TP NF11

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

Jean Vintache, Adrien Jacquet

9 juin 2016

Table des matières

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 éléments 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 définit 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 crée 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

4.2.1 Introduction

Comme nous l'avons expliqué dans la section Grammaire, un programme logo peut se décomposer en deux parties distinctes ; la première est les déclarations des fonctions et procédures, la seconde est la liste des instructions d'exécution (certaines pouvant être des appels à des fonctions).

4.2.2 Déclarations de procédures et fonctions

Les méthodes du visiteur d'arbre renvoyant des entiers, il est nécessaire de garder une référence sur la procédure en construction, afin de pouvoir lui affecter paramètres et instructions *pendant* la visite même de sa déclaration. Le visiteur d'arbre possède un attribut `currentProcedure` de type `Procedure`, qui permet cela.

La déclaration et l'utilisation des paramètres d'une procédure requiert l'utilisation de parenthèses. En effet, étant donné que nous pouvons passer en argument le résultat de l'évaluation d'une fonction, il devient difficile de délimiter quels arguments appartiennent à quelle fonction, et le nombre d'arguments d'une fonction. L'utilisation de parenthèses est donc introduite dans notre grammaire, et n'est pas optionnelle.

La visite d'un noeud de déclaration de procédure ou de fonction permet la création d'objets `Procedure` et `Function`, et leur enregistrement dans les `HashMap` du visiteur d'arbre. La clé du couple (clé, valeur) stocké dans ces maps est l'identifiant contenu dans le noeud ID suivant le terme 'pour'.

```
@Override
public Integer visitDeclarationProcedure(DeclarationProcedureContext ctx) {
    String name = ctx.ID().getText();
    ParseTree instr = ctx.liste_instructions();

    currentProcedure = new Procedure();
    currentProcedure.setInstructions(instr);

    visit(ctx.liste_params());

    procedures.put(name, new Procedure(currentProcedure));

    currentProcedure = null; // reset
    Log.appendnl("visitDeclarationProcedure ("+name+")");
    return 0;
}
```

Les déclarations contiennent deux jeux de données clés : les paramètres, et les instructions. Les paramètres sont une liste d'identifiants, stockées dans une liste de chaînes de caractères, que l'on collecte en visitant le noeud `liste_params`. La liste des instructions est un nœud que nous ne visitons pas (cela provoquerait l'exécution des instructions, ce que nous ne voulons qu'à l'appel de la procédure), mais dont nous gardons une référence, dans l'attribut `ParseTree instructions` de la classe `Procedure`.

```
@Override
public Integer visitListe_params(Liste_paramsContext ctx) {
    List<String> params = ctx.ID().stream().map(TerminalNode::getText).collect(
        Collectors.toList());
    currentProcedure.setParams(params);
    return 0;
}
```

La visite d'une déclaration de fonction est identique à la visite d'une déclaration de procédure (une fonction est une procédure), si ce n'est qu'on crée un objet `Function`, que l'on ajoute à la map `functions`.

Également, la valeur de retour d'une fonction est un nœud de type Expression. De la même manière que la liste d'instructions, il n'est pas visité, mais une référence est stockée. Elle sera évaluée à la fin de l'exécution de la fonction.

4.2.3 Appels de procédures et fonctions

L'appel d'une procédure est une `instruction` de la grammaire. Une fonction, elle, retourne une valeur. Son appel est donc reconnu comme étant une expression (`exp`) de la grammaire. Cependant, ces deux objets étant semblables, la visite de l'appel se déroule de la même manière. Le déroulement est donc détaillé pour l'appel d'une `Procedure` (remplacer par l'objet `Function`, et la map `functions` pour obtenir l'équivalent). Notons tout de même une différence pour la fonction, avec la gestion de la valeur de retour, qui sera expliqué en supplément.

Nous nous trouvons dans un état de visite ayant pour contexte `ProcedureCallContext`.

Premièrement, la visite du noeud `ID` nous permet d'accéder au nom de la procédure appelée. La correspondance avec les clés de la map `procedures` du visiteur permet de vérifier l'existence de la procédure que nous voulons exécuter, et, le cas échéant, d'en obtenir une référence.

```
@Override
public Integer visitProcedureCall(ProcedureCallContext ctx) {
    // get the procedure
    String procedureName = ctx.ID().getText();
    Procedure toCall = procedures.get(procedureName);
    if (toCall == null)
        return 1;
```

Deuxièmement, la liste des expressions constituants les arguments passés à la procédure est visitée, et les expressions sont évaluées si besoin. Si le nombre d'arguments passés est différent du nombre de paramètres attendu, la méthode est interrompue et retourne une erreur.

```
    // check number of params
    List<Integer> parameterValues = new ArrayList<>();
    for (ExpContext expCtx : ctx.exp()) {
        if (visit(expCtx) != 0) {
            return 1;
        }
        parameterValues.add(getAttValue(expCtx));
    }
    if (parameterValues.size() != toCall.getParams().size()) {
        Log.appendnl("visitProcedureCall ("+procedureName+") : Error : invalid number
            of argument");
        return 1;
    }
```

Afin de limiter l'accès de la procédure à ses paramètres et variables locales, une nouvelle table de symboles est pushée sur la pile de tables de symboles. Cela implémente le concept de *scope* et d'accessibilité des symboles ; le *scope* global étant la table de symboles se situant au plus profond de la

pile. On utilise la méthode donne (utilisée pour la création de variables) afin d'attribuer les valeurs d'arguments aux noms des paramètres, et on place les variables ainsi créées dans la table de symboles.

```
    SymTable symTable = new SymTable();
    // add the parameters
    for (int i=0; i<parameterValues.size(); i++) {
        symTable.donne(toCall.getParams().get(i), parameterValues.get(i));
    }
    // and push
    symTableStack.push(symTable);
```

On visite le nœud d'instructions de la procédure, ce qui déclenche leur exécution.

```
    // execute procedure
    visit(toCall.getInstructions());
```

Dans le cas d'une fonction, nous devons visiter le nœud Expression stocké dans le paramètre `ExpContext returnExp` afin d'évaluer la valeur de retour. Nous plaçons ensuite cette valeur dans la table `atts` , avec pour clé le contexte de la fonction.

```
    // set the return value
    ExpContext expCtx = toCall.getReturnExp();
    if (visit(expCtx) != 0) {
        return 1;
    }
    setAttValue(ctx, getAttValue(expCtx));
```

Enfin, la table des symboles est retirée de la pile, car nous quittons le bloc d'exécution.

```
    // remove symTable from the stack
    symTableStack.pop();

    Log.appendnl("visitProcedureCall ("+procedureName+")");
    return 0;
}
```