NICOLAS BOURNEUF
RAYAN LA ROZE
APP4 INFO

RAPPORT DE PROJET

06/01/2020

Compilateur pour Ministack Machine

Le but du Projet : Réalisation d'un compilateur minimaliste.

Table des matières

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.

1. 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 tout 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. On commence d'abord par traiter les caractères spéciaux composés de plusieurs caractères. Dans le cas contraire, 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.

2. 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. A la fin de l'analyse, on obtient un arbre contenant des nœuds fournissant la priorité de l'exécution du code.

3. Analyseur Sémantique

L'analyseur sémantique récupère les nœuds déjà créés par l'analyseur syntaxique 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.

4. 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.

ELEMENTS DEMANDES

Vous trouverez dans la racine de l'archive le fichier « test.txt » qui regroupe l'ensemble des tests indiqués ci-dessous.

N.B. Le fichier README.txt explique à quel emplacement mettre les fichiers pour qu'ils soient visibles par le compilateur.

Elément	Implémenté	Commentaires	Code de test
Expression	OUI	Le traitement des expressions est implémenté dans la classe « AnalyseurSyntaxique ». Il est appelé par la méthode qui traite les instructions ou par la méthode qui traite les « primaires » dans des cas particuliers. Il commence par aller chercher un « primaire » puis un « opérateur » et se rappelle récursivement s'il a bien trouvé un opérateur. Ces appels récursifs vont permettre de créer un arbre formé de plusieurs termes « primaires » rangés par ordre de priorité.	`print(21+321*2+3*3+1*43);`

Conditionnelles		OUI	Il n'existe pas de « elseif » cependant les if/else sont implémentés et peuvent être imbriqués. Le « if » est une instruction qui prend une expression en paramètre et une instruction à réaliser si le résultat de l'expression est « != 0 ». Si une clause « else » est stipulée après le « if », son instruction sera alors exécutée en cas d'expression « ==0 ».	`var a=10;`
				`if`
				`(a==10)`
				`{`
				`  var b=a+5*3;`
				`  print(b);`
				`}`
				`else`
				`{`
				`  print(0);`
				`}`

Variables	**Déclaration**	OUI	On déclare une variable grâce au mot-clé « var ». L' analyseur syntaxique perçoit la déclaration comme une instruction. On donnera un « slot » à l'identifiant de la variable afin de pouvoir y accéder dans les blocs et sous-blocs où elle sera manipulée.	`var vari=3+4;`
				`var varia=4;`
				`var a=10;`

Variables	**Utilisation**	OUI	On peut la déclarer et lui affecter directement une valeur. Cependant la déclaration étant perçue comme une instruction il faut déclarer une variable avant de l'utiliser comme paramètre d'une boucle « for » ou autres éléments prenant en paramètre des « expressions » et non des « instruction ». Lors de l'utilisation, la variable est vue comme un « primaire » et est donc utilisée grâce à « Expression() ».	`print(a);`
	Affectation	OUI	L'opérateur « = » permet d'affecter une « rvalue » à une « lvalue », la « rvalue » étant une « expression » et la « lvalue » étant l'identifiant de la variable.	`var b=a+5*3;`

	Portée	OUI	Une variable doit être contenue à minima dans une fonction. Elle pourra être utilisée dans le code qui suit sa déclaration mais uniquement dans son bloc et les sous-blocs de celui-ci.	`{`
				`  var a = 5;`
				`  print(a);`
				`  {`
				`  print(a);`
				`  }`
				`}`
				`print(a);`

