编译原理 Lab1 实验报告

Nosolution

一、 目标

通过自己动手实现 Lex 程序，加深理解编译原理课程中学习到的理论知识，主要熟悉和思考正则表达式到 DFA 的转换过程。

二、 项目描述

2.1 项目简介

项目使用 java 语言实现，载体为 maven 项目。在 lex 包下的每一个模块 (单独的文件或者子包) 分别实现一部分功能。
其中 Main 文件为项目的入口和驱动文件，Global 文件定义全局需要的常量，LexSourceParser 负责分析给定的 lex 源规则文件，ReNormalizer 紧接着对读入的正则表达式进行规范化处理，ReTranslator 将规范化后的正则表达式翻译为 NFA，fa 模块提供 FA(包括 NFA 和 DFA) 的定义和交互逻辑，主要实现 NFA 到 DFA 和 DFA 到数据表的转换，最后由 CodeGenerator 生成对应的能分析符合给定规则的字符流的代码。

2.2 编译

进入 pom 文件所在目录，执行打包命令：
`mvn package -Dmaven.test.skip=true`
如果成功，会在在 target 子目录下生成 njlex-1.0.jar 和 njlex-1.0-jar-with-dependencies.jar 文件

2.3 运行

njlex-1.0-jar-with-dependencies.jar 文件已经打包好所需依赖，直接运行：
`java -jar njlex-1.0-jar-with-dependencies.jar lex <lex源规则文件路径>`
在当前目录下生成词法分析器源文件 YyLex.java。

2.4 使用分析器

编译：`javac YyLex.java`
运行：`java YyLex <待分析文件路径>`
分析结果写入名为”yy.out” 的文件

三、 基本思想和方法

实验实现主要采用了结构化编程的方式，那些单独的文件基本在单例的状态下解决问题 (单例或者静态方法)，fa 包中的几个类有一定的交互，主要分为两类，FA 类和 FANode 类。其中 FANode 代表一个有限状态机转换图中的一个节点，以自身的状态作为标志，知晓自己所有的转移条件和转移目标。FA 类则是节点的集合，聚集了所有节点的信息并且有额外的操作，比如 NFA 之间可以完成 concat, or, repeat 的操作 (后见 THOMSON 算法)，DFA 可以通过 NFA 构造自身，或者实现状态的最小化。可以见到类的设计并没有很好地符合面向对象的设计范式，算是时间限制下的一个不完美之处。

四、 假设

假设，也就等同于本实验实现了多少功能

4.1 Lex 源规则文件的编写方式

我所定义的 Lex 源文件规则与标准 Lex 的类似，主要构成：

```
(raws)
definition
%%
(locals)
rules
%%
user code
```

考虑到 java 语言代码的特性，在其上做了一些修改，其中各部分解释如下：

1. raws 部分: 在第一条 definition 之前定义
   - 以空白符 (空格或者制表符) 开头，到本行末尾，此行会被识别为一条`raw definition`
   - 以"%" 开始，以"%" 结束，开始结束符号单独成行，中间所有行会被识别为`raw definition`
   - `raw definition`部分的所有内容会按原样输出到生成文件`YyLex`的类定义之前，诸如`import`或者`package`之类的定义可以在此写入
2. definition 部分：用户编写的正则表达式定义，格式`<name> <definition>`，中间用空格或者制表符分隔
   - `name`只能由字母组成
   - definition 为`name`所标志的正则表达式
   - 在规则部分将会把所有规则中`{<name>}`形式部分替换成其定义
3. locals 部分：类似 raws 部分，在第一条 rule 之前定义，有单行和多行两种形式
   - local 部分的所有内容会被拷贝到`yylex()`方法的头部，用户可以自行声明和初始化局部变量，希望变量名不要以"yy" 开头，以免与程序中预先声明的变量冲突。
4. ruls 部分：用户定义的正则表达式规则，格式`<pattern> <action>`，中间用空格或者制表符分隔

