute les définitions nécessaires pour les fonctions standard d'entrées/sorties en assembleur, telles que **putchar**, **getchar**, **getint**, et **putint**.

La fonction **traductionASM** initialise les sections .bss, .data, et .text, s'occupant de la déclaration des variables globales et de la structure du programme assembleur.

En résumé, la fonction **parcoursMain** convertit l'arbre en code assembleur en gérant divers types de nœuds et structures. Les fonctions supplémentaires s'assurent que les sections et les définitions standard nécessaires sont correctement configurées pour l'exécution du programme

Organisation :

Durant ce projet nous avons collaboré étroitement pour mener à bien ce projet en utilisant Git et Notion pour une gestion de projet optimale. Chacun d'entre nous a travaillé sur des branches distinctes de Git, se concentrant sur des aspects spécifiques tels que la traduction sémantique, ce qui nous a permis de développer et tester nos versions de fichier de manière indépendante avant de les merge. Avec cette organisation nous avons assuré la stabilité du code.
De plus, pour une organisation efficace de nos tâches, nous avons utilisé Notion pour créer et gérer des tickets. Ces tickets ont servi à planifier nos tâches, à définir des objectifs clairs, et à suivre notre progression. Grâce à cette méthode, nous avons pu prioriser nos efforts, résoudre rapidement les problèmes, et maintenir une communication tout au long du projet. Cette organisation a été essentielle pour respecter nos délais et atteindre les objectifs du projet avec succès.

Compilation :

Lancer la commande **--help** ou **-h** qui va afficher les commandes générales pour compiler et exécuter.
Sinon :

- **-t** ou **--tree** affiche l'arbre abstrait sur la sortie standard
- **-h** ou **--help** affiche une description de l'interface utilisateur et termine l'exécution **-s**,
- **--symtabs** qui affiche toutes les tables des symboles sur la sortie standard ;
- **python3 script.py** qui lance le banc de test pour les 4 dossiers demandés.

Conclusion :

Ce projet nous a offert une expérience enrichissante dans le domaine de la compilation et du développement logiciel. En travaillant sur notre analyseur syntaxique et en utilisant des outils tels que Flex et Bison, nous avons approfondi notre compréhension du processus de compilation.
Nous avons résolu divers défis, notamment l'optimisation de l'arbre abstrait et la mise en place des tables de symboles.
Cette expérience nous a permis de comprendre ce qui se passe derrière les compilateurs que nous utilisons depuis nos débuts en programmation, ce qui est utile pour avoir une idée plus précise du fonctionnement de chaque instruction.
Grâce à une collaboration efficace, réalisée à l'aide d'outils prisés dans le monde professionnel, Git et Notion, nous avons assuré la stabilité du code et respecté les délais. En fin de compte, ce projet nous a permis de renforcer nos compétences en programmation, en gestion de projet et en collaboration.