Projet Compilation

N3 informatique

2022-2023

1 Instructions générales

Le travail est à réaliser en équipe de deux (binôme) ou trois (trinôme), tous du même groupe TP. Le rendu doit être fait sur **l'espace moodle du cours**, avant le **13 janvier, 21h00**. Aucun rendu en retard ne sera accepté.

Un seul rendu par équipe est attendu. Ce rendu consiste en un unique fichier `Calculette.g4` qui doit impérativement compiler (*i.e.*, la commande `antlr4 Calculette.g4 && javac Calculette*.java` doit fonctionner) [1]. Les premières lignes du fichier doivent être des lignes commentées contenant l'identification des différents membres de l'équipe, et le numéro de l'extension implémentée (*c.f.* section 4), comme sur le modèle suivant :

```
//membre 1: Prénom NOM (numéro)
//membre 2: Prénom NOM (numéro)
//membre 3: Prénom NOM (numéro)
//extension: 2, 4

grammar Calculette;
...

...

//bug constatés
//fonctionalités non-implémentées
//autres commentaires
```

où chaque *numéro* correspond au numéro étudiant, les extensions implémentées sont *2* et *4* et correspondent aux extensions *"incrément et décrément"* et *"Opérateur d'arrondi pour les flottants"* spécifiées en section 4.

En fin de fichier, vous pouvez écrire, en commentaire, les différents manques de votre compilateur (*e.g.*, fonctionnalités non implémentées, bugs constatés mais non corrigés), ou autres commentaires.

1. Toutes les méthodes Java que vous aurez écrites doivent être incluses dans le fichier Calculette.g4 dans l'entête `@parser::members{}`.

Vous pouvez soumettre plusieurs fois votre travail (vous y êtes même encouragés), la dernière soumission seule sera prise en compte pour l'évaluation. [2] Ceci vous permet de ne pas attendre la dernière minute pour envoyer votre projet et ainsi éviter de risquer de ne rien envoyer du tout.

2 Sujet

Vous devez écrire un compilateur permettant de traduire en code MVaP du code écrit dans le *langage de calculette* défini dans ce sujet.

La définition du langage source (langage de calculette) se fait en deux temps. Il y a d'une part le langage de base (section 3) que votre compilateur doit supporter, et, d'autre part, un ensemble d'extensions proposées (section 4). Vous devez implémenter **au moins deux extensions** et indiquer votre choix dans l'entête commenté de votre fichier (*c.f.* modèle en section 1).

Le langage cible est le langage de MVaP. Le code MVaP généré par votre compilateur devra toujours terminer par les commandes `WRITE` (pour afficher le résultat final même si cela n'a pas été explicitement demandé dans le code source) et `HALT` (pour terminer l'exécution).

3 Langage source de base

Le langage de calculette (qui enrichit celui vu en TP) manipulera trois types d'expression : entières, flottantes et booléennes. Un programme dans ce langage consistera en **une** expression (entière, booléenne ou flottante, *c.f.* section 3.4), éventuellement précédée, dans cet ordre, d'une séquence de déclarations de variables (*c.f.* section 3.2) et d'une séquence d'instructions (*c.f.* section 3.3). Le résultat d'un programme sera toujours le résultat de l'expression du programme.

3.1 Délimiteurs

Les trois parties principales du code sus-mentionnées (déclarations, instructions, expression finale) seront séparées par un délimiteur, qui peut être un retour à la ligne ou un point-virgule. Ce délimiteur sera également utilisé pour séquencer les déclaration et les instructions. Il pourra toujours être précédé ou suivi d'un nombre arbitraire de retours à la ligne, mais il ne doit pas être possible d'avoir deux point-virgules successifs (même avec des retours à la ligne entre les deux), *e.g.* `int x; x=3; ; 3*x` et `int x; x=3;\n; 3*x` sont des codes invalides que votre compilateur doit rejeter.

2. Vérifiez que le fichier soumis compile avant de soumettre !

3.2 Déclarations

Les déclarations, nécessairement faites en début de programme, sont séparées par un délimiteur ou un retour à la ligne. Chaque déclaration précise un nom de variable et son type (entier, booléen ou flottant).

