实验 2 词法分析实验

1120141952 宋傲

1 实验目的和内容

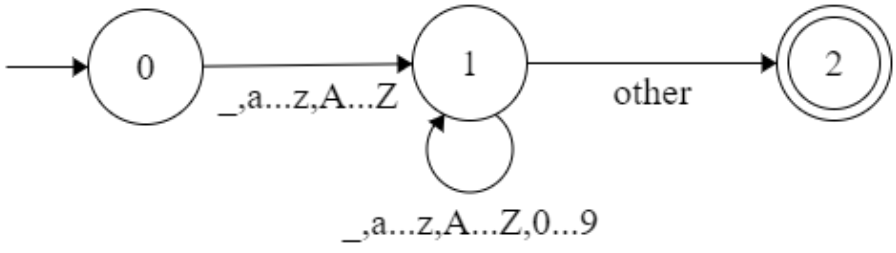
本次实验基于 bit-minicc 框架，对 C 语言文法子集的词法进行了分析，并将分析结果按格式输出成 XML 属性字流。词法分析器使用 Java 语言进行实现，DFA 的实现方式为程序中心法。

在构造 DFA 的过程中，我们可以进一步熟练 DFA 的构造以及化简的步骤。在实现词法分析的过程中，我们能够熟悉 DFA 在程序中的实现方式。在输出成 XML 的过程中，我们能够熟悉 XML 解析和编辑的基本方法。

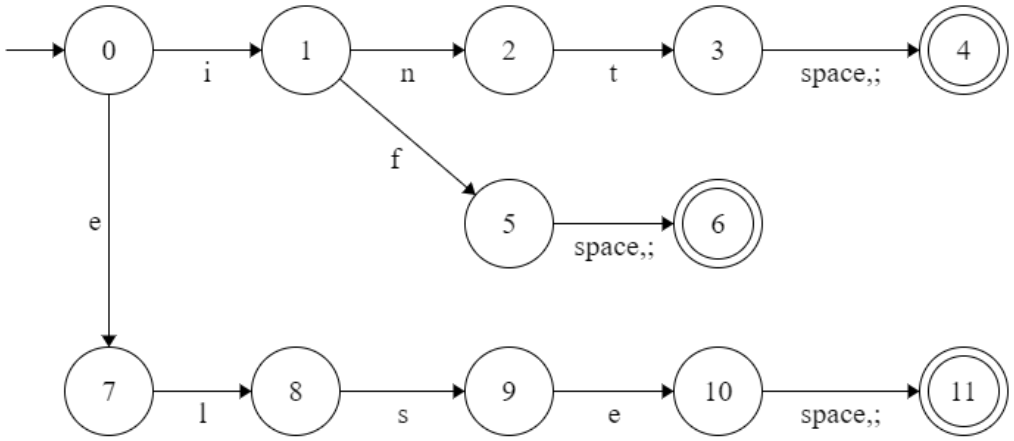
2 实现的具体过程和步骤

2.1 DFA 的构造和化简

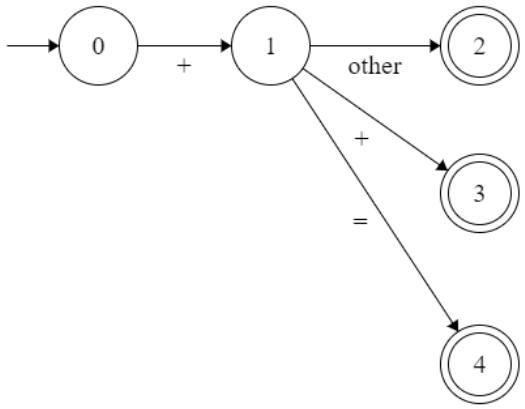
- **标识符：**按照文档要求，以下划线和字母开头。其他位置可以为字母，下划线或数字。Other 代表读入了其他字符，此自动机停止读取并开始回退字符。



