TP5 à TP9 -- L3 -- Langage et Compilation

Compiler un langage de calculatrice scientifique en MVàP

A - Objectifs et Rythme de cette série de TP

L'objectif principal de cette série de TP est de vous aider à finaliser un travail qui tiendra lieu de DM.
Nous basculons dans une organisation moins rigide afin de vous permettre d'avancer à votre rythme. La dernière semaine sera l'occasion de nous expliquer plus en détail votre travail lors d'une petite soutenance afin que nous l'évaluions de manière sommative.
Nous allons toutefois vous fournir le moyen de vérifier seul l'avancement de votre travail en auto-correction avec une série de tests. Des tests identiques et d'autres similaires seront appliqués automatiquement à votre travail pour déterminer une partie importante de votre note finale. Pour plus de détails concernant l'évaluation de votre travail reportez vous svp à la page du cours, où plus d'informations seront apportées à ce sujet.
L'organisation du cours et des TDs vous permet de traiter les sections suivantes dès le TP indiqué entre parenthèse.

1. Calcul d'expression arithmétique sur la MVàP (TP5)
2. Gestion des variables globales (TP6)
3. Gestion des entrées et sorties (TP6)
4. Les incrémentations et les regroupements (TP6)
5. Le while (TP6)
6. Conditions simples (TP6)
7. Combinaisons booléennes (TP6)
8. Branchements (if) (TP6+)
9. Le for (TP6+)
10. Fonctions (TP7)
11. Cas de base des flottants (TP8)
12. Quelques programmes classiques (TP8)
13. Autres améliorations (TP8 / TP9)

Rappels sur la manière d'utiliser antlr

On commence par faire en sorte que java sache où trouver antlr en configurant le `CLASSPATH` correctement (mettre la ligne adéquate dans `.bashrc` est une bonne idée).
On génère du code java pour le lexeur et le parseur à partir de la grammaire antlr (fichier `.g4`) avec

```
java org.antlr.v4.Tool ma_grammaire.g4
```

On compile ces fichiers générés qui ont tous comme préfixe le nom de la grammaire.

```
javac ma_grammaire*.java
```

On utilise par exemple pour visualiser l'arbre :

```
java org.antlr.v4.runtime.misc.TestRig ma_grammaire 'monaxiome' -gui < fichierentre
```

ou encore

```
java org.antlr.v4.runtime.misc.TestRig ma_grammaire 'monaxiome' -gui
```

avec saisie des entrées et fin par `control D` (pour générer `EOF`).

Dans un second temps, on pourra enlever l'option `-gui` et rediriger la sortie standard vers un fichier (avec `> fichiersortie`)
Changer l'axiome (règle de départ du parseur) permet de tester directement une partie bien choisie de la grammaire.

Sur les machines de la fac, vous pouvez utiliser les alias

- `antlr4` raccourci pour `java org.antlr.v4.Tool`
- `antlr4-grun` raccourci pour `java org.antlr.v4.runtime.misc.TestRig`

B - Génération de code MVàP

La machine MVàP

Notre compilateur va générer du code exécutable par la machine MVàP.

> **Étape :**
>
> - Pour construire cette machine, récupérez les sources de la MVàP , et suivez les indications données dans le fichier `ALIRE.md` ou utilisez le `Makefile` .
> - Les instructions pour assembler et exécuter du code MVàP sont également présentées dans le fichier `ALIRE.md`
> Assurez vous de bien comprendre son fonctionnement en testant la machine sur les fichiers `.mvap` fournis.

On va maintenant générer du code MVàP. Pour cela, on va associer à chaque expression un attribut `code` , de type chaîne de caractères, qui va contenir le code MVàP pour l'exécution de cette expression.

On débute avec un squelette de code Antlr muni d'une règle d'entrée **calcul** qui se compose d'une suite d'instructions. Les éléments @init et @after permettent d'indiquer respectivement des actions à effectuer avant et après l'application de la règle.

```
calcul returns [ String code ]
@init{ $code = new String(); }   // On initialise une variable pour accumuler le code
@after{ System.out.println($code); } // On affiche le code effectivement produit

    :
        NEWLINE*

        (instruction { $code += $instruction.code; })*

        { $code += "   HALT\n"; }
    ;

instruction returns [ String code ]
    : expression finInstruction
        {
            // à compléter
        }
```

```
    | finInstruction
        {
            $code="";
        }
    ;

expression returns [ String code ]
    :
    // à compléter
    ;

finInstruction : ( NEWLINE | ';' )+ ;
```

Question : Compléter la grammaire ci-dessus pour générer du code MVàP effectuant le calcul de l'expression. On fera attention à bien choisir le comportement adéquat dans le lexeur pour le jeton NEWLINE qui peut servir maintenant à terminer une instruction. On pourra réutiliser la grammaire de la calculette du précédent TP. On ajoutera le support du moins unaire

```
-8+-50+100
```

ainsi que du modulo

```
240%99
```

en faisant attention à leur priorité.

