Maxime Fumel
Robin Moalic

Informatique de Confiance
2ème année

Théorie et concepts des langages de programmation

Présentation de l'interpréteur

Comme vu précédemment, jjTree permet de générer un arbre à partir d'une grammaire javacc. Pour cela, il faut définir des nœuds qui permettent de donner la structure de l'arbre. Chaque symbole terminal doit avoir un nœud qui le décrit (avec sa valeur), ainsi que les non-terminaux pour lesquels il faut définir des actions particulières.

```
void declarationVariable() #variable:
{ }
{
    typage() ";"
}

void typage() #void:
{ Token t; }
{
    t = < TYPE > { jjtThis.value = t.image; }#type
    t = < IDENT > { jjtThis.value = t.image; }#ident
}
```

Nous avons commencé par définir des options dans le fichier .jjt.

```
options
{
  static = true;
  MULTI = true;
  NODE_EXTENDS="MyNode";
}
```

MULTI = TRUE permet de générer une classe pour chaque nœud défini afin de pouvoir définir leur comportement individuel.

NODE_EXTENDS = "MyNode" permet aux classes de nœuds d'étendre la classe MyNode dans laquelle nous avons défini la table des symboles, la table des routines, la pile d'appel et la pile de données ce qui permet à chaque nœud d'y avoir accès.

```
public static HashMap<String, Variable> symbols = new HashMap<>();

public static HashMap<String, SimpleNode> routinesTable = new HashMap<>();

private static ArrayDeque<HashMap<String, Variable>> callStack = new ArrayDeque<>();

private static ArrayDeque<Object> stack = new ArrayDeque<>();
```

Voici les classes générées pour chaque nœud. Leur nom correspond à celui défini dans le fichier .jjt préfixé de "AST".

Dans chacune de ces classes, nous avons défini leur comportement.

Chaque nœud possède sa méthode run() et est chargé d'appeler celle de ses enfants.

A l'exécution du programme, le nœud Start est appelé et on exécute sa méthode run().

Il a 3 enfants qui sont la partie déclaration, la partie instructions et la partie routines.

```
public void run() throws Exception {
    ((MyNode)this.children[0]).run(); // declarations
    ((MyNode)this.children[2]).run(); // routines
    ((MyNode)this.children[1]).run(); // instructions
}
```

Avant de commencer à interpréter les instructions, il faut au préalable peupler les tables de symboles, ce qui correspond à la partie déclarations du programme.

On commence le parcours de l'arbre par la définition des variables. On ajoute chaque variable à la table des symboles.

Puis on fait de même pour la partie routines.

Enfin, on passe à l'exécution des instructions.

Le nœud instructions se contente de lancer la méthode run() de chacun de ses enfants.

```
public void run() throws Exception {
    for (Node n : this.children) {
        ((MyNode)n).run();
    }
}
```

Prenons le nœud affectation par exemple.

Voici sa méthode run().

```
public void run() throws Exception {
    HashMap<String, Variable> sym = get_context_here();

    String varname = (String)((SimpleNode)this.children[0]).value;
    Variable var = sym.get(varname);
    if (var == null)
        throw new ParseException("Use of unknown variable "+varname);

    ((MyNode)this.jjtGetChild(1)).run();

    Object new_value = MyNode.pop();
    var.setVal(new_value);
}
```

```
ret <- 1;
```

Dans le programme exécuté, on souhaite affecter la valeur 1 à la variable ret.

```
instructions
 affect
  ident ret
  value
   arith 1
```

Dans l'arbre généré, on voit que le nœud affect possède 2 enfants qui sont ident et value.

Dans l'exécution du run() de affect, on récupère le nom de l'identifiant (premier enfant) et on exécute le nœud value (deuxième enfant). Cette exécution aura pour effet de mettre une valeur de retour dans la pile. On récupère ensuite la table des symboles, on vérifie que la variable existe, et on dépile pour lui affecter la valeur.

La table des symboles récupérée dépend du contexte de l'exécution, si on est dans le programme principal, c'est la table des symboles qui est utilisée. Si on est dans une procédure le contexte récupéré est celui au sommet de la pile d'appel.

Le type des variable est vérifié lors de l'affectation avec la méthode setVal.

```
public void setVal(String o) throws ParseException {
    if (constant)
        throw new ParseException("Attempted to modify a constant");
    try {
    switch (type) {
      case INT:
          this.val = Integer.parseInt(o);
          break;
      case FLOAT:
          this.val = Float.parseFloat(o);
          break;
      case STRING:
          this.val = o;
          break;
      case BOOL:
          this.val = Boolean.parseBoolean(o);
      }
    } catch(NumberFormatException e) {
        throw new ParseException("incorrect type for variable "+nom+" expecting "+type+" got "+o);
    }
}
```

Voici un exemple de ce qui arrive à l'exécution lorsque on affecte une valeur du mauvais type a une variable.

```
algo exemple
var
  int in;
begin
  print("Entrer un nombre");
  in <- read();
end
```

```
"Entrer un nombre"
Alice
Oops.
grammaireV3.ParseException: incorrect type for variable in expecting INT got Alice
```