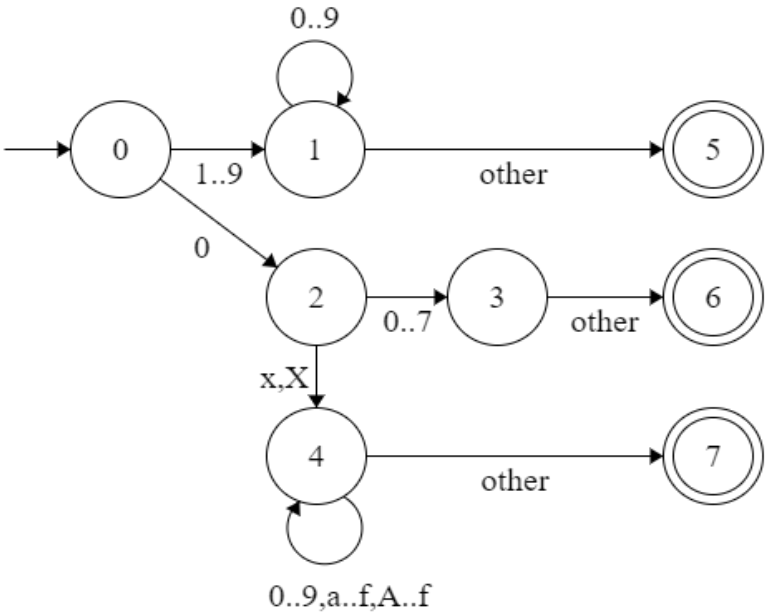
- **关键字：**图中给出了关键字 if,else,int 的状态转换图，其他关键字实现方法类似，此处略去。当识别到空格或分号时，回退并退出此自动机。



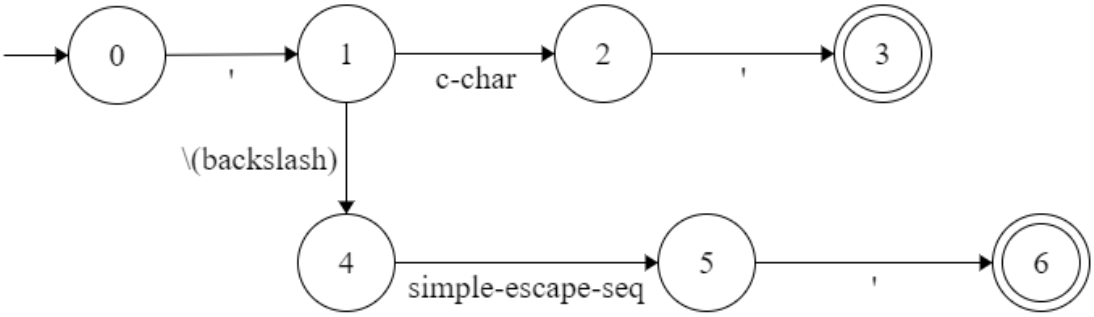
- **运算符：**此处给出了+，++，+=的状态转换图，其他运算符及分隔符实现方法类似。



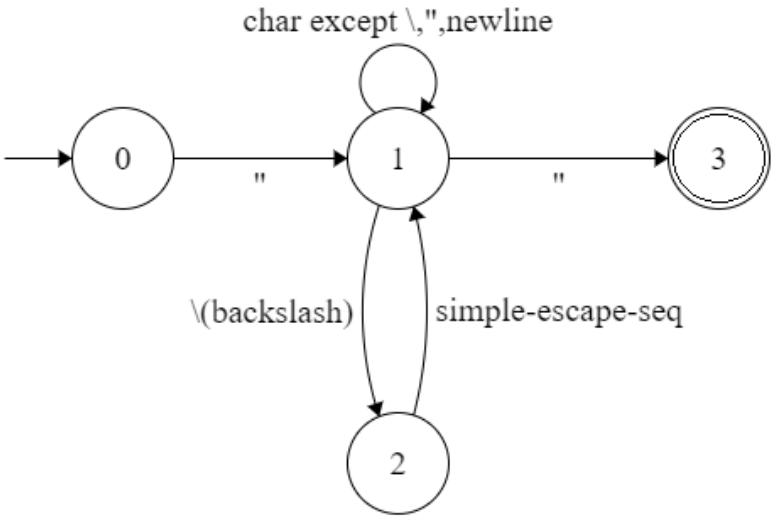
- **整型常量：**支持格式为十进制，八进制和十六进制三种类型的整型常量的状态转换图如下，通过识别八进制的前导 0 和十六进制的前导 0x(或 0X)来判断整型常量的进制。此处十六进制支持小写 a-f 和大写 A-F。当读入非数字或 a-f(A-F)的其他字符时，退出此自动机并进行字符回退。



- **字符型常量：**当识别到转义标识\后，判断下一个读入的字符是否为文档中规定的转义序列，包括 :\’ \” \? \\ \a \b \f \n \r \t \v 此处由于能够明确识别到字符型常量的结尾，因此不进行字符回退。