Note : à la place de `antlr4-grun` qui, essentiellement, crée un programme principal qui effectue l'analyse lexicale et syntaxique, on peut définir son propre programme principal `MainCalculette.java` instanciant un lexeur, puis un parseur.

```
import java.io.*;
import org.antlr.v4.runtime.*;
public class MainCalculette {
    public static void main(String args[]) throws Exception {
        CalculetteLexer lex;
        if (args.length == 0)
            lex = new CalculetteLexer(CharStreams.fromStream(System.in));
        else
            lex = new CalculetteLexer(CharStreams.fromFileName(args[0]));

        CommonTokenStream tokens = new CommonTokenStream(lex);

        CalculetteParser parser = new CalculetteParser(tokens);
        try {
            parser.start();    // start l'axiome de la grammaire
        } catch (RecognitionException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Tests : Tester de manière approfondie avant de passer à la suite. Vous devez être capable :

- de compiler votre fichier g4
- d'exécuter la calculette pour obtenir le code MVàP

- d'assembler le code MVàP
- d'interpréter le code obtenu pour afficher la valeur de l'expression

Vous pouvez vérifier avec les exemples suivants qui doivent tous donner la même réponse.

```
42
24+24-6
5*8+2*1
1194/45%16*4+2
6*4/5+38
42+1+2+-3
5*8+2*-1/-1
(5*6*7*11 + 2)/11*5-1008
(5*6*7*11 + 2)/(11*5)
(5*6*7*11 + 2)/11/5
(5*6*7*11 + 2)/(11/5)-1114
```

Ces opérations sont exigibles lors de la soutenance.

Question : Ajouter à la grammaire le support des commentaires monolignes (avec le symbole `//`) et multilignes (entre les balises `/*` et `*/`).

C - Traitement des variables

On va enrichir le langage de notre machine à calculer scientifique pour autoriser les variables globales. On pourra déclarer une variable (entière) x avec l'instruction `int x`. Une telle variable a une valeur de 0 par défaut. Pour l'assignation on fera par exemple `x=3` ou encore `x=3+y` (on accède à la valeur d'une variable -- ici y -- via son identifiant). On va aussi considérer des variables réelles, qu'on déclarera de manière similaire en écrivant `double r`. Pour les calculs et affectations, il faudra dans un premier temps faire attention à ce que les types soient les mêmes. L'analyse syntaxique de l'emploi des variables est assurée par des règles supplémentaires :

```
decl returns [ String code ]
    :
        TYPE IDENTIFIANT finInstruction
        {
            // à compléter
        }
    ;

assignation returns [ String code ]
    : IDENTIFIANT '=' expression
        {
            // à compléter
        }
    ;

// lexer
TYPE : 'int' | 'double' ;

//...
```

On modifie la règle d'entrée **calcul** pour ajouter une suite de déclarations avant la suite d'instructions.

```
calcul returns [ String code ]
@init{ $code = new String(); }   // On initialise une variable pour accumuler le code
@after{ System.out.println($code); } // On affiche le code effectivement produit
```

```
    :   (decl { $code += $decl.code; })*

        NEWLINE*

        (instruction { $code += $instruction.code; })*

        { $code += "   HALT\n"; }
    ;
```

On ajoute l'assignation parmi les instructions:

```
instruction returns [ String code ]
    :
        //...
    | assignation finInstruction
        {
                // à compléter
        }
        //...
    ;
```

Question : Compléter votre grammaire antlr en ajoutant aux bons endroits ces règles et en modifiant les règles **calcul** et **instruction**. Attention à l'ordre des déclarations dans le lexeur : les mots-clés réservés ne peuvent pas être des identifiants.

Pour traiter des variables globales (ainsi que plus tard des paramètres de fonctions qu'on appellera variables locales) il faut avoir une table de symboles (des tables pour plus tard). Comme nos variables sont typées, il va falloir stocker pour chaque variable, non seulement son adresse dans la pile mais aussi son type et le fait qu'elle soit locale ou globale. Pour ceci nous vous fournissons:

- une classe `VariableInfo` qui sert à encapsuler les informations d'une variable,
- une classe `TableSimple` qui met en œuvre une table de hachage,
- une classe `TablesSymboles` qui sert à stocker la table des Symboles, c'est cette dernière que nous utiliserons.

On va utiliser directement la classe `tablesSymboles` (plusieurs tables). Pour cela, il suffit d'ajouter au début du fichier g4 le code suivant:

```
@members {
   private TablesSymboles tablesSymboles = new TablesSymboles();
        }
```

On peut ensuite ajouter une variable globale entière à la table (ne pas oublier en même temps de réserver un espace dans la pile) à l'aide de:

```
tablesSymboles.addVarDecl($IDENTIFIANT.text,"int");
```

Et récupérer ses informations avec:

```
VariableInfo vi = tablesSymboles.getVar($IDENTIFIANT.text);
```

Il est possible d'accéder aux champs avec vi.address, vi.type et vi.scope.

Question : Ajoutez la génération de code pour les variables globales (entières

seulement pour l'instant).

Question : On change un petit peu la syntaxe de notre calculette. On autorise maintenant une déclaration et assignation par un entier en une seule instruction. Par exemple, `int x = 42`. Modifiez votre grammaire en conséquence.

D - Traitement des entrées / sorties

Maintenant que nous avons des variables, nous allons ajouter la possibilité pour la calculatrice scientifique de lire une valeur avec les instruction `input(x)` qui lit l'entier saisit par l'utilisateur et donne cette valeur à la variable `x`. Nous ajoutons son pendant `output(x)` qui affiche la valeur de `x`. Pour l'instant, vous pouvez travailler uniquement avec des variables entières, mais à terme on travaillera également avec les flottants.

Question : Ajoutez la génération de code pour ces deux instructions.

Pour la suite du sujet, nous vous mettons à disposition des *autotests* permettant de valider par vous même certains aspects du travail. Pour cela, vous trouverez dans le dossier BMs des exemples de programmes. Le dossier se compose:

- du dossier `calcs` contenant les programmes de test
- du dossier `read` contenant (au besoin) les entrées à donner au programme
- du dossier `writes` contenant le résultat attendu

Sur les machines de l'université, vous disposez également d'un script shell `tp-compil-autocor` qui lance automatiquement les différents tests. Pour cela, vous devez lui fournir:

- votre fichier `.g4` qui doit contenir une règle **instruction**
- les fichiers `.java` nécessaires à la compilation.

Ce script crée un dossier `Compil-`*`date`* contenant pour chaque programme de test les fichiers intermédiaires suivants :

- Le *log* de la compilation avec votre programme (fichier `.comp.log`)
- Le *MVaP* généré (fichier `.mvap`)
- Le *log* de l'assemblage par la MVaP (fichier `.ass.log`)
- Le *CBaP* généré (fichier `.mvap.cbap`)
- Le *log* de l'exécution par la MVaP (fichier `.run.log`)
- Le résultat de l'exécution (fichier `.res`)

Au besoin, vous pouvez télécharger ce script et l'adapter à votre machine personnelle.

Pour faciliter l'analyse, on ajoute une règle **start** pour assurer la lecture de l'ensemble de l'entrée:

```
start : calcul  EOF ;
```

Tests : À partir de maintenant, vous pouvez tester votre travail avec les benchmarks (le préfixe du nom d'un fichier correspond au nom de section de ce

TP).

Attention : le script `tp-compil-autocor` est une aide mais il sera exigé que vous soyez capable de tester à la main les différentes étapes.

E - Les opérations d'incrémentations et les blocs

Pour faciliter l'écriture dans la suite, nous souhaitons ajouter le support de l'opérateur `+=` (et donc permettre une affectation du type `x += 1`.

Question : Ajoutez la génération de code pour cet opérateur.

Nous ajoutons dorénavant la notion de bloc au langage de notre calculette programmable. Un bloc est une suite d'instructions qui apparaît entre des crochets `{ ... }` (comme pour le corps d'une fonction). on se donne le code pour le bloc:

```
bloc returns [ String code ]  @init{ $code = new String(); }
    : '{'
          // à compléter
      '}'
      NEWLINE*
    ;
```

Question : Complétez la génération de code pour les blocs (on pourra s'inspirer de ce qui est déjà fait dans la règle **calcul**).

F - Traitement du « while »

Question : Ajoutez la génération de code pour le « while (...) ... ».

Utilisez la méthode newLabel pour générer les noms d'étiquettes.

