基于框架的词法分析器

实验报告

姓名：刘天祺

班级：07121502

学号：1320151097

日期：2018 年 4 月 23 日

目录

一、实验目的和内容

1.1 实验目的

通过实践环节深入理解与编译实现有关的形式语言理论基本概念,掌握编译程序构造的一般原理、基本设计方法和主要实现技术,并通过运用自动机理论解决实际问题,从问题定义、分析、建立数学模型和编码的整个实践活动中逐步提高软件设计开发的能力。

1.2 实验内容

以 C 语言作为源语言，构建 C 语言的**词法分析器**，对于给定的测试程序，输出 XML 格式的属性字符流。词法分析器的构建按照 C 语言的词法规则（**C11** 标准）进行。

二、实现过程和步骤

2.1 实验环境

操作系统：Ubuntu 14.04.5 LTS 64bit 4.4.0-112-generic
Java 版本：1.7.0_171
宿主语言：Python 2.7.6
测试代码：test.pp.c (经过预处理后的源代码)

```
1
2  int main ( ) {
3   int a = 0 ;
4   a = 4 * 5 + 6 - 7 ;
5   int b = 0x12ull ;
6   int c = 0023 ;
7   int d = 045 ;
8   float f = .53f ;
9   float g = 0x.efp10f ;
10  int h = '2\0' ;
11  char p [ ] = "234566432\n" ;
12  return a ;
13  float 4hat_is_wrong = 0x.p3f ;
14  }
```

2.2 实现过程

2.2.1 熟悉框架源

本次实验使用的词法分析器使用 python 语言编写，因此阅读相关源代码：

BITMiniCC.java 是框架的入口程序，程序接受一个 c 语言源代码作为参数，创建 MiniCCompiler 对象并执行编译程序(run)；

MinCCompiler.java 是编译程序，程序调用 readConfig 函数读取配置文件 config.xml 并解析生成对应的接口文件(输入文件，输出文件标准格式和文件名)，runPy 函数是预留给 python 的接口，runPy 函数中根据 config.xml 文件中的配置，执行对应的 python 脚本，python 脚本分析 c 语言源程序，识别单词，输出与源程序等价的属性字流，以 XML 格式文件输出。

2.2.2 处理输入的源程序流（分词）

根据框架的处理流程，词法分析器接受一个经过预处理后的 c 语言源代码。input()函数用于处理**字符串形式**的源程序流，将源程序流中的各个单词提取出来，并以**<单词值，行号>**二元组的数据结构存储为数据流。以 test.pp.c 为例，转化后的数据流如下图所示：

```
$ ./scan.py test.pp.c a.xml
['int', 2] ['main', 2] ['(', 2] [')', 2] ['{', 2]
['int', 3] ['a', 3] ['=', 3] ['0', 3] [';', 3]
['a', 4] ['=', 4] ['4', 4] ['*', 4] ['5', 4]
['+', 4] ['6', 4] ['-', 4] ['7', 4] [';', 4]
['int', 5] ['b', 5] ['=', 5] ['0x12ull', 5] [';', 5]
['int', 6] ['c', 6] ['=', 6] ['0023', 6] [';', 6]
['int', 7] ['d', 7] ['=', 7] ['045', 7] [';', 7]
['float', 8] ['f', 8] ['=', 8] ['.53f', 8] [';', 8]
['float', 9] ['g', 9] ['=', 9] ['0x.efp10f', 9] [';', 9]
['int', 10] ['h', 10] ['=', 10] ["'2\\0'", 10] [';', 10]
['char', 11] ['p', 11] ['[', 11] [']', 11] ['=', 11]
['"234566432\\n"', 11] [';', 11] ['return', 12] ['a', 12] [';', 12]
['}', 13]
```

2.2.3 识别单词类型

① 对 C11 标准中由定义的各类单词，根据其文法定义，画出与之等价的 FA，并转化为**正规式**。

② 对数据流中的每个<单词值，行号>数据项，按顺依次以①中构造的**正规式**对其**单词**进行识别，识别出其所归属的**类型**。对于被识别出的单词，标记其合法性属性为 True，对未被识别出的单词，标记其合法性属性为 False。