<table>
<tr><td rowspan="3">Boucles</td><td>for</td><td>OUI</td><td>La boucle for est perçue comme une instruction par l'analyseur syntaxique. Elle prend en paramètre 3 expressions. On exécute une fois la première expression. Tant que la deuxième expression est valide l'instruction de la boucle sera exécutée suivit de la troisième expression. Il est possible de mettre une « break » pour sortir de la boucle ou un « continue » pour passer à l'itération suivante.</td><td><pre>for(i=0; i<15;i++){
    if(i==2) continue;
    if(i==3){
        for(r=1;r<4; r++){
            print(r*(-1));
        }
    }
    print(i);
    if(i==4) print(Power(2*2,6));
    if(i==5) break;
}</pre></td></tr>
<tr><td>while</td><td>OUI</td><td>Le while est une instruction qui prend en paramètre une expression. Tant que l'expression est valide on exécute l'instruction contenue dans la boucle. Il est possible de mettre une « break » pour sortir de la boucle ou un « continue » pour passer à l'itération suivante.</td><td><pre>while
(a>5)
{
print(a);
a--;
}</pre></td></tr>
<tr><td>do while</td><td>OUI</td><td>Fonctionne de la même façon que le « while » à part l'instruction qui est placée entre le « do » et le « while ».</td><td><pre>do
{
print(r);
r--;
}
while(r!=0)</pre></td></tr>
</table>

Fonction	**Déclarations**	OUI	On utilise le mot-clé « function » pour stipuler que l'identifiant qui suit est une fonction. Elle est suivie de paramètres (0…n) placés entre parenthèses. L'analyseur syntaxique commence son analyse en recherchant les déclarations de fonctions. C'est cette méthode qui appellera par la suite les instructions, les instructions appelleront les expressions, enfin les expressions appelleront les primaires. C'est donc pour ça qu'il est important de ne mettre aucun code en dehors des fonctions. Le code à exécuter doit être contenu dans la fonction « main() ». D'autres fonctions peuvent être implémentées avant le « main() » pour pouvoir les utiliser à l'intérieur. Pour retrouver la fonction on lui attribue un « slot » comme pour les variables. Les noms de fonctions ne peuvent contenir que des lettres.	`function test(a){`
				`  print(a+10);`
				`}`

Fonction	**Appels**	OUI	L'appel des fonctions se situe au niveau des « primaires » et est donc trouvé lors de la recherche d'une expression. Pour la différencier d'une variable on regarde si une parenthèse ouvrante suit l'identifiant.	`test(vari);`
	Passage d'arguments	OUI	Lors de la déclaration de la fonction les paramètres sont représentés comme des variables utilisées pour le code de la fonction. Puis à l'appel de la fonction ces variables prennent la valeur de celles passés en paramètre identifiées grâce à l'analyseur sémantique. Lors de l'appel d'une fonction on vérifie que le nombre d'arguments passé est correct.	`test(vari);`

Pointeur et tableau	**Pointeur**	OUI	Afin d'identifier un pointeur, le token « tok_multiply » est recherché dans la fonction « primaire() » en plus d'être un opérateur. Si la fonction « primaire() » détecte ce token, on sait qu'elle doit rechercher la case mémoire représentée par cette expression.	`var b = 4;`
				`*(b+1) = 3;`
				`print(*(b+1));`

<table>
<tr><td rowspan="2">Pointeur et tableau</td><td>Tableau</td><td>OUI</td><td>Le tableau se sert des pointeurs, on définit une variable représentant la première case mémoire qu'on additionne à la case mémoire à laquelle on veut accéder dans le tableau. Cependant pour éviter de déborder sur d'autres cases mémoires il faut penser à allouer de la mémoire au tableau avant de l'utiliser. Nous avons aussi rajouté un peu de sucre syntaxique pour que l'utilisateur mette le numéro de la case mémoire entre crochet derrière la variable qui représente le tableau.</td><td><pre>var t = alloc(10);

for(i=0; i<10; i++){
    t[i] = 10-i;
}</pre></td></tr>
<tr><td>Fonction d'allocation</td><td>OUI</td><td>Expliquée dans la partie 3.</td><td><pre>var t = alloc(10);</pre></td></tr>
</table>

FONCTIONS DE LA LIBRAIRIE STANDARD IMPLEMENTEE

Le fichier « standar.lib » contient l'ensemble des fonctions mentionnées ci-dessous.

