APP4 Compilation – Rapport

Pour réaliser ce compilateur, nous avons choisi le langage C# de Microsoft ainsi que le framework .NET. Il peut donc être compilé sous Windows avec les outils proposés par Microsoft ou sous Linux en utilisant Mono.

Architecture du code

Le code est structuré de manière à séparer chaque étape de la compilation. Dans la solution *CLightCompiler*, on retrouve donc cinq projets :

- *LexicalAnalysis,* qui convertit du texte en série de tokens;
- *SyntaxAnalysis*, qui convertit une série de tokens en arbre en vérifiant la syntaxe;
- *SemanticAnalysis*, qui vérifie que la sémantique de l'arbre est correcte;
- *CodeGeneration, qui génère le code assembleur à partir de l'arbre*;
- *CLightCompiler*, qui regroupe toute la chaîne de traitement.

Analyse lexicale

Dans le projet *LexicalAnalysis*, on retrouve la classe qui définit ce qu'est un token ainsi que la liste de tous les tokens valides pour notre langage. L'énumération tokens regroupe tous les types de tokens disponibles. La classe *LexicalAnalyser* est constituée d'une méthode Convert qui permet de transformer le code en entrée en liste de token. Si le code possède un caractère invalide ou le token correspondant au mot n'est pas n'existe pas, alors une exception de type *LexicalException* est lancée.

Analyse syntaxique

Dans le projet *SyntaxAnalysis*, on retrouve la classe qui définit un nœud (node) ainsi que la liste de tous les types de nœud (nodes) disponibles. La classe *SyntaxAnalyser* implémente une méthode Convert qui permet générer un arbre de node à partir d'une liste de tokens valides. Pour sortir obtenir un arbre, on suit les règles de grammaire vues en cours. Si au cours d'une analyse, la règle n'est pas vérifiée, alors la compilation s'arrête et une exception de type *SyntaxException* est lancée.

Analyse sémantique

Dans le projet *SemanticAnalysis*, on retrouve la classe définissant un symbole, et la classe qui permet de gérer la table des symboles. Cette dernière est représenté par une pile de dictionnaires. Chaque dictionnaire contient l'ensemble des variables qui ont été déclaré dans un bloque. Ici la clé est l'identifiant du token et la valeur est le symbole. La classe *SemanticAnalyser* contient une méthode Analyse qui parcourt deux fois l'arbre. Une fois

pour vérifier si les breaks et continue sont écrits dans le corps d'une boucle. Une deuxième fois pour générer une table des symboles à partir d'un arbre de node. De plus, elle complète l'arbre en ajoutant les bons slots aux nœuds. Trois exceptions peuvent être lancées au cours de cette analyse :

- SymbolAlreadyExistsException : un symbole existe déjà (par exemple : une déclaration multiple d'une même variable) ;
- *SymbolNotFound* : le symbole cherché n'existe pas (par exemple : une variable non déclarée) ;
- SymbolIsNotIdent : on tente de chercher ou d'ajouter un symbole à partir d'un token qui ne correspond pas à un identifiant.

Génération de code

Dans le projet *CodeGeneration*, on retrouve la classe *MSMCodeGenerator* qui implémente l'interface *ICodeGenerator*. Elle permet de générer un code assembleur à partir d'un arbre. C'est à ce niveau-là que sont géré les labels des boucles et des conditions.

Compilateur

Dans le projet *CLightCompiler*, on retrouve deux classes. La classe *CLightCompiler* contient une méthode *Compile* qui permet de générer le code assembleur. Cette méthode exécute les différentes analyses (lexicale, syntaxique et sémantique). Elle compile aussi le run time (std.c). Cette méthode *Compile* est appelée dans le main de la classe *Program*. C'est aussi dans le *main* que sont attrapées les exceptions.

Tests

Les tests pass.XXX.txt issus du site https://perso.limsi.fr/lavergne/ ont été passés avec succès. De plus nous avons nos propres tests pour chaque étape de la compilation. Ces tests sont inclus dans une série de tests unitaires automatisés de manière à valider le bon fonctionnement du compilateur à chaque étape du développement.

Spécificités

Dans cette partie, nous allons voir quelles sont les différences entre le langage que l'on peut compiler et celui que nous avons vu en cours.

- Le seul type de variables est int;
- Il n'y a pas de point-virgule après le while dans une boucle "do...while";
- Le compilateur ne vérifie pas si une fonction a le bon nombre d'argument lors d'un appel;
- L'utilisation de variables non initialisées est autorisée.

Fonctionnalités

Fonctionnel :

- Variables (déclaration et affectation)
- Structures de contrôle (if et else)
- Boucles (while, do while et for)
- Break et continue
- Fonctions (doivent être déclarées avant l'appel)
- Pointeurs
- Tableaux
- Allocation dynamique

Non-fonctionnel :

- Déclaration et affectation d'une variable en une instruction
- Les out.i sont utilisé à la place des out.c
- La fonction *free(int ptr)* n'est pas implémentée
- La fonction *print(int x)* n'est pas implémentée