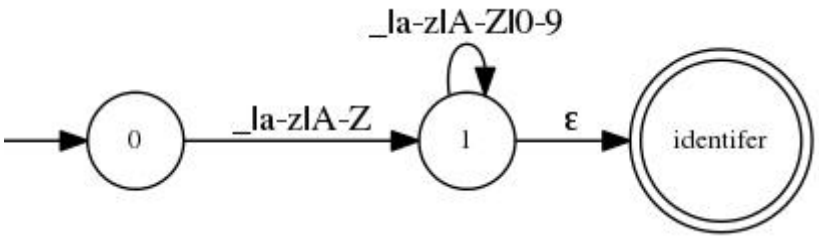
③ 以**(序号，单词值，类型，行号，合法性)**的五元组形式存储属性字流。

2.2.3.1 标识符

文法表示（EBNF）

<标识符> → <非数字字母> | <标识符> <非数字字母> | <标识符> <数字字母>
<非数字字母> → _ | a | ... | z | A | ... | Z
<数字字母> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

FA 状态图



正规式（python re 语法）

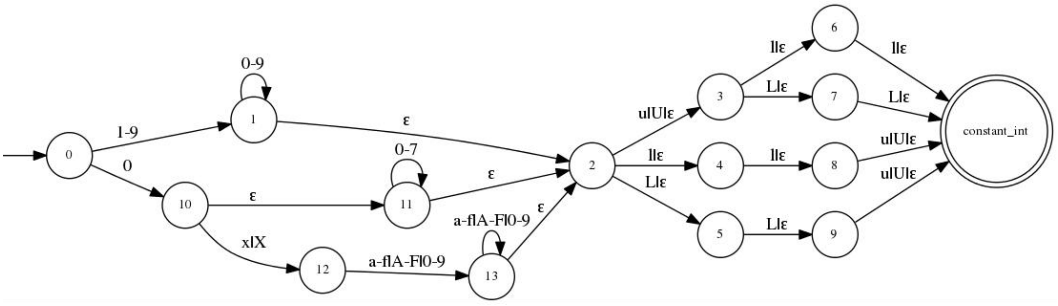
```
R = [_a-zA-Z][_a-zA-Z0-9]*
```

2.2.3.2 整型常量

文法表示（EBNF）

```
<整型常量> → ( <十进制整型常量> | <八进制整型常量> |
                    <十六进制整型常量> ) [ <整型后缀> ]
<十进制整型常量> → <非零数字字母> | <十进制整型常量> <数字字母>
<八进制整型常量> → 0 | <八进制整型常量> <八进制数字字母>
<十六进制整型常量> → <十六进制前缀> <十六进制数字字母> |
                          <十六进制整型常量> <十六进制数字字母>
<十六进制前缀> → 0x | 0X
<数字字母> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<非零数字字母> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<八进制数字字母> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<十六进制数字字母> → 0 | ... | 9 | a | ... | f | A | ... | F
<整型后缀> → <无符号型后缀> ( [ <长型后缀> ] | <长长型后缀> ) |
               ( <长型后缀> | <长长型后缀> ) [ <无符号型后缀> ]
<无符号型后缀> → u | U
<长长型后缀> → l | L
<长长型后缀> → ll | LL
```

FA 状态图



正规式（python re 语法）

```
R = ([1-9][0-9]*|0[0-7]*|0[x|X][a-fA-F0-9]+)([u|U]?[l|L]{0,2}[u|U]?)
Rd = lL|Ll|[u|U].*[u|U]
```

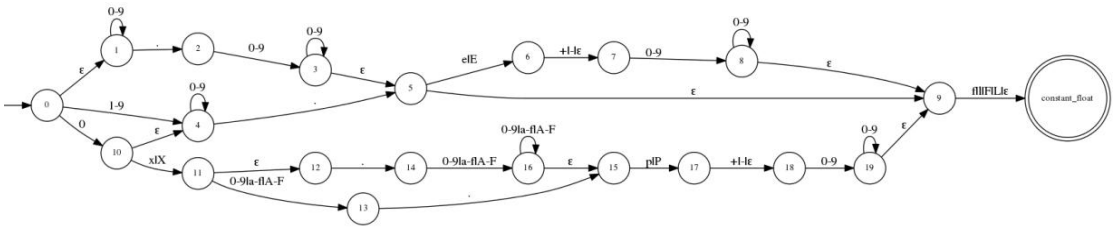
注：R 所识别的字符串集合包含冗余的字符串，通过 R_d 去除多匹配的字符串，使其与文法等价，即 $L(R)-L(R_d) = L(G)$