3.3 Instructions

Avant une expression, un programme peut effectuer une séquence d'instructions. Ces instructions permettent :

- les affectations et ré-affectations des variables (une variable préalablement déclarée prend la valeur d'une expression de même type) ;
- l'affichage de la valeur d'une expression : `afficher(`⟨expr⟩`)` ;
- des instructions conditionnelles, possiblement imbriquées de la forme suivante : `si` ⟨exprC⟩ `alors` ⟨A⟩ `sinon` ⟨S⟩.
- des instructions itératives, possiblement imbriquées de la forme suivante : `repeter` ⟨instr⟩ `jusque` ⟨exprC⟩ (qui exécute l'instruction ⟨instr⟩ au moins une fois et tant que la condition ⟨exprC⟩ n'est pas satisfaite).

3.4 Expressions

Le langage des expressions devra permettre :

- les expressions arithmétiques bien parenthésées sur les entiers et sur les flottants avec les opérateurs standards : `+`, `*`, `-` (à la fois binaire et unaire) et `/` (division flottante, *e.g.*, `3/4` doit donner `0.75`). Les priorités habituelles devront être respectées.
- les expressions booléennes bien parenthésées avec les opérateurs standards : `et`, `ou` et `non`. Les priorités suivantes devront s'appliquer : `non`>`et`>`ou`.
- les comparaisons strictes et larges binaires (*i.e.*, entre deux éléments) d'expressions entières et flottantes, avec les opérateurs standards : `<`, `<=`, `>`, `>=`, `==` et `<>` (différent).
- les opérateurs supplémentaires suivants sur les expressions entières et flottantes : `min(`⟨expr⟩`,`⟨expr⟩`)` et `max(`⟨expr⟩`,`⟨expr⟩`)`.
- les opérateurs supplémentaires suivants sur les expressions entières (mais pas sur les flottants) : `//` (quotient dans la division euclidienne), `%` (modulo, c'est à dire le reste dans la division euclidienne), `^` (puissance) et `abs(`⟨expr⟩`)` (la valeur absolue). Les priorités suivantes devront être appliquées : `^`>`%`>`//`.
- l'opération d'entrée `lire()` : lecture d'un entier, d'un flottant ou d'un booléen, selon le contexte, *e.g.*, `3 + lire()` vaudra 7 si l'utilisateur a rentré 4 et vaudra 5.6 s'il a rentré 2.6.
- des expressions (entières, flottantes ou booléennes) conditionnelles, possiblement imbriquées : `si` ⟨exprC⟩ `alors` ⟨A⟩ `sinon` ⟨S⟩ où ⟨A⟩ et ⟨S⟩ sont des expressions (booléenne, entière ou flottante) de même type.

— des blocs d'instructions suivi d'une expression, le tout délimité par des accolades, *e.g.*, `3 * x+{x=2 * x;x}` vaudra $3\times2+4$ si `x` vaut initialement 2. Dans cet exemple, `{x=2 * x;x}` aura comme résultat la valeur de `x` qui est l'expression de `{x=2 * x;x}` et du coup on peut l'utiliser dans une expression arithmétique.

4 Extensions

4.1 Bloc de déclarations enrichi

L'extension propose une syntaxe plus pratique pour les déclarations et affectations initiales des variables :

— d'une part chaque déclaration pourra être suivie d'une affectation, *e.g.*, `int x=3 * 8; float f;`.

— d'autre part les déclarations de variables de même type pourront être factorisées de façon à n'indiquer qu'une seule fois le type pour un ensemble de variable à déclarer, *e.g.*, `int x, y, z; float f,g; int u;`.