```
@members {
    private int _cur_label = 1;
    /** générateur de nom d'étiquettes pour les boucles */
    private String newLabel( ) { return "Label"+(_cur_label++); };
    // ...
}
```

Il faudra aussi générer du code pour les opérateurs relationnels, car on ne génère du code que pour 'True' et 'False' :

```
condition returns [String code]
    : 'True'  { $code = "   PUSHI 1\n"; }
    | 'False' { $code = "   PUSHI 0\n"; }
    ;
```

ou alors faites vos premiers tests avec

```
while (True) {i += 1; output(i);}
```

et

```
while (True) i = i + 2;
```

G - Traitement des conditions de base

Question : Ajoutez le traitement des conditions basées sur les opérateurs relationnels (égal `==`, différent `!=`, inférieur `<`, supérieur `>`, inférieur ou égal `<=`, supérieur ou égal `>=`).

Tests : Vous pouvez tester avec les benchs qui vous sont donnés. Vous pouvez aussi en proposer d'autres, si vous pensez que c'est nécessaire.

H - Traitement des expressions logiques

Réfléchissez à la priorité des opérateurs logiques avant de faire la question ci-dessous (on rappelle en particulier que `and` est prioritaire sur `or`).

Question : Ajoutez l'analyse syntaxique des opérateurs logiques `or` (disjonction) , `and` (conjonction) et `not` (négation).

Comme la MVàP n'offre pas d'opcode pour faire du calcul booléen, il faut simuler chaque opération booléenne par une ou plusieurs opérations arithmétiques. Comme indiqué ci-dessus, notre parti pris est de coder faux par 0. Pour vrai, il est possible de choisir 1 ou encore le fait d'être différent de 0.

Question : Ajoutez la génération de code pour ces opérateurs logiques.

Comme indiqué dans le TD7, il est possible de générer du code optimisé dans certains cas.

Étape : Si vous arrivez à ce point avec du retard par rapport au déroulé normal des TPs, nous vous conseillons de faire d'abord la partie K pour bénéficier de soutien pendant les TPs puis de reprendre le reste chez vous.

I - Branchements

Pour l'instant notre calculette n'est pas très pratique puisque nous n'avons pas vraiment de mécanisme de branchement (certes le lecteur observateur nous fera remarquer que les boucles `while` suffisent, comme les exemples de cette section le montrent, mais nous voulons un langage raisonnable). La syntaxe est la suivante.

```
if (condition) then intructionthen
else instructionelse.
```

La seconde ligne avec le else étant optionnelle. On autorisera évidemment les instructions de chaque branche à être des blocs.

Question : Ajoutez le support de ces branchements.

J - Le retour des boucles : le for

Notre langage reste peu pratique sans la célèbre boucle `for` parmi nos instructions, dont nous choisissons la syntaxe ci-dessous.

```
for ( assignation ; condition ; assignation ) instruction
```

Question : Ajoutez le support du `for`.

K - Fonctions

Il faut lire le cours sur les fonctions et faire le TD sur les fonctions pour faire cette partie. Bien préparée sur papier cette étape prend entre 30 minutes et 1 heure à réaliser puisque tout le code est donné. Il s'agit de comprendre l'esprit de la solution proposée et de remplir les trous avec quelques lignes de java permettant de générer du code MVàP. L'expérience montre qu'un étudiant allergique au cours et au papier peut passer de nombreuses semaines à se torturer sans succès sur cette partie. L'équipe enseignante décline donc toute responsabilité dans les cas où cette étape n'est pas prise au sérieux. En cas de tentative infructueuse sur papier, nous sommes ici pour vous aider et nous tenterons de répondre à vos questions.

Étape : Cette section vous guide pas mal. Dans un premier temps, prenez un exemple de code MVàP d'une fonction fabriquée en TD (ou prenez l'exemple du cours) et distribuez chaque ligne de ce code MVàP sous forme de commentaire dans les nouvelles règles de la grammaire ci-dessous, là où vous pensez que le code MVàP doit prendre son origine. Dans un second temps, vous pouvez réfléchir au processus de compilation proprement dit. Imaginez que vous êtes la grammaire g4 et que vous lisez le code de haut niveau qui deviendra le code MVàP. Que faudra-t-il stocker et quand, pour pouvoir gérer tous les petits détails (les adresses des variables etc) ?

K1 - Fonction sans argument

La grammaire suivante permet de reconnaître les déclarations de fonctions sans argument et leurs appels.