2.2.3.3 浮点型常量

文法表示（EBNF）

```
<浮点型常量> → <十进制浮点型常量> | <十六进制浮点型常量>
<十进制浮点型常量> → ( <小数常量> [ <指数> ] | <数字串> <指数> )
                        [ <浮点型后缀> ]
<十六进制浮点型常量> → <十六进制前缀> ( <十六进制小数常量> |
                          <十六进制数字串> ) [ <浮点型后缀> ]
<十六进制前缀> → 0x | 0X
<小数常量> → [ <数字串> ] . <数字串> | <数字串> .
<指数> → ( e | E ) [ <符号> ] <数字串>
<符号> → + | -
<数字串> → <数字字母> | <数字串> <数字字母>
<数字字母> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<十六进制小数常量> → [ <十六进制数字串> ] . <十六进制数字串> |
                        <十六进制数字串> .
<二进制指数> → ( p | P ) [ <符号> ] <数字串>
<十六进制数字串> → <十六进制数字字母> |
                    <十六进制数字串> <十六进制数字字母>
<十六进制数字字母> → 0 | … | 9 | a | … | f | A | … | F
<浮点型后缀> → f | l | F | L
```

FA 状态图



正规式（python re 语法）

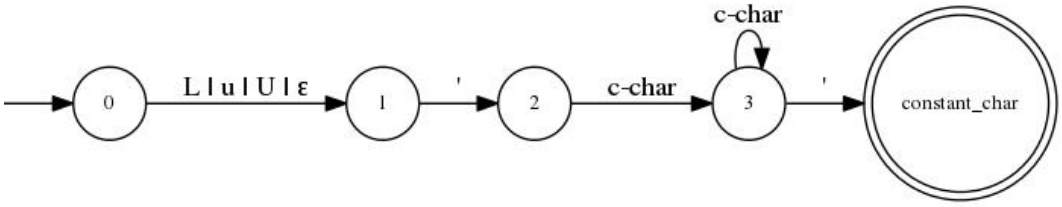
```
R = ((([0-9]*\.[0-9]+ | [0-9]+\.)[e|E][+|-]?[0-9]+) | ([0-9]*\.[0-9]+ | [0-9]+\.) |
        ([0-9]+[e|E][+|-]?[0-9]+) | (0[x|X])(([0-9a-fA-F]*\.[0-9a-fA-F]+) |
          ([0-9a-fA-F]+\.) | ([0-9a-fA-F]+) |)([p|P][+|-]?[0-9]+)) [f|l|F|L]?
```

2.2.3.4 字符常量

文法表示（EBNF）

```
<字符常量> → [ L | u | U ] ' <c-字符串> '
<c-字符串> → <c-字符> | <c-字符串> <c-字符>
<c-字符> → <任意字符> | <转义字符>
<任意字符> → … ( ' 和 \ 除外)
<转义字符> → \ ( ?| ' | " | ? | \ | a | b | f | n | r | t | v | … )
```

FA 状态图



正规式（python re 语法）

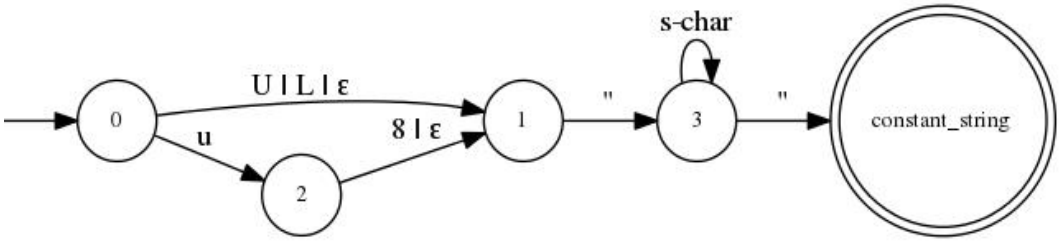
```
R = [L|u|U]? \' [^\\\']+ \'
```

2.2.3.5 字符串常量

文法表示（EBNF）

```
<字符串常量> → [ <字符串前缀> ] " [ <s-字符串> ] "
<字符串前缀> → u8 | u | U | L
<s-字符串> → <s-字符> | <s-字符串> <s-字符>
<s-字符> → <任意字符> | <转义字符>
<任意字符> → ... ( " 和 \ 除外)
```