- pattern 为可匹配的正则表达式
- action 为匹配后执行的动作，分为单行和多行
  - 直接写入动作便是单行
  - action 部分为"%" 代表会编写多行，直到某一行除去空白符后为"%" 则停止记录该 pattern 所指定的 action

5. user code 部分：用户自行定义的类成员部分
   - 会被拷贝到`YyLex\verb`类内部，`yylex()\verb`方法之外
   - 可以包括类成员变量和类方法

4.2 Lex 源规则文件

类似标准 Lex 源文件，我也同样提供了一些可用的变量和函数，比如：

- `yytext()`，代表被识别到的字符串
- `yyleng()`，被识别字符串的长度
- `yy_out`，输出流，类型为 PrintStream

等，因不是此次实验的重点因此不多谈

4.3 正则表达式的表示范围

已支持的正则表达式：
普通匹配，`|`运算符，`*`运算符，`+`运算符，`?`运算符，字符类 (中括号表示)，`raw string`(引号表示)
未支持的正则表达式，主要为所有上下文有关的正则表达式符号：
`^`行首符，`$`行末府，`/`后缀符

4.4 正则表达式的表示范围

因为 jvm 在编译时限制单个源文件最大为 64KB，超出会编译失败，而此次实验目前在使用静态声明的数组存储 DFA 的信息，因而 DFA 的大小也就是规则部分的 RE 的数量和匹配能力直接影响了 YyLex.java 文件的大小，所以在编写 Lex 源规则文件的时候需要加入此方面的考虑。

4.5 支持的字符集

可以识别 ASCII 表中的所有可见字符，不可见字符中的`\0`,`\b`和`\v`暂未考虑
其中`\b`被作为程序中的 EPSILON 符号使用，为了简化考虑而没有将其本身加入支持字符集

4.6 用户对换行符的了解

因为生成的分析器代码不会对读入字符有任何的额外处理，因此用户可能在跨平台的换行符识别规则上碰上困难。
Windows 平台的换行符号为`\r\n`，类 Unix 平台上为`\n`，我假设用户如果要识别换行，能准确地写出符合自己使用的平台的规则

五、 FA 相关描述

我觉得代码描述得最清楚，包括其职责和能力

六、 重要的数据结构描述

6.1 Tuple 类

自行实现了元组，构造成列表由NFANode使用，代表从其开始的所有转换条件和转换目标

6.2 Node 类

通过 FA 保存Node,Node保存转换条件和转换目标这种间接使用来避开直接保存<source, condition, destination>
元组这种开销很大的做法。(虽然最后实现出来好像效率也不怎么高)

6.3 FA 类

NFA和DFA的字段如下，除了initial，accepts，nodes，actIdxMap这四个必要的字段，nodeMap作为信息冗余，通过牺牲空间来换取时间

七、 核心算法

7.1 正则表达式后缀化

采用标准的中缀转后缀算法 (在此前需要加入二元的连接符号)，定义运算符优先级，使用栈这种数据结构来辅助完成正则表达式的后缀化。
注意转义属于特殊情况，不能单独作为运算符或者运算数看待，不受后缀化影响，是后缀表达式中的特例，遇上转义符号代表其后的一个符号要转义。

7.2 THOMPSON 算法构造 NFA

RE 后缀化之后，按顺序读取表达式，遇上普通字符则构造一个NFA，表示只有一次转换。
(nfa-construction.png)
再使用栈保存已构造的NFA，遇上运算符推出栈顶的一个或者两个 NFA 进行运算。
(nfa-operate.png)
遍历 RE 后即得所求 NFA

7.3 DFA 状态最小化

构造状态束集合，最初的元素有：所有非终态集合，根据执行动作划分的不同终态集合。
注意因为此 DFA 负责代表所有的规则，因此执行不同动作的终态需要区分开。
不断探测查看是否可以分裂当前状态束集合，直到不能再分裂为止。