```
fonction returns [ String code ]
    : TYPE IDENTIFIANT
        {
            //  Enregistre le type de la fonction
```

```
            }
            '('  ')' bloc
            {
                // corps de la fonction
                $code += "RETURN\n";  //  Return "de sécurité"
            }
        ;
```

On modifie la règle calcul afin de mettre en place la déclaration de fonction:

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

            (fonction { $code += $fonction.code; })*
            NEWLINE*

            { $code += "LABEL Start\n"; }
            (instruction { $code += $instruction.code; })*

            { $code += "  HALT\n"; }
        ;
```

Et on ajoute l'appel comme expression.

```
    expr returns [ String code, String type ]
        :
        //...
        | IDENTIFIANT '('')' // appel de fonction
            {

            }
```

On a dans `TablesSymboles`, les méthodes `addFunction` et `getFunctionType` : la première sert de déclaration (et vérifie qu'il n'y a pas de doublon), la seconde permet de récupérer le type d'une fonction. On utilisera le nom de la fonction comme label.

Notez que comme les table de symboles des fonctions et des variables sont distinctes, on peut créer une fonction de même nom qu'une variable.

> **Question :** Complétez la génération de code pour les appels de fonction sans argument.

K2 - Fonction avec paramètres

Tables locales

Une table des symboles de variables locales à une fonction (dont les paramètres) est seulement utile pendant la définition de cette fonction. Comme nous choisissons de ne pas autoriser des définitions de fonctions imbriquées, une seule table des variables locales au plus est suffisante à tout instant ; dans le cas contraire, il nous faudrait une pile de tables.

Pour activer (et désactiver) ces tables, la classe `TablesSymboles` dispose des fonctions `enterFunction()` et `exitFunction()`.

Question : Complétez la déclaration de fonction pour activer puis désactiver la tables locales (on pourra utiliser `@init` et `@after`).

Déclaration et utilisation des paramètres

Les adresses des paramètres sont comptées négativement à partir de fp. Vision de la pile:

```
                                  fp
                                  \/
          ...  rr  p1  p2  ... pcr  fp(-1)
```

où :

- rr est la place pour la valeur retournée par la fonction (éventuellement)
- p1 est la place du 1er paramètre
- p2 est la place du 2e paramètre
- ...
- pcr est le compteur de programme (pc) au retour (là où il faut se brancher à la sortie de la fonction)
- fp(-1) est la valeur du fp précédent (nécessaire pour restaurer l'environnement)
- fp est la valeur du fp courant (c'est juste là où est stocké fp(-1))

La distance dans la pile séparant la place de la variable locale correspondant au premier paramètre de la place pointée par le fp courant est donc égale : au nombre de paramètres de la fonction plus 2 (à cause de pc et fp qui sont empilés par `CALL`).

L'adresse par rapport à fp d'une variable est toujours négative et se calcule comme : son rang, moins le nombre de paramètres, moins 2 (si on a des int seulement, sinon il faut compter -1 ou -2 selon la taille prise par chaque variable : ceci dépend du type de la variable. La classe `VariableInfo` dispose d'une méthode publique pour trouver cette taille).

Si la table locale a été activée correctement (cf. ci-dessus), la classe `TablesSymboles` dispose d'une fonction permettant de déclarer des paramètres.

```
    tablesSymboles.addParam($IDENTIFIANT.text,"int");
```

On ajoute le code antlr suivant permettant la définition des paramètres

```
    params
        : TYPE IDENTIFIANT
            {
                // code java gérant le premier paramètre
            }
            ( ',' TYPE IDENTIFIANT
                {
                    // code java gérant un paramètre
                }
            )*
        ;

    fonction returns [ String code ]
        // ...
        : TYPE  IDENTIFIANT '('  params ? ')'
        // ...
        ;
```

et le code permettant le passage des arguments

```
// init nécessaire à cause du ? final et donc args peut être vide (mais $args sera non null)
args returns [ String code, int size] @init{ $code = new String(); $size = 0; }
    : ( expr
    {
        // code java pour première expression pour arg
    }
    ( ',' expr
    {
        // code java pour expression suivante pour arg
    }
    )*
      )?
    ;

expr returns [ String code, String type ]
    :
    //...
    | IDENTIFIANT '(' args ')'                  // Appel de fonction
    {
       //...
       // Ajouter ici le nettoyage de la pile
    }
    //...
```

Question : Complétez la déclaration de fonction pour pouvoir déclarer et utiliser des arguments.

