语法分析器

作者：訾源

学号：161250220

邮箱：ziyuan@smail.nju.edu.cn

目标

本次实验的目标是给定没有歧义的、预处理过（消除空产生式）的上下文无关文法，使用LR(1)语法分析方法进行语法分析，并输出结果，类似YACC。

注：测试文件在example目录下。

内容描述

程序采用Java编写。程序读取一个描述上下文无关文法的文本文件和待分析的词法单元序列，来对其进行语法分析，并输出到指定的文件中。本次实验使用LR(1)方法进行分析，输出为分析过程的产生式序列。

思路方法

1. 自定义文法
2. 为了方便起见，消除空产生式，并构造增广文法。
3. 使用LR(1)方法分析文法，并且构造LR(1)分析表。
4. 使用分析表分析语法。

假设

1. 文法不含有二义性
2. 输入不含有任何错误
3. 上下文无关文法以指定的格式描述：

```
开始符号
%%
除了开始符号之外的非终结符1
除了开始符号之外的非终结符2
%%
产生式
```

其中，产生式要求是`A -> b c d`这样，中间使用`->`分割，每个单元之间都有空格。

重要数据结构

转换表

一个非常实实在在的数据结构就是转换表。

```
public class Table {
    private Map<Token, List<Action>> tableColumns;

    public Action getByRowAndCol(int row, Token column) {
        return tableColumns.get(column).get(row);
    }

    public void setByRowAndCol(int row, Token column, Action action) {
        List<Action> actionList = tableColumns.get(column);
        while (row >= actionList.size())
            actionList.add(null);
        actionList.set(row, action);
    }
}
```

通过该转换表，可以获取的对应当前输入和状态的下一步行动。

行动

另一个非常实实在在的数据结构是行动类。

```
public class Action {

    private Integer next;

    private ActionType actionType;
```

```
    public Action(Integer next, ActionType actionType) {
        this.next = next;
        this.actionType = actionType;
    }

    public Integer getNext() {
        return next;
    }

    public void setNext(Integer next) {
        this.next = next;
    }

    public ActionType getActionType() {
        return actionType;
    }

    public void setActionType(ActionType actionType) {
        this.actionType = actionType;
    }

    public static enum ActionType {
        R, // 规约
        J, // 用在goto部分表示跳转
        S, // 入栈
        ACC, // 接受
    }

    @Override
    public String toString() {
        return actionType + "" + next;
    }
}
```

定义了转跳类型和转跳到的行。

产生式

```
public final class Production {

    private final int id;

    private final String name;

    private final List<Token> items;
}
```

产生式由一系列token和名字组成。

Token

用来表示一个语法单元，有可能是终结符，也可能是非终结符，其中有字段标识出它的类别。

```
public final class Token {

    public static enum Type {
        TERMINAL,
        NON_TERMINAL,
        START
    }

    private final String content;

    private final Type type;

    public Token(String content, Type type) {
        this.content = content;
        this.type = type;
    }
}
```

LR项

```
public final class LRItem {

    private final Set<String> probe; // 展望符集合

    private final int pointer; // 指针为k表示在第k个的前面，所以指针后面的元素为items[pointer]

    private final Production production; // 对应的产生式

    public LRItem(Set<String> probe, int pointer, Production production) {
        this.production = production;
        this.probe = probe;
        this.pointer = pointer;
    }
```

`probe`字段表示展望符的集合，`pointer`字段为当前点的位置，`production`訾源为对应的产生式。

核心算法

1. 求First集合

   通过不断迭代所有的符号，直到没有任何新的First元素被加入任何一个非终结符的First集中，才停止。

2. 计算与某状态等价的所有状态

   也是一个不断迭代的过程，对于已有的LR(1)项，查看原点后面的符号是不是非终结符，如果是，那么就将该非终结符的所有产生式加入其中，并且使用First集合来求出展望符集合。不断迭代，直到没有新的LR(1)向加入。

3. 构造LR(1)分析表

   每一步先计算所有的出边，然后计算什么时候规约，一并填入LR(1)分析表中。

4. LR(1)分析表解析输出单词序列

   使用3个栈进行分析，有两个是课堂上讲过的：状态栈和符号栈。还有一个是输入序列栈，一开始把输入单词逆序压入栈，读取时从该栈中读取，在进行规约时候，向该栈中压入元素。

测试

上下文无关文法描述文件：

```
E'
%%
E
T
F
%%
E' -> E
E -> E + T
E -> E - T
E -> T
T -> T * F
T -> T / F
T -> F
F -> ( E )
F -> id
```

输入串：

```
id + ( id - id ) * id / id
```

输出：

```
F -> id
T -> F
E -> T
F -> id
T -> F
E -> T
```

```
F -> id
T -> F
E -> E-T
F -> (E)
T -> F
F -> id
T -> T*F
F -> id
T -> T/F
E -> E+T
E' -> E
```

问题

本次实验比较简单，没有遇到什么问题。

感受与评论

本次实验虽然很简单，但是非常有趣。体会到了语法分析的一般方法，比起简单易实现的LL(1)分析器，能更深入的理解语法分析的流程。