Projet compilation APP4

Pour ce projet, nous avons utilisé le langage C#, développé principalement avec Visual Studio et JetBrain Rider.

Afin de suivre l’avancement de la compilation, une image de l’arbre sera affichée dans le terminal avant d’afficher le code généré pour notre machine simulée. Ce code généré sera aussi copié dans un fichier de sortie que nous avons appelé « code.txt »

# Parties fonctionnelles

* Lecture du code source (en langage C) à partir d’un fichier : Lors du lancement du programme, spécifier le chemin du fichier à lire avec l’option « -d ».  
  Exemple : Compil.exe -d c:\Compilation\test.c

## Analyser syntaxique

Partie fonctionnelle, les parties (« mots ») du code sont bien découpé en Token.  
Ces tokens sont enregistrés dans une énumération qui nous permet de standardiser les données. Ce sont ces énumération que nous utilisons pour identifier les tokens entre les différents analyser donc des différentes classes de notre code.

## Analyse Syntaxique

Cette partie sert à construire l’arbre de notre compilateur. Il récupère les différents tokens et les joint selon les règles que l’on a définies.

* Création de l’arbre et affichage dans la console.
* Gestion des opérateurs unaires.
* Gestion des déclarations et affectation des variables.
* Gestion des condition IF/ELSE/ELSEIF dans l’arbre.
* Gestion des boucles avec les nœuds « loop » infinies.
* Gestion des boucles « While » et « For » en utilisant la notion des boucles « loop » avec condition d’arrêt de type IF.
* Gestion de la notion de Block ***avec portée des variables***
* Exceptions verbeuses levées en cas de crash (« Primaire attendue », « missing variable x », etc).

## Analyser sémantique

Cette partie sert à comprendre les symboles qui sont dans les tokens. Grâce à une énumération des symboles on est capable de détecter quel nœud correspond à quel symbole.

* Table des symboles implémentée.
* Scope des variables gérées (les variables déjà déclarées peuvent être reconnues)
* Exceptions levées (« Variables already declared in this scope”, etc)

## Générateur de code :

Le générateur de code est capable de construire les instructions pour la machine virtuelles pour les opérations suivantes :

* Opérations arithmétiques (+, -, \*, /)
* Valeurs constantes
* Opérateur unaire pour les signes des valeurs numérique (+ et -)
* Blocs de code
* Expressions
* Conditions (if, else, elseif), il est possible d’imbriquer différentes conditions les unes dans les autres
* Opérateurs de comparaison (==, !=, <, >, <=, >=)
* Les boucles (de façon imbriqué)
  + While
  + For : la déclaration de la variable de condition doit être déclaré en dehors de la parenthèse de condition)  
    Exemple :  
    var i ;  
    for (i=0 ; i<5 ; i++) {}  
    Il s’agit d’un choix de conception et non d’une erreur.

# Parties non fonctionnelles

## Parties non implémentées

Malgré nos efforts, quelques fonctionnalités n’ont pas pu être implémentées c’est le cas de :

* Les blocks de type « Switch / Case”
* Les tableaux : par manqué de temps, nous n’avons pas pu gérer le système de pointeur qui permet de les gérer.

# Codes de test

Pour tester notre compilateur, nous avons créer un certain nombre de code de test en C.  
Nous avons ensuite exécuter les codes avec la machine virtuelle et suivi l’exécution avec le système de débogage. Le cahier de tests que nous avons utilisé est dans un fichier Excel (« cahier\_test\_Capodano\_Hamel.xlsx »).

Vous pourrez retrouver tous ces codes de tests dans le dossier « Codes\_tests » joint dans l’archive.

# Amélioration

Afin d’ajouter des fonctionnalités à notre compilateur, nous avons ajouté les librairies standards du langage C avec les directives « #include ». Ce sont elles qui nous permettent d’utiliser les commandes comme « print » dans notre code original et que cette dernière soit interprété simplement.

# Difficultés rencontrées

Nous avons eu du mal à établir toutes les règles de l’interpréteur de code pour les différents systèmes d’exploitation. Nous avons développé notre compilateur sur un environnement Mac et l’autre Windows. Par exemple, pour les sauts de ligne, sur Mac le caractère « \n » était reconnu alors que sur Windows ce caractère est « \r ». Ce genre de différence nous a retarder car nous avons eu du mal à trouver d’où venait le problème.

# Organisation du compilateur

Le découpage du compilateur tel qu’il a été présenté nous a permis de bien comprendre son fonctionnement, nous pensons qu’il était important de prendre du temps pour comprendre ce découpage pour développer un système structuré pour la suite.

Afin de suivre le découpage vu en cours, nous avons voulu regrouper le code de notre compilateur de la même façon. C# permet de créer des dossiers dans lequel nous pouvons y placer les différentes classes que nous avons utilisé.