八、 用例和运行情况

8.1 用例 1

lex 源规则如下

```
"int"  yy_out.print("INT");
"char"  yy_out.print("CHAR");
{W}+ yy_out.print(" ");
\r?\n yy_out.print("\n");
{L}({L}|{D})* {
yy_out.print("IDENTIFIER: " + yytext());
//indent test
}
```

输入如下：

```
int  char
iden char ccc
  int
```

输出：

```
INT CHAR
IDENTIFIER: iden CHAR IDENTIFIER: ccc
  INT
```

8.2 用例 2

lex 源规则如下

```
D [0-9]
L [a-zA-Z_]
H [a-fA-F0-9]
E ([Ee][+-]?{D}+)
P           ([Pp][+-]?{D}+)
W  [ \t]
FS (f|F|l|L)
%%
"int"  yy_out.print("INT");
"char"  yy_out.print("CHAR");
"public" yy_out.print("PUBLIC");
"void" yy_out.print("VOID");
"return" yy_out.print("RETURN");

"+=" {
yy_out.print("ADD_ASSIGN");
```

```
}
"-=" {
yy_out.print("SUB_ASSIGN");
}
"*=" {
yy_out.print("MUL_ASSIGN");
}
"/=" {
yy_out.print("DIV_ASSIGN");
}
"++" {
yy_out.print("INC_OP");
}
"--" {
yy_out.print("DEC_OP");
}
"->" {
yy_out.print("PTR_OP");
}
"&&" {
yy_out.print("AND_OP");
}
"||" {
yy_out.print("OR_OP");
}
"<=" {
yy_out.print("LE_OP");
}
">=" {
yy_out.print("GE_OP");
}
"==" {
yy_out.print("EQ_OP");
}
"!=" {
yy_out.print("NE_OP");
}
";" {
yy_out.print(";");
}
"{" {
yy_out.print("{");
}
```

```
"}" {
yy_out.print("}");
}
"," {
yy_out.print(",");
}
":" {
yy_out.print(":");
}
"=" {
yy_out.print("=");
}
"(" {
yy_out.print("(");
}
")" {
yy_out.print(")");
}
"[" {
yy_out.print("[");
}
"]" {
yy_out.print("]");
}
"." {
yy_out.print(".");
}
"-" {
yy_out.print("-");
}
"+" {
yy_out.print("+");
}
"*" {
yy_out.print("*");
}
"/" {
yy_out.print("/");
}
"<" {
yy_out.print("<");
}
">" {
```

```
yy_out.print(">");
}
{W}+ yy_out.print(" ");
\r?\n yy_out.print("\n");
{L}({L}|{D})* {
yy_out.print("IDENTIFIER: " + yytext());
//indent test
}
[1-9]{D}*{FS}? {
yy_out.print("DECIMAL_INTEGER: " + yytext());
%%
```

输入如下：

```
public int main() {
int a = 1;
int b = 2:
a += b;
retrun a;
}
```

输出：

```
PUBLIC INT IDENTIFIER: main() {
INT IDENTIFIER: a = DECIMAL_INTEGER: 1;
INT IDENTIFIER: b = DECIMAL_INTEGER: 2:
IDENTIFIER: a ADD_ASSIGN IDENTIFIER: b;
IDENTIFIER: retrun IDENTIFIER: a;
}
```

九、 出现问题及相关解决方案

- 正则表达式处理和 DFA 的优化需要十分关注细节
- FA 中要找 bug 十分困难

解决都是靠花时间

十、 个人感受

至少通过此项目我能更深刻地理解 Lex 中那些算法和 FA 的相关概念，自己编码也会涉及到许多关于具体实现上细节的考虑。
除此之外，代码生成器及分析代码的框架设计也是一个很值得研究部分。可惜时间有限，不能做到进一步的优化和精细化。