Documentation de conception

Sommaire

I -Structure et Organisations des classes.

Partie B : Tree Toujours dans le répertoire `main/java/fr/ensimag`, se trouve un répertoire `ima/pseudocode`. Dans ce dernier, on retrouve tout un ensemble de classes permettant de programmer l'étape C de chaque langage. En effet, une fois l'ensemble du programme déca vérifié syntaxiquement et contextuellement, il faut désormais le traduire en langage assembleur. Pour cela, on va réutiliser bien évidemment les classes du répertoire *deca* qui sont la base de notre structure, mais les méthodes permettant la traduction se trouve dans *ima/pseudocode*. On retrouve tout d'abord l'ensemble des instructions en assembleur : ADD, BEQ, CMP, HALT... tous les mots clés reconnu par ima permettant de construire un programme assembleur. Ces classes sont très simples, elles permettent simplement en créant un `new ADD(op1, op2)` d'écrire `ADD OP1, OP2` dans le programme .ass. Deux autres classes jouent un rôle fondamental : `GPregister` et `Register`. Celle-ci, comme leurs noms l'indiquent, permet de créer et gérer les registres du tas et de la pile. C'est ici que sont créer les registres `LB GB` et `SP`, et qu'est créée la liste des 16 registres disponibles. Une des classes importantes permettant justement cette écriture dans le programme .ass est `IMAProgram`. Cette classe possède des méthodes telles que `addInstruction` ou `addLine`. Ce sont ces dernières qui gèrent directement l'insertion des lignes assembleurs. Le reste des classes apportent de nouvelles fonctionnalités telles que la gestion des labels et de leurs adresses, mais aussi l'écriture de int, float ou string en assembleur. L'organigramme de ces classes est donné en annexe ... . ### II - Spécificité Lors de la conception de la 'phase C', c'est-à-dire la traduction du langage déca en langage assembleur, plusieurs choix de conception ont dû être réalisés. Un des points importants a été la gestion des registres. Une première idée, nous paraissait la plus simple était de tout gérer grâce à la pile en ne réalisant que des `PUSH` et des `POP`. Cependant, en étudiant la gestion mémoire et les cycles internes de ces instructions, nous en avons convenu qu'une bonne utilisation des registres était requise. Afin de déterminer quel registre n'était pas utilisé à un temps et du code, et ainsi le récupérer afin de s'en servir pour différentes instructions, nous avons mis en place un système simple d'appariement. En effet, nous avons créé une liste de 16 éléments, tous initialisés à 0. Cette liste est un attribut de la classe `Register`. Ainsi, lorsque qu'une nouvelle instruction nécessite l'utilisation d'un registre, on fait appel à `findRegsitre`qui parcourt la liste en question et renvoie le premier registre libre trouvé (la recherche se fait à partir de 3, pour ne pas utiliser les premiers registre au fonctionnement particulier). Cette fonction met alors la case du registre trouvé à 1, considéré comme indisponible. Il ne

reste plus qu'à utiliser la fonction `getR` de `Register` pour accéder au registre souhaité. La fonction `freeRegistre` permet elle de libérer le registre une fois que celui-ci ne sert plus. Le code permettant cette gestion se résume en cela :

```
private static int[] RegisterUsed = new int[getNbreRegistres()];
  public static int findRegistre(){ //throws DecacFatalError{
      int compteur =3;
      boolean trouve = false;
      while (!trouve){
          if(compteur>=getNbreRegistres()){
              //throw new DecacFatalError("Manque de registres pour compiler le programme");
              return 0;
          }
          if (RegisterUsed[compteur]==0){
              RegisterUsed[compteur]=1;
              trouve=true;
          }
          else {compteur ++;}
      }
      return compteur;
  }
  public static void freeRegistre(int i){
      RegisterUsed[i]=0;
  }
```

Comme l'on peut le voir, si aucun registre n'est disponible, alors la fonction ne renvoie pas une erreur comme initialement, mais la valeur zéro. Cette valeur nous indique que l'on ne peut plus procéder comme souhaité, et que nous devons stocker les valeurs dans la pile en utilisant les registres 0 et 2. Cette méthode nous à permis de passer de l'un à l'autre assez facilement, et la plupart de nos fonctions `CodeGen` pour des opérations binaires sont structurées de cette façon :

```
int registre1 = GPRegister.findRegistre();
if(registre1 == 0){
  super.codeGenInstOPPushPop(compiler);
  /// fin des instructions
}else{
  /// fin des instructions
  GPRegister.freeRegistre(registre1);
}
```

La gestion de `PUSH` et des `POP` est entièrement gérée dans la méthode `codeGenInstOPPushPop`. Le code est simple et efficace, plaçant les valeurs des deux opérandes dans les registres R0 et R2. Celui-ci n'est écrit qu'une fois dans `AbstractBinaryExpr` et permet de gérer toutes les classes héritantes.

```
public void codeGenInstOPPushPop(DecacCompiler compiler){
  getRightOperand().codeGenInst(compiler,0);
```

```
  compiler.addInstruction(new PUSH(GPRegister.R0));
  compiler.increaseNeedStack();
  getLeftOperand().codeGenInst(compiler, 0);
  compiler.addInstruction(new POP(GPRegister.getR(2)));
  compiler.decreaseNeedStack();
}
```

On peut observer en plus dans cette méthode la présence de `increaseNeedStack()` et `decreaseNeedStack()` Ces fonctions permettent de connaître précisément la taille de la pile nécessaire afin de compiler le code. En effet, à chaque ajout et retrait d'un élément dans la pile, on utilise ces fonctions. Une fois le code parcouru, on sait, grâce à une sauvegarde de la valeur maximale de l'attribut `needStackCurrent` dans `needStackMax` à chaque ajout, la taille de la pile nécessaire pour compiler le programme. Cela nous a permis de mettre une valeur cohérente à l'instruction `TSTO` et de générer une erreur dans le cas où la pile est trop petite. Ainsi, la commande `Decac -r x` avec `3 < x < 16` dans le cas d'un programme sans objet est 100% efficace et fonctionnel, et alerte le programmeur en cas d'une utilisation de pile trop importante.

