UCA - Licence Informatique

TP Compilation - N3, S5
TP Noté

Instructions. L'objectif de ce TP c'est de produire un langage de programmation dont les calculs sont des expressions arithmétiques et Booléennes, et le compilateur doit génèrer instructions en langage machine MVaP. Il est à faire en **binôme ou trinôme**, et vous rendrez un fichier Calculette_Nom1Prenom1_Nom2Prenom2_Nom3Prenom3.g4 qui contient la grammaire attribuée de votre langage de programmation, que l'on compilera avec ANTLR pour produire le compilateur. La date limite de rendu est le **Vendredi 07 Janvier 2022 à 12h**, les dépôts, par groupe de TP, seront ouverts le **lundi 03 Janvier 2022 à 8h**. Il y aura une séance de questions par binôme/trinôme, vous serez informés des créneaux dans la semaine du 03 Janvier 2022. Il n'y a pas de présentation à faire pendant la séance de questions, mais nous testerons devant vous le compilateur et poserons des questions sur votre grammaire attribuée.

Instructions du langage de programmation. Chaque instruction se terminera par défaut par un saut de ligne, mais vous pourrez l'enrichir, si vous voulez, en ajoutant d'autres délimiteurs d'instructions. Voici les principales instructions du langage de programmation :

- Déclarer des variables de type entier, Booléen et flottant. Chacune des variables aura une valeur par défaut à la déclaration.
- Une variable entière ne peut contenir que des expressions arithmétiques entières, une variable Booléenne ne peut contenir que des expressions Booléennes et une variable flottante peut contenir des expressions arithmétiques entières comme flottantes.
- Affecter des valeurs (ou expressions) à des variables et utilisation de variables dans des expressions.
- Utiliser des expressions conditionnelles (imbriquées ou pas).
- Utiliser des expressions itératives (imbriquées ou pas).
- Utilisation de fonctions d'entrées-sorties.

Un programme va avoir la structure suivante : on fait d'abord des déclarations de variables, puis les différentes instructions. Vous pouvez par exemple vous inspirer du début de grammaire suivante (dont la règle d'entrée c'est calcul) :

```
calcul returns [ String code ]
@init{ $code = new String(); } // On initialise $code, pour ensuite l'utiliser comme accumulateur
@after{ System.out.println($code); } // on affiche le code MVaP stocké dans code
: (decl { $code += $decl.code; })*
NEWLINE*
(instruction { $code += $instruction.code; })*
{ $code += " HALT\n"; }
;


finInstruction
    : (NEWLINE |  ';')+
    ;

decl returns [ String code ]
    : TYPE IDENTIFIANT finInstruction
{
```

```
            // à compléter
}
;

instruction returns [ String code ]
: expression finInstruction
{
   //à compléter
}
| assignation finInstruction
{
            // à compléter
}
| finInstruction
{
$code="";
}
;
assignation returns [ String code ]
: IDENTIFIANT '=' expression
{
         // à compléter
}
;
// lexer
TYPE : 'int' | 'float' | 'bool'; // pour pouvoir gérer des entiers, Booléens et floats
IDENTIFIANT : ; //à compléter
```

Fichers de test. Dans le dossier Fichiers TP (cours en ligne), vous trouverez le fichier archivé tests-code.zip, qui contient un ensemble de fichiers de tests, qui seront utilisés pour évaluer le travail rendu.

Top départ. Commencez par créer le fichier .g4 de votre binôme/trinôme et y ajouter les règles génèrant du code MVaP pour des expressions arithmétiques et Booléennes. Comme dans l'exemple ci-dessus, il faut que votre point d'entrée s'appelle calcul.

Gestion des variables. Pour pouvoir gérer les variables, il faut utiliser une table des symboles qui permet de stocker pour chaque variable son type, son nom et d'autres informations permettant de vérifier si les affectations sont correctes, si une variable est déclarée lorsqu'on l'utilise dans une expression ou pour génèrer le code MVaP associé à la manipulatin des variables, etc. Pour faire simple, je vous propose d'utiliser le type HashMap. L'exemple suivant permet par exemple de garder les paires (identifiant, type). On utilise le namespace members de ANTLR pour des variables et fonctions globales pour le parser.

```
@parser::members {
// pour avoir une table des symboles, vous pouvez la modifier pour avoir le comportement voulu
 HashMap<String, String> tablesSymboles = new HashMap<String, String>();
}
```

Ajoutez à votre grammaire les règles pour manipuler des variables. Attention, vous devrez faire les tests de compatibilité avant toute affectation (ou utilisation de variables dans une expression). Vous devrez aussi connaître l'adresse de chaque variable dans la pile pour pouvoir y faire référence dans les instructions MVaP.

Gestion des E/S. Pour faciliter la suite, il faut que l'on puisse lire et écrire des valeurs à stocker dans des variables. Pour cela, on va utiliser les instructions lire(x) qui affecte la valeur entrée par l'utilisateur à la variable x, et afficher(exp) qui affiche à l'écran la valeur de l'expression exp. Ajoutez aux bons endroits les règles permettant de gérer les entrées-sorties.

Blocs. Pour pouvoir gérer les structures conditionnelles et les boucles avec plusieurs instructions, il faut que l'on puisse identifier les instructions à l'intérieur d'une structure conditionnelle ou itérative, des instructions en dehors. Ajouter à votre grammaire la possibilité de manipuler des blocs d'instructions qui seront encapsulées entre des accolades : {...}.

Boucles répéter ...tant que. On voudrait maintenant gérer les boucles. La syntaxe sera (*cond* est une expression booléenne)

```
repeter
    instructiondo
tantque (cond)
```

Le sens d'une boucle repeter ... tantque c'est de répéter le corps de la boucle tant que la condition est vraie, et on exécute au moins une fois le corps de la boucle. Ajouter les règles nécessaires pour manipuler les boucles repeter ... tantque. Il faudra que l'on ait la possibilité d'écrire des boucles du genre

```
repeter  // 1 seule instruction sans accolades
tantque (cond)


// ou des blocs d'instruction
repeter {
//plusieurs instructions
}
tantque (cond)
```

Structures conditionnelles. Nous voulons ajouter des branchements à notre langage avec la possibilité de faire des tests. La syntaxe sera

```
si (cond)
   instructionSi
sinon instructionSinon
```

Le sinon sera facultatif. Aussi, les instructions du si ou sinon doivent pouvoir être une seule instruction (et donc sans accolades) ou des blocs d'instructions. Ajouter les règles nécessaires pour la manipulation des structures conditionnelles.

Exposants. Ajouter aux expressions arithmétiques la possibilité d'avoir des exposants, avec le symbole ^(qui est prioritaire sur le produit).