N.B. Le fichier README.txt explique à quel emplacement mettre les fichiers pour qu'ils soient visibles par le compilateur.

Fonction	Paramètres	Commentaires	Code de test
Power	a, b : entier	La fonction retourne la partie entière de a exposant b.	`print(Power(2,4)+1);`
factorial	a : entier	La fonction retourne !a.	`print(factorial(10));` `print(!factorial(-50));`

printx	a : entier à afficher	La fonction écrit le nombre dans le terminal.	`function print(a)`
			`{`
			`  //Si a vaut zero cela correspond au code ascii 48`
			`  if(a==0) send 48;`
			`  //Sinon on appelle printx pour décomposer le nombre`
			`  else printx(a);`
			`  //on saute une ligne après écriture`
			`  send 10;`
			`}`

print	a : entier à afficher	La fonction écrit le nombre dans le terminal grâce à printx(a).	`print(testIncr++);`

alloc	n : entier de case mémoire à allouer	La fonction alloue n bits d'espace pour un tableau.
		Elle est également utilisée dans la fonction init qui appelle le main afin d'allouer de la mémoire au programme.
			`function init(){`
			`  alloc(Power(42, 3)-9000);`
			`  main();`
			`}`

DIFFICULTES RENCONTREES

- Gestion des boucles et des conditions imbriquées : Dans un premier temps nous avions une gestion des boucles et conditions de manières statique ce qui empêchait l'utilisation de boucles imbriquées car on ne savait pas quand les boucles/conditions se terminaient. Pour y remédier nous avons utilisé des piles dans lesquels on rajoute un « bloc » représentant une condition ou une boucle à chaque fois qu'il y a une déclaration. Ce bloc contient des informations concernant les flags de la boucle/condition.
- Analyseur sémantique : On ne peut pas déclarer des fonctions/variables en dehors de blocs, afin de résoudre ce problème on a simulé l'ouverture d'un bloc dans le Main() de notre compilateur.
- Implémentation du « continue » : afin de permettre l'utilisation de « continue » nous avons rajouté un « flag » avant le code de la troisième expression d'un « for » ou avant le flag de fin du « while ». Ainsi, lorsque l'utilisateur utilise un « continue » on peut ignorer les instructions suivantes et sauter directement à l'endroit approprié.
- Les classes « AnalyseurSemantique » et « CodeGenerateur » ont leurs méthodes implémentées en statique car elles n'ont pas de traitement et données spécifiques à une instance. Elles n'ont pas vocation à être instanciées mais leurs méthodes et attributs doivent être utilisées par d'autres classes pour modifier des nœuds. C'est donc la solution la plus appropriée que nous ayons trouvée pour répondre à des besoins d'analyse tout en gardant un cadre logique et chronologique au développement du compilateur.
- Passage de paramètres à une fonction : dans l'analyseur sémantique on oubliait de faire défiler récursivement l'analyse sémantique sur les enfants du nœud d'appel de fonction, on ne pouvait donc pas récupérer les bons « slots » attribué aux variables passé en paramètre. Les valeurs n'étaient donc pas les bonnes.
- Analyse lexicale : Lors de la détection de doubles caractères tel que « == » on ne vérifiait pas que l'indice (i+1) était bien dans la ligne ce qui entrainait des erreurs de type « out of bounds ».
- Ajout de librairie : en chargeant les librairies en même temps que le code, l'indication des lignes/colonnes (des erreurs) dans le code étaient décalée. Pour y remédier nous analysons le code des librairies en ignorant la gestion des erreurs. Nous considérons que la librairie standard est développée par les concepteurs du compilateur et est donc sans fautes.
- Ajout de l'obligation d'un point-virgule afin de délimiter les « expression » : lorsqu'on n'avait pas de points-virgules la détection de deux expressions séparées pouvait être mal interprétée. Le point-virgule permet donc de délimiter les expressions.