Polytech Paris Sud

Compilation

Rapport du projet

GROUPE : WARREN WEBER, VENDE REMI

Table des matières

I – Fonctionnement de notre compilateur

Avant de lancer le compilateur il faut compiler la machine à pile avec les commandes suivante :

- cd msm

- make clean

- make all

Une fois cette étape passée, on va pouvoir lancer notre compilateur.

Pour cela nous utilisons le langage python et plus exactement python3.

Puis pour enfin lancer le compilateur il faut lancer la commande suivante :

python3 main.py NomDeFichier

II - Organisation du compilateur

L'analyse lexicale est faite dans le fichier '*lexical.py*'.

L'analyse syntaxique est faite dans le fichier '*syntax.py*'.

L'analyse sémantique est faite dans le fichier '*semantique.py*'.

La génération de code est faite dans le fichier '*genCode.py*'.

L'ensemble des éléments ci-dessus est appelé dans le fichier '*main.py*'.

La bibliothèque standard se trouve dans le fichier '*bibliotheque_standard.c*'.

III – Liste des fonctionnalités du compilateur

Fonctionnalités	Opérationnalité
Expression	OUI
Conditionnelles	OUI
Les variables	OUI
Déclarations	OUI
Utilisation	OUI
Affectation	OUI
Portée	OUI

Les boucles	
Boucle for	OUI
Boucle while	OUI
Boucle Do While	NON
Les fonctions	OUI
Déclaration	OUI
Appel	OUI
Passage d'arguments	OUI
Puissance	OUI
Pointeurs et tableaux	NON
Syntaxe pointeur	
Syntaxe tableaux	
Fonction d'allocation	
Fonction de libération	

NB : On a implémenté la fonction puissance, mais on a pas fait la génération de code pour la puissance. Donc on ne peut pas utiliser la puissance. Cependant nous avons créer une fonction puissance dans la bibliothèque standard.

IV – Code programme principal

Dans le fichier *main.py,* on commence par créer un fichier « main.c ». Ce fichier est une concaténation du fichier donné en argument et de la bibliothèque standard que nous avons fournit.

Ensuite, on fait l'analyse lexicale. Pour cela on donne le fichier main.c en argument. On lance l'analyse lexicale puis, tant qu'on n'atteint pas la fin du fichier, on réalise l'analyse syntaxique à laquelle on passe l'analyse lexicale en argument. L'analyse syntaxique renvoi un arbre à chaque fonction analysée. Ces arbres sont stockés dans une liste. Puis nous lançons l'analyse sémantique auquel nous passons les arbres, un par un, en argument.

Puis nous lançons la génération de code. Cette étape va venir écrire dans le fichier « *genCode* ».

Enfin nous lançons un sous processus pour exécuter le fichier *genCode.* Le sous processus affiche le résultat du code s'il y en a un a afficher, mais il affiche aussi l'arbre correspondant à l'analyse complète.

V – Analyse lexicale

L'analyse lexicale se fait via la classe Lexical du fichier « *lexical.py* ». Cette classe contient une liste *les_tokens* qui liste l'ensemble des tokens analysés dans le fichier. Nous avons découpé les

tokens possibles en 4 types : les mots clés, les opérateurs binaires, les ponctuations et les comparaisons.

Mot clés	
if	tok_if
else	tok_else
for	tok_for
var	tok_var
while	tok_while
function	tok_function
return	tok_return
break	tok_break
continue	tok_continue
send	tok_send
recv	tok_recv
Opérateur binaire	
+	tok_plus
-	tok_moins
*	tok_multiplication
/	tok_division
^	tok_puissance
%	tok_modulo
&	tok_et
|	tok_ou
Ponctuation	
(	tok_parenthese_ouvrante
)	tok_parenthese_fermante
{	tok_accolade_ouvrante
}	tok_accolade_fermante
;	tok_point_virgule
,	tok_virgule
Comparaison	
!=	tok_different
<	tok_inferieur
>	tok_superieur
<=	tok_inferieur_egal
>=	tok_superieur_egal
=	tok_affectation
==	tok_egal

La fonction next() permet de renvoyer un token.

La fonction skip() permet de passer au token suivant.

La fonction accept(t), prend en paramètre un token t, la fonction permet de vérifier si le prochain token est de type t, si oui alors elle passe au prochain token.

La fonction main(), vient lire le fichier donner en argument à la classe Lexical. Elle va incrémenter un compteur à chaque lettre lue. C'est également elle qui gère les sauts de ligne, les commentaires et la fin du fichier.

La fonction tokens(c), prend en paramètre la position d'un caractère. A partir de cette position la fonction vient créer un token. Ce token est soit déjà définit parmi les quatre catégories définit plus haut, sinon il s'agit d'un nombre dans ce cas on va créer un token constante (tok_constante) ou d'un mot auquel cas il s'agit soit d'un mot clé (catégorie définit plus haut) soit d'un identificateur. Dans ce dernier cas on va créer un token, le token : *tok_identificateur*.

L'ensemble des fonctionnalités liées à l'analyse syntaxique est valide. Les tokens sont bien créés en concordance de ce qu'il se trouve dans le fichier.

VI – Analyse syntaxique

L'analyse syntaxique se fait dans le fichier « *syntax.py* » via la classe Syntax. Dans cette classe, on peut y retrouver deux dictionnaires *op_unaire* et *op_binaire* qui permettent d'associé les nœuds de ces opérations aux instructions en assembleur. On peut y retrouver aussi le dictionnaire *tableauPriorite* qui référence les tokens opérations et qui les associes aux nœuds ainsi que de sa priorité et son associativité. La fonction qui permet de chercher à l'intérieur du dictionnaire *tableauPriorite* est la méthode chercherOp qui prend un token et renvois un dictionnaire avec les informations associées. Cette classe permet de transformer la liste des tokens de l'analyse lexicale en nœud puis de construire un arbre syntaxique. Pour cela, il y a plusieurs fonctions permettant de les transformer.