Lors de la recherche d'une adresse d'une variable, il faudra d'abord chercher dans la table paramètres, puis, seulement si elle n'est pas trouvée dans la table des variables globales. On a ainsi deux espaces de noms et un paramètre peut masquer l'accès à une variable globale. Cette fonctionnalité est directement réalisée par la classe `getVar`.

Les adresses des paramètres sont par rapport à fp (et négatives) il faut appliquer `PUSHL` et `STOREL` au lieu de `PUSHG` et `STOREG`. Pour cela, on utilisera l'information `VariableInfo.Scope.PARAM` ou `VariableInfo.Scope.GLOBAL` présente dans `vi.scope`.

Question : Modifiez les utilisations de variables pour que celles-ci prennent en compte les paramètres. On portera une attention particulière au *nettoyage* après l'appel de fonction.

K3 - Fonction avec valeur de retour

Pour la valeur de retour, on donne le code ajoutant le support d'un mot-clef `return`

```
instruction returns [ String code ]
        //...
  | RETURN expression finInstruction
        {
        }
        // ...

// lexer
RETURN: 'return'
```

Pour connaître l'adresse de la variable de retour, la classe `TablesSymboles` dispose de la

méthode `VariableInfo getReturn()`.

En plus de remplir la partie donnée, il faudra bien penser, au bon endroit, à réserver la place pour cette valeur et à la libérer.

Question : Complétez la génération de code pour les appels de fonction en intégrant la variable de retour.

Tests : Pour rappel, testez votre travail. Des *benchmarks* sont disponibles ici (le préfixe du nom d'un fichier correspond au nom de section de ce TP).

K4 - Variables locales

On souhaite maintenant ajouter le support de la déclaration des variables locales. Pour cela, on modifie la règle fonction en réutilisant les déclarations déjà écrites.

```
fonction returns [ String code ]
    : TYPE IDENTIFIANT
        {
            //  Enregistre le type de la fonction
        }
        '('  params ? ')'

        '{'

        NEWLINE?

        (decl { $code += $decl.code; })*

        NEWLINE*

         (  instruction  {  // ....  } )*

        '}'

        NEWLINE*

         { $code += "RETURN\n"; }
    ;
```

Grâce au code présent dans `TablesSymbole`, l'appel à `addVarDecl` affecte automatiquement à une variable le *scope* `VariableInfo.Scope.LOCAL` si la déclaration est faite à l'intérieur d'une fonction (en regardant si la table locale existe). Il suffit donc de gérer les variables locales lors de l'utilisation des variables.

Question : Ajoutez le support des variables locales.

L - Support minimal des flottants

Notre langage est pour le moment (sauf si vous y avez déjà travaillé) limité aux entiers. Nous allons maintenant introduire un support minimal (le support plus avancé fait parti des améliorations) pour les flottants. La syntaxe de la déclaration d'une variable est :

```
    double x
```

Question : Ajoutez la déclaration, la lecture et l'écriture de flottants.

Important: la MVàP utilise l'affichage java pour afficher les nombres à virgules. Ceci a pour effet de changer l'affichage selon la langue du système (`4,0` en français, `4.0` en anglais). Pour lancer la machine virtuelle (ou le système de test) dans une autre langue, vous pouvez modifier la variable d'environnement `LANG`. Par exemple : `$ LANG=C.utf8 tp-compil-autocor <vos fichiers> .`

Pour le code MVàP, on dispose de `WRITEF` et de `READF`. Il est important de se rappeler que les flottants prennent 2 mots mémoire. À terme, il est possible de permettre la conversion de type (explicitement ou pas). Dans l'immédiat on se contentera de renvoyer une erreur si une expression combine des types différents en insultant copieusement l'utilisateur sur la sortie d'erreur avec un message approprié.

