TP NF11

Génération d'un analyseur lexical et syntaxique pour le langage Logo

Jean Vintache, Adrien Jacquet

7 juin 2016

Table des matières

1 Introduction **3**

2 Grammaire **3**

3 Structures de données **5**
3.1 Classes 5
3.2 Structures du visiteur d'arbre 6

4 Fonctionnement du visiteur d'arbre **7**
4.1 L'instruction repete et la variable loop 7
4.2 Gestion des procédures et des fonctions 7

utc
Université de Technologie
Compiègne

1 Introduction

Le Logo est un langage de programmation à vocation principalement éducationnelle pouvant être utilisé pour réaliser des dessins sur un tableau blanc numérique. L'objectif des différents TP suivis lors de ce semestre était d'implémenter un analyseur lexical et syntaxique permettant d'utiliser une sous-partie des instructions de ce langage.

D'un point de vue technologique, nous avons utilisé le framework libre ANTLR pour générer l'analyseur lexical et la structure principale de l'analyseur syntaxique à partir d'une grammaire.

2 Grammaire

On commence par définir les expressions régulières permettant d'identifier respectivement un nombre entier, un identifiant de fonction ou de variable et les caractères ignorés par l'analyseur lexical :

```
INT : '0' | [1-9][0-9]* ;
ID : [_a-zA-Z][_a-zA-Z0-9]* ;
WS : [ \t\r\n]+ -> skip ;
```

Un programme est constitué d'une liste de déclarations de fonction et/ou de procédure, suivi d'une liste d'instructions, chacun de ces deux élément pouvant être optionnel :

```
programme :
    liste_declarations?
    liste_instructions?
;

liste_declarations :
    (declarationProcedure|declarationFunction)+
;

liste_instructions :
    (instruction)+
;
```

On donne ensuite la structure des déclarations de procédure et de fonction. La différence entre les deux réside dans le fait qu'une fonction doit se terminer par une action retournant une valeur. Contrairement à la syntaxe originale du Logo, nous avons choisi de délimiter la liste des paramètres par des parenthèses pour des raisons qui seront explicitées plus loin dans ce document.

```
declarationProcedure :
    'pour' ID '(' liste_params? ')'
        liste_instructions
    'fin'
;

declarationFunction :
```

```
    'pour' ID '(' liste_params? ')'
        liste_instructions?
        'rends' exp
    'fin'
;

liste_params :
    (':'ID)+
;
```

On défini ensuite la structure des différentes instructions standard ainsi que celle des appels de procédure.

```
instruction :
    ID '(' (exp)* ')' # procedureCall
    | 'av' exp # av
    | 'td' exp # td
    | 'tg' exp # tg
    | 'lc' # lc
    | 'bc' # bc
    | 've' # ve
    | 're' exp # re
    | 'fpos' exp exp # fpos
    | 'fcc' exp # fcc
    | 'repete' exp '['liste_instructions']' # repete
    | 'si' exp '['liste_instructions']' ('['liste_instructions']')? # si
    | 'tantque' exp '['liste_instructions']' # tantque
    | 'donne' '"' ID exp # donne
;
```

On clôture finalement la grammaire en donnant la structure des expressions mathématiques et des appels de fonctions utilisés comme paramètres pour les instructions précédentes.

```
exp :
    ID '(' (exp)* ')' # functionCall
    | 'hasard' exp # hasard
    | exp ('*'|'/') exp # mult
    | exp ('+'|'-') exp # sum
    | exp ('=='|'>='|'<='|'>'|'<'|'!=') exp # test
    | '!' exp # neg
    | atom # arule
;

atom :
    INT # int
    | ':'ID # id
    | 'loop' # loop
    | '('exp')' # parent
;
```

3 Structures de données

3.1 Classes

On commence par définir une classe pour la table des symboles. Elle permet de sauvegarder les valeurs associées aux variables déclarées et de retrouver ces valeurs.

```
public class SymTable {
    private Map<String, Integer> symTable;

    public SymTable() {
        symTable = new HashMap<>();
    }

    public SymTable(SymTable s) {
        symTable = new HashMap<>(s.getSymTableMap());
    }

    public Map<String, Integer> getSymTableMap() {
        return symTable;
    }

    public void donne(String s, int n) {
        symTable.put(s, n);
    }

    public int valueOf(String s) throws RuntimeException {
        Integer res = symTable.get(s);
        if (res == null)
            throw new RuntimeException("variable '"+s+"' non définie");
        else
            return res;
    }
}
```

