Projet compilation APP4

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

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 du code sont bien découpé en Token.  
Ces tokens sont enregistrés dans une énumération qui nous permet de standardiser les données.

## 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), il est possible d’impbriquer 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

Les blocks de type switch / case ne sont pas implémentés ainsi que les « DO WHILE ».

# 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. Voici un tableau qui regroupe l’ensemble de ces codes de test :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| But du test | Nom fichier de test | Code source | Code généré | Retour d’exécution |
| Déclaration de variable | testDeclareVar.txt | function main()  {      var i;      return 0;  } | resn 0  .main  resn 1  get 0  dup  dbg  push 0  dup  dbg  ret  push 0  ret  .start  prep main  call 0  halt | 0  0 |
| Déclaration et initialisation variable | testDeclareInitVar.txt | function main()  {      var i;      i = 5;      return 0;  } | resn 0  .main  resn 1  get 0  dup  dbg  push 5  dup  dbg  dup  set 0  drop  push 0  dup  dbg  ret  push 0  ret  .start  prep main  call 0  halt | 0  5  0 |
| Addition de deux variables |  |  |  |  |

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

# Difficultés rencontrées

# Organisation du compilateur

Si on aurai changer des choses