编译原理 Lab2 实验报告

Nosolution

1 目标

使用 LR(1) 分析方法，仿照 Yacc 的生成代码，实现一个简单的计算器，以求对语法分析和 Yacc 更深入地了解。

2 项目描述

本次实验使用硬编码的实现方式，先手工推导出所有运算式的 LR(1) 项集和项集之间的转换关系，构造 LR(1) 预测分析表。再转化为合适的数据结构静态存储在代码中，运行代码对目标文件进行语法分析。

2.1 编译

使用`mvn package`进行编译打包。
如果成功，会在 target 目录下生成 njyacc-1.0.jar 和 njyacc-1.0-jar-with-dependencies.jar 文件

2.2 运行

直接运行: `java -jar njyacc-1.0-jar-with-dependencies.jar <source path>`
如果成功，输出文件为 yy.output，格式：

```
Source: <source content>
Reduce: ...
Reduce: ...
Reduce: ...
...
The calculation result is: ...
```

3 基本思想和方法

基于 LR(1) 和有限状态机的方法，构造 LR(1) 项集后和转换关系后便得到确定性有限状态机，在每一个状态，根据向前看符号进行相应的状态转换，最终便可解析合法的符号串。

4 假设

使用的文法如下：

```
E->E+E
E->E-E
E->E*E
E->E/E
E->E^E
E->-E
E->(E)
E->NUM
```

其中NUM代表数字，数值由 lex 进行解析。
因为中缀表达式具有二义性，因而在文法上我附加了一些限制条件:

- 运算符的优先级：() > -(单目) > ^ > */ > +-
- 结合性：+- 和 */ 都是左结合的，^是右结合的.

本次实现暂时没有错误处理，如果用户的输入存在语法错误，则程序会报错。

5 模型相关描述

我根据教学内容，将语法分析过程进行了建模，模型组件描述如下：

- 输入流 由文法符号组成，能进行顺序访问的流
- 向前看符号 指向输入流中下一个将被读取的文法符号
- LR(1) 分析表 LR(1)DFA 的表表示形式，程序根据此表进行相应的动作
- 状态栈 记录分析过程中的状态转移路径 (规约后状态会被弹出)
- 符号栈 记录所有尚未规约的文法符号

程序执行方式为循环执行，每次根据向前看符号选择对应的动作，并推进向前看符号。

6 重要的数据结构描述

6.1 LR(1) 分析表

按传统方式组织数据，每种状态 (对应一个 LR(1) 项集) 占有一行，对应所有可能输入的文法符号，并且不对字符和 token 进行区分。其中，使用正数代表移进动作，负数代表规约动作，0 便意味着错误，程序会报错。
表存储在一维数组中，按行顺序静态写入。

6.2 分析表查找

程序中yy_next为分析表，查找时使用类似段式存储的思想，yy_next[base:offset]=next state。其中base即为使用当前状态作为索引在yy_base表中的值，offset即为输入文法符号的序号 (已对所有可能的文法符号进行排序处理)。

6.3 产生式信息

yy_prod数组存储所有产生式的相关信息长度为 2*n+2。支持的产生式从序号 1 开始。2*i 为产生式头部在文法符号中的排列序号，可以作为该文法符号的标识符；2*i+1 为产生式体的长度，在规约时供算法使用。

7 核心算法描述

7.1 lex 解析

本实验中的 lex 程序十分简单，只负责判断本次读入的字符是不是数字，然后贪心查找至结尾，存储其值到yylval变量中供语法分析器使用；否则直接返回本次读入字符。遇到空格时会不断跳过，遇到换行或者文件结尾会作为终止符号返回。

7.2 状态转移

状态转移使用状态栈和符号栈共同协作实现。
状态栈的顶部状态为当前状态，当移进时压入新的状态，当规约时根据产生式体的长度弹出相应个数的状态以进行回溯，再压入产生式头部所代表的目标转移状态，等同于分层状态机中，下层状态机完成了全部转移，回到上层状态机进行一次移进。
符号栈与状态栈保持动作协调，移进时压入当前向前看符号，规约时弹出产生式体所需要的符号，压入产生式头部

8 运行用例

8.1 用例 1

输入：

```
3+(1+-2)*7/2^2
```

输出：

```
Reduce: E->NUM
Reduce: E->NUM
Reduce: E->NUM
Reduce: E->-E
Reduce: E->E+E
Reduce: E->(E)
Reduce: E->NUM
Reduce: E->E*E
Reduce: E->NUM
Reduce: E->NUM
Reduce: E->E^E
Reduce: E->E/E
Reduce: E->E+E
```

```
Reduce: E'->E
The calculation result is: 1.250000
```

8.2 用例 2

输入：

```
7.3+(-(3)+2*3.4)/(1+0.5)^2
```

输出：

```
Source: 7.3+(-(3)+2*3.4)/(1+0.5)^2
Reduce: E->NUM
Reduce: E->NUM
Reduce: E->(E)
Reduce: E->-E
Reduce: E->NUM
Reduce: E->NUM
Reduce: E->E*E
Reduce: E->E+E
Reduce: E->(E)
Reduce: E->NUM
Reduce: E->NUM
Reduce: E->E+E
Reduce: E->(E)
Reduce: E->NUM
Reduce: E->E^E
Reduce: E->E/E
Reduce: E->E+E
Reduce: E'->E
The calculation result is: 8.988889
```

9 出现问题及相关解决方案

上面所提到的较为完备的简单计算器总状态为 34 个，涉及总共 10 个文法符号，因此手写 LR(1) 项集需要花费的时间较多。并且因为需要频繁地回看之前写的状态来确定状态间的关系，也要求相当程度的细心和对这些项集的很高的把握程度。最终还是借助计算机等工具把项集和分析表写了出来。

10 个人评价

其一是 LR(1) 分析法确实是一个非常强大的工具，LR(1) 项集封装了分层有限状态机中的层间的转换和推进的逻辑，个人感觉类似于 NFA 转换后的 DFA，每一次状态转换都包含了多条产生式的推进。当构建好分析表后，只要知道下一个文法符号就可以选择正确的路径进行文法分析，即使是靠人也可以很轻松地依靠分析表来分析文法。

其二是通过在二义性语法上提供额外的限制条件来解决冲突，可以很显著地减少 LR(1) 项集的个数，我

尝试过将上述语法拆分成不同的产生式来显式地指定运算符的优先级和结合性，但很快发现项集的数量呈爆炸性增长，实在不适合人工推导。通过隐式指定优先级和结合性 (等于为分析表中冲突项选择唯一的动作)，就可以用不太多的项集完成该二义性语法的分析。

其实本次实验的完成度不高，手工推导再硬编码实现文法分析的感觉并不好。如果能实现较为完整的 yacc，由计算机来完成推导 LR(1) 分析表的工作，才算是更贴近编译原理这门课的教学目标。