|  |  |
| --- | --- |
|  | Nicolas Bourneuf  Rayan LA ROZE  APP4 INFO |
| Rapport de Projet | |
|  |  |
| 21/12/2019 | Compilateur pour Ministack Machine |
|  | Le but du Projet : Réalisation d’un compilateur minimaliste. |

Table des matières

[Organisation du compilateur 1](#_Toc27831971)

[1. Analyseur Lexical 2](#_Toc27831972)

[2. Analyseur Syntaxique 2](#_Toc27831973)

[3. Analyseur Sémantique 2](#_Toc27831974)

[4. Générateur de code 2](#_Toc27831975)

[Eléments demandés 0](#_Toc27831976)

[Fonctions de la librairie standard implémentée 0](#_Toc27831977)

[Difficultés rencontrées 0](#_Toc27831978)

# 

# Organisation du compilateur

La fonction Main du compilateur est son cœur. Dans un premier temps, elle va récupérer et analyser les arguments qui lui sont passés en paramètre.

Si elle voit un flag :

* « -f » elle va charger le fichier contenant le code à compiler et réaliser une analyse approfondie des erreurs du fichier, qui seront visible par l’utilisateur avec le mode de débogage.
* « -l » elle va charger le fichier de librairie standard qui suit cet argument. Elle va effectuer les mêmes opérations que sur le fichier à compiler, mais l’utilisateur ne pourra pas observer le traitement de ce fichier qui est censé être sans erreurs.
* « -d » elle se mettra en mode débogage. De ce fait, elle retournera la liste des tokens, la liste des nœuds, les erreurs rencontrées ou le code généré si aucune erreur n’est rencontrée.

**N.B.** : Le fichier source ainsi que celui de la librairie doivent se trouver dans le même dossier que le Main.class qui permet la compilation du code.

Au début du Main le programme va commencer par ouvrir un bloc dans l’analyseur sémantique afin que les fonctions puissent être déclarées dans le fichier sans avoir à écrire un bloc manuellement. De la même façon à la fin du Main le bloc est fermé. Par la suite le programme va lancer une analyse du fichier par l’analyseur lexical qui est une classe qui prend en entrée le fichier de code à analyser.

Une fois l’analyse lexicale effectuée, on donne en entrée à la classe « AnalyseurSyntaxique » l’analyseur lexical créé précédemment.

Tant que le programme ne détecte pas que l’analyseur lexical est arrivé au « tok\_end\_of\_file », il va successivement créer les nœud avec l’analyseur syntaxique puis les analyser sémantiquement. L’analyse sémantique se fait grâce à la classe « AnalyseurSemantique » et à sa méthode de classe « void nodeAnalyse() » qui prend en entrée le nœud. Si aucune erreur n’est rencontrée on génèrera le code avec la méthode de classe « String genCode() » (de la classe « CodeGenerator ») qui prend en entrée le nœud généré.

Une fois le code généré le programme va ajouter au début du code pour MSM les instructions :

* « .start »
* « prep init »
* « call 0 »
* « halt »

Ces instructions permettent d’appeler la fonction « init(n) » codée dans le programme du développeur qui alloue de la mémoire pour son programme puis appelle la fonction  « main() » contenant son code à compiler.

## Analyseur Lexical

L’analyseur lexical possède une liste de caractères spéciaux, qu’elle pourrait rencontrer, ainsi qu’une liste de mots-clés permettant de différencier les noms de variables des mots introduisant des instructions. Grâce à l’appel de sa méthode « void analyse() » elle va créer un tableau de tokens qui en renseignant la ligne où elle l’a trouvé et la colonne. Elle va analyser le fichier caractère par caractère en ignorant les espaces et les sauts de ligne. Si elle rencontre un caractère spécial qui pourrait être suivi d’un autre caractère spécial, elle va l’analyser en premier. Sinon elle ira rechercher dans la liste des caractères spéciaux l’opérateur correspondant. Si le caractère n’est pas un caractère spécial l’analyseur va prendre tous les caractères non spéciaux, rechercher s’il existe dans les mots clés et l’ajouter comme un identifiant le cas échéant. Un « tok\_end\_of\_file » est placé à la fin de la liste afin de marquer la fin du fichier.

## Analyseur Syntaxique

L’analyseur syntaxique étudie chaque token un à un en commençant par chercher des déclarations de fonctions. Dans ces déclarations de fonction, il recherchera des instructions qui elles-mêmes rechercheront des expressions formées de tokens primaires et d’opérateurs. Cette recherche successive est répartie sur différentes fonctions : « Node Function() » ; « Node Instruction() » ; « Node Expression() » ; « Node Primaire() » et « Operator ChercherOp() ». Chacune de ces fonctions se voient attribuées des tokens à analyser en fonction de leur rôle et s’occupent de vérifier la syntaxe du code.

## Analyseur Sémantique

L’analyseur sémantique récupère les nœuds déjà créés par l’analyseur sémantique et attribue des slots aux variables et aux fonctions. Cette analyse permet dans un premier temps de pouvoir connaître l’emplacement mémoire des variables et des fonctions. Elle permet également de vérifier qu’elles ont bien été déclarées et non dupliquées.

## Générateur de code

Par la suite, le générateur de code va simplement générer le code associé à l’arbre qu’il reçoit en le parcourant récursivement.

# Eléments demandés

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Elément** | | **Implémenté** | **Commentaires** | **Code de test** |
| Expression | |  |  |  |
| Conditionnelles | |  |  |  |
| Variables | Déclaration |  |  |  |
| Utilisation |  |  |  |
| Affectation |  |  |  |
| Portée |  |  |  |
| Boucles | for |  |  |  |
| while |  |  |  |
| do while |  |  |  |
| Fonction | Déclarations |  |  |  |
| Appels |  |  |  |
| Passage d’arguments |  |  |  |
| Puissance |  |  |  |
| Pointeur et tableau | Syntaxe et tableau |  |  |  |
| Syntaxe tableau |  |  |  |
| Fonction d’allocation |  |  |  |

# Fonctions de la librairie standard implémentée

# Difficultés rencontrées