Ces deux points de l'extension pourront être combinés ensemble, par exemple :

```
int x=3, y, z=3 * x
float z=x/7, u, w
int y2=x++
...
```

4.2 Incrément et décrément dans une expression

L'incrément et le décrément de variable est déjà intégré de base, en tant qu'instruction (section 3.3). Cette présente extension autorise à utiliser ces opérations au sein d'une expression. Ceci nécessite d'attribuer une valeur à cette "instruction" (*e.g.*, que vaut `3 * x++` ?). Cette valeur sera la valeur de la variable incrémentée ou décrémentée. L'extension propose les syntaxes préfixe et suffixe qui définissent différemment la valeur à intégrer à l'expression :

— préfixe (*e.g.*, `++x`, `--x`) : la valeur de la variable après incrément/décrément

— suffixe (*e.g.*, `x++`, `x--`) : la valeur de la variable avant incrément/décrément

Par exemple, `x=1; 3 * (x-- * ++x)` vaudra 3. Idéalement, votre compilateur devra interdire les formes `x+++y` et `x---y`, mais autoriser `x++ +y--`, `x+ ++y`, `x-- -y`, `x- --y`, `x++ * y`, *etc...* À défaut, il devra gérer convenablement l'ambigüité de tels codes.

4.3 Variable tableau typé

L'extension propose de gérer des variables tableaux. Il faudra donc gérer :

— la déclaration, *e.g.*, `int array[4] mytab` (un tableau d'entiers à quatre éléments),

— l'affectation, *e.g.*, `mytab[0]=8` et/ou `mytab=[2,3,1,0]`,

— l'accès, *e.g.*, `3 * mytab[2]` ou `mytab[myvar]` ou `mytab[i-2]`.

En bonus,

— les accès relatifs, *e.g.*, `mytab[-1]` (le dernier élément), `mytab[-2]` (l'avant-dernier élément), *etc...*

— les déclarations utilisant une longueur qui est une expression, *e.g.*, `int array[2 * 8] mytab`.

4.4 Opérateur d'arrondi des flottants

Cette extension propose d'ajouter l'opérateur binaire `round(`⟨exprF⟩`,`⟨exprN⟩`)` aux expressions flottantes. Cet opérateur devra arrondir la valeur de l'expression flottante ⟨exprF⟩ à précision 10^{-p} près où p est la valeur de l'expression entière ⟨exprN⟩. Par exemple, `round(3.4,0)` vaudra 3.0, `round(3.55,1)` vaudra 3.6 et `round(32.5,-1)` vaudra 30.0.

4.5 Sous-programmes

L'extension autorise à effectuer de nouvelles déclarations, dans un bloc d'instructions délimité par des accolades. (Ils deviennent ainsi des sous-programmes.) Ces déclarations devront être locales à ce bloc et devront masquer localement les variables préalablement définies en cas de conflit. Par exemple :

```
int x
int y
x=2
y=3 * x
4 * { int x; x=2 * y; 3 * x * y } + x
```

devra donner $4 \times (3 \times (2 \times 6) \times 6) + 2 = 86$.

4.6 Boucles `pour`

Cette extension propose d'ajouter la possibilité d'écrire une boucle `pour` (*for*), de la forme suivante : `pour i dans` ⟨Nstart⟩`,...,`⟨Nstop⟩ `faire` ⟨instr⟩ où ⟨instr⟩ est soit une instruction simple (*e.g.*, `x=2 * x`), soit une séquence d'instructions (séparées par des point-virgules ou des retours à la ligne) délimitée par des accolades (*e.g.*, `{ x=2 * x; y++; afficher(x);x=x-2; }`). Les boucles doivent pouvoir être imbriquées. La variable utilisée pour la boucle (`i`) doit être une variable entière locale. En particulier, elle ne nécessite pas de déclaration en début de programme. Si une variable globale de même nom avait été déclarée, elle sera masquée à l'intérieur de la boucle. Voici un exemple complexe de code qui devrait fonctionner :

```
int i
int x
```

```
i=2
x=0
pour i dans 0,...,2*i faire {
  x=x*2+i
  x++
}
x - i
```

et qui devrait donner le résultat 55 (la boucle fait passer `x` de 0 à 57 et, dans l'expression finale `x - i`, la variable `i` correspond à la variable globale i dont la valeur est 2).