FA 状态图



正规式（python re 语法）

```
R = [L|u|U]? \" [^\\\"]* \" | u8 \" [^\\\"]* \"
```

2.2.3.6 运算符与界限符

数量有限，可枚举，无需构造自动机识别

2.2.3.7 关键字

数量有限，可枚举，无需构造自动机识别

2.3.4 输出 xml 文件

将属性字流转存为 xml 格式。

具体步骤为：对属性字流创建 tokens 标签，对属性字流中的每个属性字创建 token 标签，对属性字中的序号、单词值、类型、行号、合法性，分别创建 number, value, type, line, valid 标签，并将相应的值填充进标签。最后将 xml 格式的属性字流以字符串形式输出到文件。

三、运行效果截图

识别出的关键字、标识符与界限符如下图所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="test.pp.c">
  <tokens>
    <token>
      <number>1</number>
      <value>int</value>
      <type>keyword</type>
      <line>2</line>
      <valid>True</valid>
    </token>
    <token>
      <number>2</number>
      <value>main</value>
      <type>identifer</type>
      <line>2</line>
      <valid>True</valid>
    </token>
    <token>
      <number>3</number>
      <value>(</value>
      <type>sep/operator</type>
      <line>2</line>
      <valid>True</valid>
    </token>
```

识别出的整型、浮点型常量如下图所示：

```
    <token>
      <number>24</number>
      <value>0x12ull</value>
      <type>constant_int</type>
      <line>5</line>
      <valid>True</valid>
    </token>
```

```
    <token>
      <number>44</number>
      <value>0x.efp10f</value>
      <type>constant_float</type>
      <line>9</line>
      <valid>True</valid>
    </token>
```

识别出的字符型、字符串型常量如下图所示：

```
    <token>
      <number>49</number>
      <value>'2\0'</value>
      <type>constant_char</type>
      <line>10</line>
      <valid>True</valid>
    </token>
```

```
    <token>
      <number>56</number>
      <value>"234566432\n"</value>
      <type>constant_string</type>
      <line>11</line>
      <valid>True</valid>
    </token>
```

对于非法字符，gcc 给出的错误信息如下图所示：

```
$ gcc test.pp.c
test.pp.c: In function ‘main’:
test.pp.c:10:10: warning: multi-character character constant [-Wmultichar]
  int h = '2\0' ;
          ^
test.pp.c:13:8: error: invalid suffix "hat_is_wrong" on integer constant
  float 4hat_is_wrong = 0x.p3f ;
        ^
test.pp.c:13:8: error: expected identifier or ‘(’ before numeric constant
test.pp.c:13:24: error: no digits in hexadecimal floating constant
  float 4hat_is_wrong = 0x.p3f ;
                        ^
```

本实验中词法分析器给出的错误信息如下图所示：

```
    <token>
      <number>62</number>
      <value>4hat_is_wrong</value>
      <type>unknown</type>
      <line>13</line>
      <valid>False</valid>
    </token>
    <token>
      <number>63</number>
      <value>=</value>
      <type>sep/operator</type>
      <line>13</line>
      <valid>True</valid>
    </token>
    <token>
      <number>64</number>
      <value>0x.p3f</value>
      <type>unknown</type>
      <line>13</line>
      <valid>False</valid>
    </token>
```

四、实验心得体会

通过实验，复习了课堂上学到的自动机理论。考虑到词法分析器的本质——**识别单词**，以及 python 语言**正规式库**(re)的强大功能，我没有用课本上“语言词法规则->状态转化图->可行的扫描器”的模型，而是采用“语言词法规则->**有限自动机**->**正规式**->可行的扫描器”的方式来构造扫描器。

本次实验中采取的方法省去了建立主表和分表的时间，而且如果以后文法规则有了增改，若使用程序中心法或是数据中心法，则需要重新构造状态图、重新命名状态、重新建立二级表，**十分麻烦**，而若采用正规式法实现，则只需要修改对应的正规式，可见采用正规式法实现增加了词法分析器的**灵活性**。

此外，本次实验中采用**先分词再识别**的方法，省去了超前搜索和回退字符等操作。