- **字符串字面量：**识别转义标识后立即读入下一个字符，若为合法的转义序列，则将刚刚读入的两个字符当做一个来进行处理。



2.2 框架的配置和原有代码的替换

在框架中我使用的语言是 Java，因此直接在源码中实现 IMiniCCScanner 接口的 run 方法，并在 MiniCCCompiler 中将第二步的 scanner 替换成 MyScannerImplements 即可运行自己编写的词法分析器。

```
public class MyScannerImplements implements  IMiniCCScanner {

    private BufferedReader reader;
    private BufferedWriter writer;

    private Document outputXML;

    private int lineNum = 0;
    private int wordNum = 0;
```

```java
// step 2: scan
String scOutFile = ppOutFile.replace(MiniCCCfg.MINICC_PP_OUTPUT_EXT, MiniCCCfg.MINICC_SCANNER_OUTPUT_EXT);

if(scanning.skip.equals("false")){
    if(scanning.type.equals("java")){
        if(scanning.path != ""){
            Class<?> c = Class.forName(scanning.path);
            Method method = c.getMethod("run", String.class, String.class);
            method.invoke(c.newInstance(), ppOutFile, scOutFile);
        }else{
            MyScannerImplements myScanner = new MyScannerImplements();
            myScanner.run(cFile, scOutFile);
        }
    }else if(pp.type.equals("python")){
        this.runPy(ppOutFile, scOutFile, scanning.path);
    }else{
        this.run(ppOutFile, scOutFile, scanning.path);
    }
}
```

配置文件 config.xml 不必进行修改，将 scanner 之后的步骤的 skip 属性改为 true，待后续实现相应步骤后再改为 false。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config name="config.xml">
  <phases>
    <phase>
      <phase skip="false" type="java" path="" name="pp" />
      <phase skip="false" type="java" path="" name="scanning" />
      <phase skip="true" type="java" path="" name="parsing" />
      <phase skip="true" type="java" path="" name="semantic" />
      <phase skip="true" type="java" path="" name="icgen" />
      <phase skip="true" type="java" path="" name="optimizing" />
      <phase skip="true" type="java" path="" name="codegen" />
      <phase skip="true" type="java" path="" name="simulating" />
    </phase>
  </phases>
</config>
```

2.3 词法分析的实现

首先，我们需要按照程序中心法的要求在程序中模拟 2.1 节所述的所有 DFA 模型，每一个 DFA 模型即对应一类语法特性。将所有语法特性综合起来则构成了整个语法分析器的核心。

在程序中心法的实现过程中，我们使用了 if 以及 switch 代表了 DFA 中

从每一个状态中进行转换的各个分支，即状态装换图上对应的每个箭弧。

例如，我们以识别+，++，+=这三个运算符的 DFA 为例，代码如下图：

```
else if(nextLine.substring(endIndex,endIndex+1).matches("\\+")) {
    endIndex++;
    if(nextLine.substring(endIndex,endIndex+1).matches("\\+|=")) {
        endIndex++;

    }
    Token t = new Token(wordNum++,nextLine.substring(beginIndex,endIndex).trim(),TokenType.TOKEN_TYPE_OPERATOR,lineNum,true);
    t.writeXML(outputXML.getRootElement());
}
//- -- -=
```

从代码中我们可以看到，我们通过 java 的正则表达式引擎来匹配下一个读入的字符，正则表达式中写入我们对于下一个字符的条件限制。而调用 subString(endIndex,endIndex+1)则代表了从当前 endIndex 指向的下标处取一个字符。

```
//append # to the end of each line to avoid out of bound exception.
nextLine = nextLine.concat("#");
```

此处需要注意一个下标越界的问题，代码如上图所示。如果当前 endIndex 指向的是读入的这一行代码的最后一个字符，则会抛出下标越界异常。为了解决这个问题，我们采取了在读入的这行代码最后添加一个#，由于预处理后代码中除了字符和字符串内不应包含#,因此，若识别到这些#，则可以判断这一行的结束，而不会造成下标越界的问题。

在程序中为了避免复杂性，没有模拟对于关键字的 DFA。为了解决识别关键字的问题，我的解决方式是将关键字先和标识符一起识别，然后对这个串在关键字集合中进行搜索，若这个串存在于关键字集合，则按一个关键字进行处理，否则按普通的标识符进行处理。代码如下图：

```java
public class KeywordVariables {
    private static ArrayList<String> keyword = new ArrayList<String>();
    static {
        keyword.add("int");
        keyword.add("char");
        keyword.add("const");
        keyword.add("struct");
        keyword.add("if");
        keyword.add("else");
        keyword.add("while");
        keyword.add("continue");
        keyword.add("break");
        keyword.add("goto");
        keyword.add("do");
        keyword.add("return");
    }

    public static boolean isKeyword(String word) { return keyword.contains(word); }
}
```

2.4 XML 的写入和属性字流的生成

我们定义令牌 Token 类型如下：