La classe `Procedure` a pour but de sauvegarder les instructions et les paramètres nécessaires à l'exécution d'une procédure déclarée au début du programme Logo.

```
public class Procedure {
    private List<String> params;
    private ParseTree instructions;

    public Procedure() {}

    public Procedure(Procedure p) {
        this.params = p.params;
        this.instructions = p.instructions;
    }

    public List<String> getParams() {
        return params;
    }

    public void setParams(List<String> params) {
        this.params = params;
```

```
    }

    public ParseTree getInstructions() {
        return instructions;
    }

    public void setInstructions(ParseTree instructions) {
        this.instructions = instructions;
    }
}
```

La classe `Function` hérite de la précédente, mais nécessite en plus de sauvegarder l'expression de retour.

```
public class Function extends Procedure {
    private ExpContext returnExp;

    public Function() {}

    public Function(Procedure p, ExpContext returnExp) {
        super(p);
        this.returnExp = returnExp;
    }

    public ExpContext getReturnExp() {
        return returnExp;
    }

    public void setReturnExp(ExpContext returnExp) {
        this.returnExp = returnExp;
    }
}
```

3.2 Structures du visiteur d'arbre

En utilisant notamment les classes définies précédemment, on créé dans le visiteur d'arbre les structures suivantes :

- `atts` : Une structure associant à certains nœuds de l'arbre une valeur entière. On peut ainsi récupérer la valeur de retour d'un nœud fils après l'avoir visité.
- `loopIndex` : Une pile permettant de stocker la valeur des différents compteur de boucles.
- `procedures` : Un dictionnaire associant à un nom de procédure, l'objet procédure correspondant.
- `functions` : Un dictionnaire similaire pour les fonctions.
- `symTableStack` : Une pile contenant les table de symboles des différents appels de procédures et de fonction.

```
private ParseTreeProperty<Integer> atts = new ParseTreeProperty<>();
private Stack<Integer> loopIndex = new Stack<>();
private Map<String, Procedure> procedures = new HashMap<>();
private Map<String, Function> functions = new HashMap<>();
private Stack<SymTable> symTableStack = new Stack<>();
```

4 Fonctionnement du visiteur d'arbre

4.1 L'instruction repete et la variable loop

En logo, l'instruction `repete` permet d'exécuter un nombre donné de fois une série d'instruction. La variable `loop` permet, elle, de récupérer le numéro d'itération de la boucle `repete` de plus bas niveau.

Techniquement parlant, au début de l'analyse d'une instruction `repete`, on ajoute à la pile `loopIndex` une nouvelle entrée initialisée à 1. À chaque itération, on visite tous les nœuds fils de type instruction et on incrémente la valeur de haut de pile. Si une des instructions visitée contient une expression incluant la variable `loop`, on lui substitue la valeur en haut de `loopIndex`. Une fois l'instruction répète terminée, on dépile le compteur ajouté.

```
public Integer visitRepete(RepeteContext ctx) {
    if (visit(ctx.exp()) != 0) {
        return 1;
    }
    int n = getAttValue(ctx.exp());
    int index = loopIndex.size();
    loopIndex.add(1);
    for (int i = 0; i < n; i++) {
        if (visit(ctx.liste_instructions()) != 0) {
            return 1;
        }
        loopIndex.set(index, loopIndex.get(index) + 1);
    }
    loopIndex.remove(loopIndex.size() - 1);
    Log.appendnl("visitRepete");
    return 0;
}

public Integer visitLoop(LoopContext ctx) {
    if (loopIndex.size() > 0) {
        setAttValue(ctx, loopIndex.get(loopIndex.size() - 1));
        Log.appendnl("visitLoop");
        return 0;
    } else {
        return 1;
    }
}
```

4.2 Gestion des procédures et des fonctions