Une des spécificités de notre code pourrait être la gestion, pour l'étape C, des opérateurs de comparaison binaire (>, >=, <, <=...). En effet, les méthodes de ces classes sont très restreintes comme on peut le voir sur l'exemple ci-dessous :

```
public class Lower extends AbstractOpIneq {

   public Lower(AbstractExpr leftOperand, AbstractExpr rightOperand) {
       super(leftOperand, rightOperand);
   }

   public void getInstruction1(DecacCompiler compiler, Label inst){
       compiler.addInstruction(new BLT(inst));
   }

   public void getInstruction2(DecacCompiler compiler, Label inst){
       compiler.addInstruction(new BGE(inst));
   }
```

On peut apercevoir qu'aucune méthode permettant de gérer la comparaison entre les deux opérandes n'est présente, mais seulement les deux instructions en lien avec l'opérateur sont accessibles. L'ensemble des instructions sont en fait communes aux autres opérations de comparaison binaire. Dès lors, les fonctions sont écrites dans la classe `AbstractOpCmp`. Cette factorisation permet une meilleure compréhension du code, et une modification plus simple.

III - Algorithme et Structure de données

Afin de gérer la partie *Objet* de notre compilateur Deca, nous avons dû créer un certain nombre de classes. En effet, les classes existantes étaient pour la plupart rattachées à la partie *Sans Object*. En s'appuyant sur les règles de grammaire et le fonctionnement des classes en java, nous avons pu ajouter des classes spécialisées *Object* s'intégrant entièrement à la structure. Par exemple, `ListDeclClass`, `ListParams`, `ListDeclField` et `ListeDeclMetho` étendent de `TreeList<Abstract...>`. Toutes ces classes sont donc basées sur une treeList. Cela permet de parcourir aisément les éléments de la classe. Afin de bien déterminer comment s'agencent les classes entre elles, nous allons rentrer un peu plus précisément dans chacune d'elles. La première classe initialisée est `ListDeclClass`. Comme son nom l'indique, elle permet à chaque ajout de classe dans le programme de stocker celle-ci. Les méthodes de cette classe bouclent sur des `DeclClass`. Celles-ci sont rattachées à la déclaration d'une classe. Une `DeclClass` prend comme attribut son nom, son extension, une liste de ses champs et une liste de ses méthodes. Une liste de champs est définie par la classe `ListDeclField`. Elle permet d'itérer sur les champs d'une classe, définie par `DeclField`. Son fonctionnement est similaire à `DeclVar`, mais il faut également prendre en compte sa visibilité : public ou protected. On arrive ensuite sur la liste de méthodes d'une classe définie par `ListDeclMethod` qui itère sur des `AbstractMethod`. Celles-ci peuvent être de type `Method` simple, ou bien `AssemblyMethod`. Ces deux classes ont en commun le type, l'ident , et la liste des paramètres. Cette dernière est associée à une classe `ListParams` qui elle même itèrent sur des éléments de type `Params`. Un paramètre quant à lui prend simplement un ident et un type. Pour revenir sur la classe `AssemblyMethod`, celle-ci prend également en argument un `StringLiteral`. Celui-ci rentrera directement dans le code assembleur au moment de son appel. Une dernière classe, légèrement isolée des dernières évoquées puisqu'elle étend de `AbstractLValue`, est `MethodCall`. Celle-ci est appelée dès lors que l'on fait appel, soit à une méthode, soit à un champ d'une classe. Cette méthode est probablement celle la plus difficile à gérer puisqu'il faut bien évidemment distinguer l'appel d'un champ à celui d'une méthode, mais également puisque qu'un tel appel peut s'effectuer sur un premier appel, et s'enchaîner ainsi sur plusieurs termes comme ici : `currentClass.getClassDefinition().getMembers().declare()`. Une des fonctions que nous avons implémenté afin de gérer la table de méthodes est `AddMethodList`. En effet, lors de la première passe pour la génération du code assembleur, il est nécessaire d'écrire dans la pile les méthodes de chaque classe. Or, il faut également prendre en compte les méthodes des classes mères, sauf si celles-ci sont redéfinies dans la classe courante. Alors, `AddMethodList` commence par parcourir la classe Object, ajoute le label de la méthode `Equals`, puis parcourt les classes inférieures jusqu'à la classe courante. On obtient alors une pile des lablels des méthodes appelables depuis la classe étudiée. À noter que cette fonction permet également de définir le label de chaque méthode dans la table des méthodes. Toujours dans une obtique de gestion des classes, il est également possible grâce à la fonction `isSubClassOf` de déterminer si une classe

étend d'une autre. Son fonctionnement est assez simple : on parcourt les classes supérieurs dont hérite la classe en question, l'on retourne *true* si l'on tombe sur la classe recherché, *false* si l'on atteint la classe *Object*.

IV - Conclusion.