MATHIEU Yanis
MARTINEZ Baptiste

RAPPORT DE PROJET

SOMMAIRE :

I) NOS CHOIX :

Tout d'abord, pour moduler notre projet nous avons fais le choix de le séparer en plusieurs fichiers sources (symbol_table.c, semantique.c, traduction.c) pour une meilleure lisibilité et maintenance.

symbol_table.c :
Nous avons choisi d'implémenter une **table de hachage** (SymbolTable) :

```
typedef struct SymbolTable {
    Symbol *tab[MAX_SYMBOLS];
} SymbolTable;
```

pour retrouver rapidement les symboles (variables, fonctions) à partir de leur nom.

- Utilisation d'enums :
Pour structurer la table des symboles et l'arbre abstrait, nous avons utilisé plusieurs énumérations (SymbolType, VarType) permettant de différencier :
- le type des symboles (variable locale, globale, fonction) :

```
typedef enum {
    VAR_GLOBAL,
    VAR_LOCAL,
    FUNC
} SymbolType;
```

- le type des variables (int, char, void, double) :

```
typedef enum {
    INT,
    CHAR,
    VOOID,
    DOUUBLE
} VarType;
```

semantique.c :

Nous avons mis dans ce fichier toutes les fonctions qui gèrent les erreurs sémantiques auxquelles nous avons pensé. Notamment :
- Les erreurs sur les égalités entre un char et un int,
- Les erreurs sur les if et while qui ne contiennent pas d'expression booléenne,
- Les erreurs sur les opérations avec des identifiants non déclarés ou entre des types différents, (par exemple 'h' + 5)
- Les erreurs sur les identifiants qui ont le même nom,
- Les erreurs sur les fonctions qui ne retourne pas une valeur du type attendu, (par exemple une fonction char f(void) ; qui retourne un int)
- Les erreurs sur les division par 0.

traduction.c :

Nous avons mis dans ce fichier toutes les fonctions qui gèrent la traduction en langage assembleur, pour traduire les while, les return, les égalités, les expressions, les if, et le corps des fonctions.

II) DIFFICULTÉS RENCONTRÉES :

Gestion des conditions (if et while) :
Nous avons réussi à traduire les instructions if et while simples, avec des conditions contenant des expressions booléennes élémentaires telles que a > 5 ou a <= 4.
En revanche, nous avons rencontré de nombreuses difficultés lors de la traduction d'expressions booléennes plus complexes, incluant des opérations arithmétiques imbriquées, comme par exemple (a + 5) > 5 ou encore (a * 8) < 5 || a == 4.
Nous n'avons finalement pas réussi à traduire correctement ces cas plus complexes.

Expressions arithmétiques longues :
Au début du projet, les expressions arithmétiques complexes, par exemple a*5+4-5+6, posaient des problèmes lors de l'analyse. Nous avons finalement réussi à les traiter correctement.

Erreurs sémantiques :
Concernant la vérification sémantique, nous n'avons pas réussi à couvrir toutes les erreurs possibles.
En effet, notre banc d'essai obtient un score de **82,93 %** sur la plateforme e-learning, ce qui suggère que certaines erreurs ne sont pas détectées par notre analyseur sémantique.
Nous pensons que ces lacunes viennent principalement de cas spécifiques que notre implémentation ne prend pas encore en compte.

III) AMÉLIORATIONS :

Depuis le projet d’analyse, nous avons apporté plusieurs modifications et améliorations à notre analyseur lexical (lexer), notre analyseur syntaxique (parser) ainsi qu’à la structure de notre arbre syntaxique.

Concernant l’arbre, nous l’avons retravaillé afin qu’il soit plus facile à parcourir, notamment pour les phases de traduction et de construction de la table des symboles.

Du côté du lexer, nous avons ajouté la récupération des valeurs pour les opérateurs addsub, divstar, eq, order et pour les types. Pour gérer cela, nous avons intégré dans le parser trois tableaux (operation, type, boolean) qui nous permettent :

-de connaître précisément l’opération utilisée dans un nœud addsub ou divstar.

-d’identifier le type exact d’une variable (par exemple int ou char).

-de traiter correctement les comparaisons comme eq ou order.

Ces ajouts facilitent l’analyse sémantique et la traduction en assembleurs.

IV) LANCEMENT DU PROGRAMME :

Pour compiler le programme, il faut utiliser la commande make, qui créera deux dossiers :
-un dossier obj pour les fichiers objets,
-un dossier bin contenant l’exécutable tpcc.

Ensuite, il faut exécuter le programme avec la commande :

./tpcc -option < fichier.test

Les options disponibles sont :

-t ou -tree : pour afficher l’arbre abstrait,
-s ou -symtabs : pour afficher la table des symboles,
-h ou -help : pour afficher l’aide.

Nous avons également mis en place un banc de tests comprenant différents cas :

-des tests avec erreurs syntaxiques,
-des tests avec erreurs sémantiques,
-des tests générant des warnings,
-des tests valides.

Pour utiliser le banc de tests, il faut d’abord rendre le script exécutable avec :

chmod +x testScript.sh

puis lancer les tests avec :

./testScript.sh

Le banc de tests s’exécutera automatiquement et un rapport affichera les résultats, en indiquant si chaque test a été réussi ou échoué.