```java
public class Token {
    private int number;
    private String value;
    private TokenType type;
    private int line;
    private boolean valid;

    public Token(int number, String value, TokenType type, int line, boolean valid) {
        this.number = number;
        this.value = value;
        this.type = type;
        this.line = line;
        this.valid = valid;
    }

    public void writeXML(Element root)
    {
        Element token = root.addElement("token");
        token.addElement("number").addText(number+"");
        token.addElement("value").addText(value);
        token.addElement("type").addText(type.toString());
        token.addElement("line").addText(line+"");
        token.addElement("valid").addText(valid+"");

    }
}
```

我们使用 dom4j 来进行 XML 的处理。Token 类中有 writeXML 方法，则代表向 Element 中写入一个属性字。当然，此处的 Element 为文档中要求的<project>节点。效果如下图：

```xml
<token>
    <number>0</number>
    <value>int</value>
    <type>keyword</type>
    <line>1</line>
    <valid>true</valid>
</token>
```

为了标识 Token 的类型，我们定义了一个枚举类型 TokenType，代码如下：

```java
enum TokenType {

    TOKEN_TYPE_KEYWORD("keyword"),
    TOKEN_TYPE_INDENTIFIER("identifier"),
    TOKEN_TYPE_SEPRATOR("seprator"),
    TOKEN_TYPE_CONSTINT("const_i"),
    TOKEN_TYPE_CONSTCHAR("const_c"),
    TOKEN_TYPE_STRING("stringLiteral"),
    TOKEN_TYPE_OPERATOR("operator");

    private String typeName;

    private TokenType(String name) { typeName = name; }


    @Override
    public String toString() { return typeName; }
}
```

上述工作完成后，我们就可以在 DFA 识别成功后创建 Token 对象并调用对象的 writeXML 方法进行写入了，代码如下：

```java
//indentifier
if(nextLine.substring(endIndex,endIndex+1).matches("[a-z]|[A-Z]|_")) {
    endIndex++;
    while(nextLine.substring(endIndex,endIndex+1).matches("[a-z]|[A-Z]|_|[0-9]")) {
        endIndex++;
    }
    String dst = nextLine.substring(beginIndex,endIndex).trim();
    if(KeywordVariables.isKeyword(dst)) {
        Token t = new Token(wordNum++,nextLine.substring(beginIndex,endIndex).trim(),TokenType.TOKEN_TYPE_KEYWORD,lineNum,true);
        t.writeXML(outputXML.getRootElement());
    }
    else {
        Token t = new Token(wordNum++,nextLine.substring(beginIndex,endIndex).trim(),TokenType.TOKEN_TYPE_INDENTIFIER,lineNum,true)
        t.writeXML(outputXML.getRootElement());
    }

}
```

3　运行效果截图

待分析的源代码为(test2.c)：

```
int a = 1;
int fun(int a,int b)
{
    return a+b;
}
1>>2
2<=3
1+2;
+ ++ +=
- -- -=
* *=
& && | || ~ !
{}
()
[]
01234 0xddd
"\nhello world\t"
122222
_variable 0x33ff
_abcde000
123
0x2233ff
```

产生的属性字流如下（部分）：

```
<token>
  <number>52</number>
  <value>0xddd</value>
  <type>const_i</type>
  <line>16</line>
  <valid>true</valid>
</token>
<token>
  <number>53</number>
  <value>"\nhello world\t"</value>
  <type>stringLiteral</type>
  <line>17</line>
  <valid>true</valid>
</token>
<token>
  <number>54</number>
  <value>122222</value>
  <type>const_i</type>
  <line>18</line>
  <valid>true</valid>
</token>
<token>
  <number>55</number>
  <value>_variable</value>
  <type>identifier</type>
  <line>19</line>
  <valid>true</valid>
</token>
```

可以看到程序正确识别了源代码中的所有属性字，并将他们按照正确的格式写入了 test2.token.xml.

4　实验心得体会

这次实验难度较上一次明显提高了一个等级。对于词法分析的过程，最初我也只是能停留在把 DFA 画在纸上的程度。这次实验给了我们一个将书本上的知识变成能力的机会。虽然 DFA 的实现在程序设计上并没有什么太大的难度，但是属性字的种类非常多，每一个都能正确实现并输出也是一件不容易的事情。最终完成的一刻也获得了很大的成就感。

希望后续的实验也能顺利完成，最终形成一个完整的编译程序。