Université de Mons

Compilation

Implémentation d'un compilateur du language Dumbo

Auteurs :
Florent Delgrange
Clément Tamines

Professeur:
Véronique Bruyère
Assistant:
Noémie Meunier

23 mai 2016

Contents

1 Introduction

Le but de ce projet est d'implémenter un compilateur pour le language Dumbo. Il est demandé d'adapter la grammaire fournie dans l'énoncé pour qu'elle gère les expressions arithmétiques, les booléens ainsi que les structures de controle **if** et les boucles **for**.

Nous expliquerons dans ce rapport les techniques utilisées ainsi que la grammaire finale que nous avons obtenue.

2 Méthodes utilisées

2.1 Arbre de dérivation

2.2 Stockage des variables

arbre de dérivation

3 Grammaire

Voici la grammaire finale utilisée dans notre projet :
< programme > → < txt >
<programme > → < txt > < programme >
< programme > → < dumbo_bloc >
< programme > → < dumbo_bloc > < programme >
< txt > → [a-zA-Z0-9; & <> ” .\/ \n \p :,]+
< dumbo_bloc > → < expressions_list >
< expressions_list > → < expression > ; < expressions_list >
< expressions_list > → < expression > ;
< expression > → print < string_expression >
< expression > → for < variable > in < string_list > do < expressions_list > endfor
< expression > → for < variable > in < variable > do < expressions_list > endfor
< expression > → if < boolean_expression > do < expression_list > endif
< expression > → < variable > := < string_expression >
< expression > → < variable > := < string_list >
< expression > → < variable > := < operation >
< operation > → < integer >
< operation > → < variable >
< operation > → < operation > + < operation >
< operation > → < operation > - < operation >
< operation > → < operation > * < operation >
< operation > → < operation > / < operation >
< string_expression > → < string >
< string_expression > → < variable >
< string_expression > → < string_expression > . < string_expression >
< string_list > → (< string_list_interior >)
< string_list_interior > → < string >
< string_list_interior > → < string >, < string_list_interior >

< boolean_expression > → < boolean >
< boolean_expression > → < boolean_expression > and < boolean_expression >
< boolean_expression > → < boolean_expression > or < boolean_expression >
< boolean_expression > → < variable > < < variable >
< boolean_expression > → < variable > > < variable >
< boolean_expression > → < variable > = < variable >
< boolean_expression > → < variable > != < variable >
< boolean > → [true|false]
< integer > → [+]
< variable > → [a-z|A-Z]+[a-z|A-Z|0-9_]*
< string > → '[a zA Z0 9; & <> " .\/ \n\p :, =]+'

Nous avons tout d'abord ajoutés les opérations qui permettent de traiter des opérations arithmétiques sur des entiers ou des variables contenant des entiers et stocker le résultat dans une variable. Nous avons aussi ajoutés les expressions booléennes qui permettent le and et le or ainsi que des booléens qui permettent de co comaprer des entiers contenus dans des variables. Ces booléens sont utilisés pour la structure de contreol if et doivent être mis tel qu'ils sont et non dans des variables.

4 Gestion du if et for

Le if a été implémenté via la règle < expression > → if < boolean_expression > do < expression_list > endif
La détection du mot clé if commence la détection de cette règle, nous détectons ensuite une expression booléenne et effections ensuite des expressions. Ceci est géré dans l'arbre par un noeud spécial créé lors de la détection de cette règle. Celui-ci vérifie d'abord si la condition est vérifiée (l'expression correspond d directement à du code python) si c'est le cas, le corps sera exécuté via le noeud correspondant nous récupérons le résultat de son exécution et l'ajoutons. Si ce n'est pas le cas alors nous n'affichons rien. L'utilisation de noeuds gérant les différents aspects permet une gestion intuitive de ces structures. En effet, voici le code correspondant à :

```
class IfNode():
   def __init__(self, *args):
      self.sons = args

   def ex(self):
      condition = self.sons[0]
      action = self.sons[1]

      res = ""
      if(condition.ex()):
         res += str(action.ex())
      return res
```

L'éxécution de ce noeud vérifie si l'éxécution du noeud correspondant à l'expression booléenne est vraie, si c'est le cas il exécute le noeud correspondant à la condition, sinon il retourne une chaine vide (ce qui est équivalent à ne rien retourner).

Le for se fait de la même façon, nous prenons en paramètre une condition

```
class ForNode():
  def __init__(self, *args):
     self.sons = args

  def ex(self):
     var_name = str(self.sons[0]) #Nom de la variable

     initial_value = "" #Valeur initiale de la variable

     #Si la variable éxiste déjà et a une valeur, nous la retenons
     if var_name in variables:
        initial_value = variables[var_name]

     list_or_var = self.sons[1].ex() #Récupérons le contenu de la v

     action = self.sons[2]
     res = ""
     for var in list_or_var:
        variables[var_name] = var
        res+=action.ex()

     #Si la variable avait une valeur initiale, nous la réinstallons
     if initial_value != "":
        variables[var_name] = initial_value

     return res
```

Nous vérifions d'abord si la varable utilisée dans le for est une variable déjà initialisée quelque part. Il est important que cette variable garde sa valeur initiale à la fin de l'éxécution de la boucle. Nous nous occupons donc de retenir la valeur initiale si elle éxiste.

5 Difficultés rencontrées

5.1 Mauvaise détection des lexèmes

5.2 Création et parcours de l'arbre