Fonction	Noeud
Primaire	nœud_constante nœud_moins_unaire nœud_plus_unaire nœud_variable nœud_recv nœud_appel_fonction
Expression	noeud_affectation noeud_different noeud_inferieur noeud_superieur noeud_inferieur_egal noeud_superieur_egal noeud_egal noeud_plus_binaire noeud_moins_binaire

	noeud_multiplication noeud_division noeud_modulo noeud_puissance
Instruction	nœud_conditionnel nœud_declaration nœud_return nœud_bloc nœud_declaration nœud_function nœud_break nœud_continue nœud_loop nœud_send nœud_expression

VII – Génération de code

La génération de code se fait dans le fichier « *genCode.py* » via la classe GenerationCode. Après avoir créé l'objet, on lance la génération avec la méthode *lancementGenerationCode* avec la liste des nœuds généré par l'analyse syntaxique. La méthode lance la génération de code sur tous les nœud et ainsi généré un fichier nommé *genCode* qui contiendra tout le code assembleur généré. A chaque nœud est associé une conditionnelle qui permet de généré le code associé.

Dans cette partie, on a rencontré des problèmes avec le nœud_loop, le nœud_conditionnel ainsi que le nœud_break et nœud_continue. On a eu des problèmes avec les labels qu'on gère via l'attribut *cpt* de la classe GenerationCode ce qui faisait que les instructions *jump* ne faisaient pas ce qu'ils devaient faire.

VIII – Analyse sémantique

L'analyse sémantique se fait dans le fichier « *semantique.py* » via la classe Analyse_semantique. C'est avec cette classe que nous gérons la pile de variables ainsi que le nombre de variable.

La fonction *debut_bloc* ajoute un dictionnaire vide dans la pile. La fonction *fin_bloc* supprime le dernier élément sur la pile. La fonction *declarer* prend en paramètre un identificateur, si l'identificateur est présent dans la pile alors on lève une erreur, sinon on l'ajoute. La fonction *chercher* permet de vérifier si une variable est déclaré auquel cas on la renvoie sinon on lève une erreur. La fonction *analyse* permet de faire le traitement des variables en fonction des déclarations, des blocs, fonctions, affectations.

IX – Arbres

Pour la gestion des arbres nous avons créé une classe *arbre* dans le fichier « arbre.py ». Cette classe contient des attributs : *type, valeur, fils (le tableaux d'enfant), slot et le nombre d'arguments (pour gérer les fonctions).* Cette classe possède également deux méthodes, la fonction *ajouterFils* qui

permet d’ajouter un arbre dans le tableaux d’enfants nommés *fils*. Et la fonction *afficher* qui est la fonction d’affichage des arbres avec l’indentation spéciale.

X – Les tests

Nous avons créé trois séries de test. Une permettant des tests unitaires dans le fichier « test_unitaire.py », une autre des tests avec les fonctions pour tester le comportement global dans le fichier « test_fonction.py ». La dernière série de test nous permet de vérifier la levé d’exception dans genCode.py et se trouve dans le fichier « test_gencode.py » Nous avons également utilisé un outil nommé *coverage* qui permet de voir la couverture de code avec nos tests. Selon cet outils nous arrivons à une couverture de code de 96 %. On vous a laissé le rapport de cet outil (format html) dans le dossier « coverage ».

Test unitaire (ici on vient tester la construction de l’arbre pas les résultats)	
+3	Permet de tester l’opérateur plus unaire
-3	Permet de tester l’opérateur moins unaire
2+3	Permet de tester l’opérateur plus binaire
2-3	Permet de tester l’opérateur moins binaire
2/3	Permet de tester l’opérateur division
2*3	Permet de tester l’opérateur multiplication
2%3	Permet de tester l’opérateur modulo
2^3	Permet de tester l’opérateur puissance
-2+3*6/2%8^2	Permet de tester la construction de l’arbre
Test fonctions	
test_fonction_recursive_un_arg	Permet de tester le bon fonctionnement d’une déclaration de fonction, ici récursive avec un seul argument
test_fonction_recursive_deux_arg	Permet de tester le bon fonctionnement d’une déclaration de fonction, ici récursive avec deux arguments
test_fonction_recursive_trois_arg	Permet de tester le bon fonctionnement d’une déclaration de fonction, ici récursive avec trois arguments
test_fonction_recursive_trois_arg_with_loops_and_break	Permet de tester le bon fonctionnement d’une déclaration de fonction, ici récursive avec trois arguments, mais aussi avec des boucles et des conditions et des breaks
test_portee_variables	Permet de tester la portée des variables
test_affichage_negatif	Permet de tester l’affichage d’un nombre négatif

test_break_else	Permet de tester la boucle for avec des conditionnelles et un break
test_boucle_for	Permet de tester la boucle for
test_boucle_while	Permet de tester la boucle while
test_boucle_while_boucle_for_break	Permet de tester la boucle while avec conditionnel et un for imbriqué et un break
test_boucle_while_boucle_for_break_continue	Permet de tester la boucle while avec conditionnel et un for imbriqué et un break et un continue
Test genCode	
test_noeud_continue	Permet de tester la levé d'une erreur si le continue n'est pas dans une boucle
test_noeud_break	Permet de tester la levé d'une erreur si le break n'est pas dans une boucle