M - Tester sur quelques programmes classiques

Nous fournissons également des tests un peu plus ambitieux qui nécessitent tous les ingrédients développés au cours de cette série de TP. De nombreux exemples n'ont pas besoin de flottants, et seul de rares exemples nécessitent le cast explicite.

O - Autres améliorations

Les éléments ci-dessus bien réalisés vous permettent d'obtenir une mention bien au DM. Réaliser correctement et intelligemment une ou plusieurs améliorations (selon leur difficulté et la qualité de votre apport) vous permettront d'obtenir une meilleure note, voir la reconnaissance bienveillante de l'équipe enseignante et le titre d'expert en compilation en MVàP. Comme il s'agit de choses nouvelles et non comprises dans les tests que nous vous donnons, il faudra que vous proposiez des benchmarks adaptés.

Ci-dessous quelques suggestions (elles ne sont pas classées par ordre de difficulté, donc regarder les toutes avant de décider la ou lesquelles faire).

Supporter plusieurs types

Nous l'avons déjà évoqué. On peut enrichir la calculette afin de faire du calcul sur les flottants dans les expressions. Les différents *opcodes* relatifs aux flottants sont explicités sur la **MVÀP cheat sheet** sur la page du cours. L'analogue de `PUSHI` pour les flottants est `PUSHF`. Il faut se rappeler que les flottants utilisent 2 mots dans la MVàP donc l'analogue de `POP` est `POP POP`. On peut faire des opérations arithmétiques sur les flottants avec `FADD,FSUB,FMUL,FDIV` et des comparaisons avec `FINF,FINFEQ,FSUP,FSUPEQ, FEQUAL, FNEQ`.

Un autre type simple à mettre en oeuvre est le type booléen. Dans ce cas, il suffit de transformer ce qu'on a appelé jusqu'à présent des conditions en des expressions de type `bool`.

Il est fort utile de pouvoir faire explicitement des conversions explicites d'un type à l'autre. Pour les `bool` et les `int`, c'est indépendant de la représentation dans la MVàP. Pour les flottants qui sont gérés de manière native, la MVàP offre les deux opcodes suivants pour les conversions : `ITOF,FTOI`.

Vous pouvez regarder les benchmarks proposés, mais essentiellement il s'agit de quelque

chose de raisonnable et cohérent pour passer entre int et double via la partie entière. Pour les booléens, comme on vous a laissé partiellement le choix de leur représentation en MVàP par des int, le comportement en écriture d'une variable booléenne initialisée autrement que par `True` et `False` n'est pas normé. On insiste juste pour que le cast d'une expression en tant que booléen se comporte comme `False` ssi l'expression vaut `0` (int) ou `0.` (double).

Casts implicites

Si l'utilisateur mélange plusieurs types dans une expression, mettez un warning sur la sortie d'erreur et faites un cast de manière raisonnable.

Autoriser les tableaux

Support de tableaux. En particulier, leur déclaration, comment on y accède ? et la question intéressante : comment proposez vous de gérer les tableaux comme argument d'une fonction ? Il est probable que `STORER` et `PUSHR` soit des instructions utiles.

Break/continue dans les boucles

Étendre au moins un mécanisme d'itération avec les break/continue, par exemple le for.

Un peu de tolérance : compilation pour les humains

Un bon compilateur essaye de réparer les faiblesses des humains et se montre en général assez tolérant, tout en émettant des mises en garde (warnings en franglais).

- Déclaration de variables si implicites dans le code avec génération de warnings)
- Plus généralement, rendre l'ordre des déclarations de variables et des fonctions plus flexibles.
- Détection d'incohérence, comme par exemple un `return` dans le main
- Tentative de détection de problèmes (pas de return dans toutes les branches d'une fonction, while sans condition, etc)

Optimisation

Support d'un ou plusieurs mécanismes simples d'optimisation évoqués en cours.

- détection et suppression de code mort
- propagation de constantes
- dépliage de boucles constantes
- ...

Déclarations un peu partout

En particulier, déclarer des fonctions localement à une fonction (titre d'experte en MVàP garanti). Indication : utiliser des piles de tables de symboles; ajoutez en 3ème champ dans le bloc d'activation de la pile (en plus de pcr et oldfp) la valeur de pc après avoir lu le label de la fonction ; employez `STORER` et `PUSHR` qui permettent de faire des actions relativement à une adresse écrite sur la pile.