

编译原理 实验三 词法分析实验

1120180488 王梓丞

实验目的

熟悉 C 语言的词法规则, 了解编译器词法分析器的主要功能和实现技术, 掌握典型词法分析器构造方法, 设计并实现 C 语言词法分析器

实验内容

根据 C 语言的词法规则, 设计识别 C 语言所有单词类的词法分析器的确定有限状态机, 并编程实现。词法分析器的输入为 C 语言源程序, 输出为属性字流。

实验环境

操作系统: Ubuntu 20.04 LTS
CPU: Intel i7-10875H (8) @ 2.304GHz
CPU 核数: 8
内存大小: 7929MiB
Java 版本: openjdk 16 2021-03-16

具体过程

我用的方法是手动构造自动机识别 C 语言的单词。在开始前, 需要先通过阅读 ISO/IEC 9899:201x 的 C 语言规范确定需要识别的单词的构造规则, 接着进行自动机的构造, 最后再编程实现。

状态机构造

通过阅读 C 语言语法规范，我确定了主要需要识别的 C 语言单词的种类，分别为：

- 关键字
- 运算符
- 常量
 - 整形常量
 - 浮点常量
 - 字符常量
 - 字符串常量
- 变量标识符

由于其中每一种单词的识别规则都较为复杂，因此我选择将其分模块构造自动机。最终将其分为了如下模块：

- 关键字识别自动机
- 运算符识别自动机
- 数字常量识别自动机
- 字符/字符串常量识别自动机
- 变量识别自动机

以数字常量识别的有限自动机构造为例。

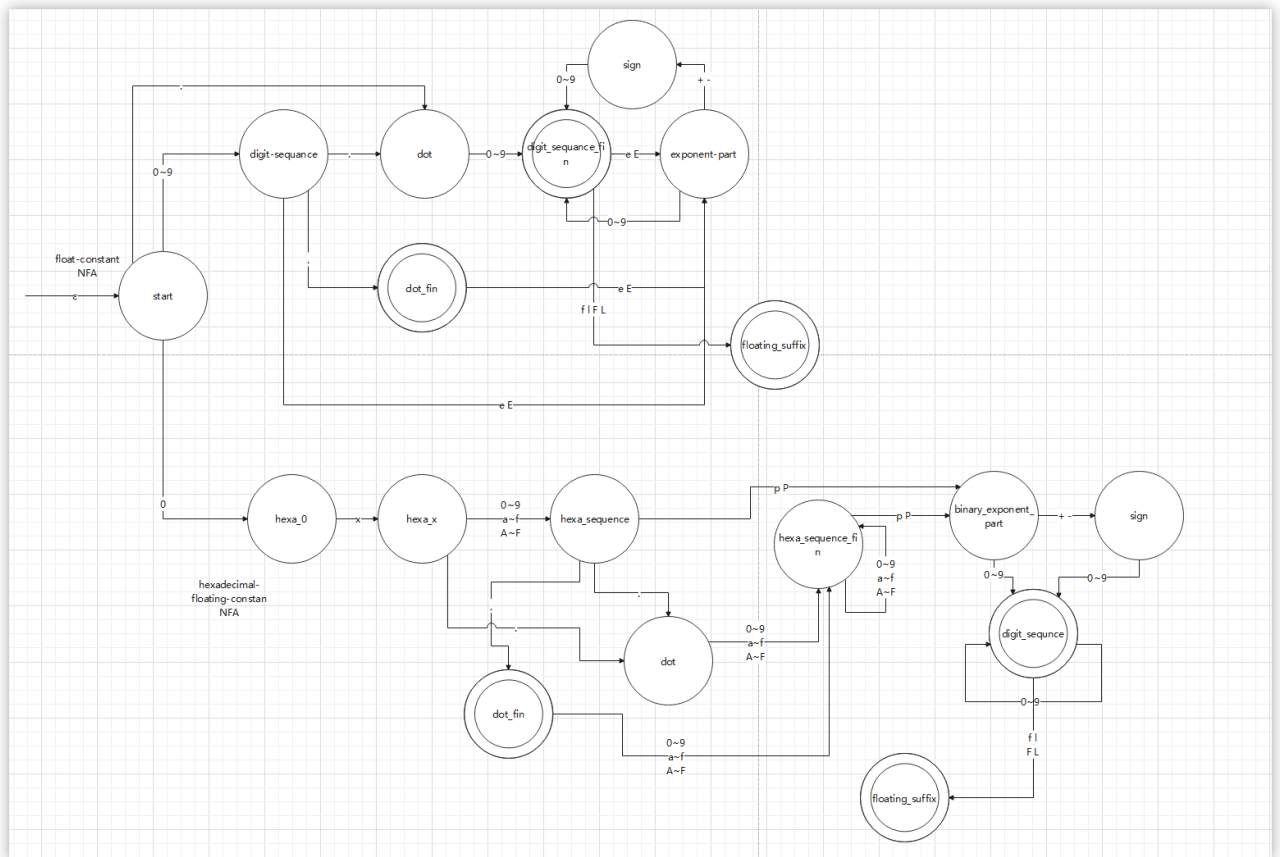
模块内的子模块构造

数字常量识别模块内需要识别整型与浮点型，而整形又分为十进制、八进制、十六进制，浮点型分为十进制浮点和十六进制浮点。因此，这个子模块内的子模块为：

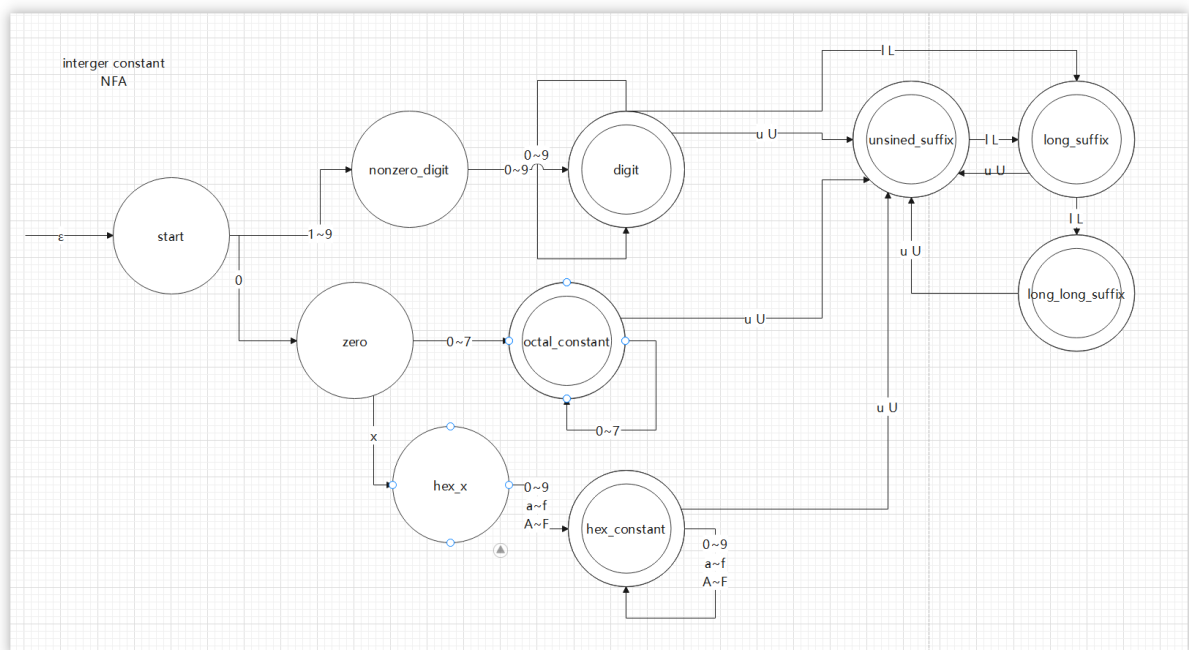
- 整型
 - 十进制
 - 八进制
 - 十六进制
- 浮点型
 - 十进制
 - 十六进制

根据 C 语言标准文档构造非确定自动机如下：

浮点型识别非确定自动机:

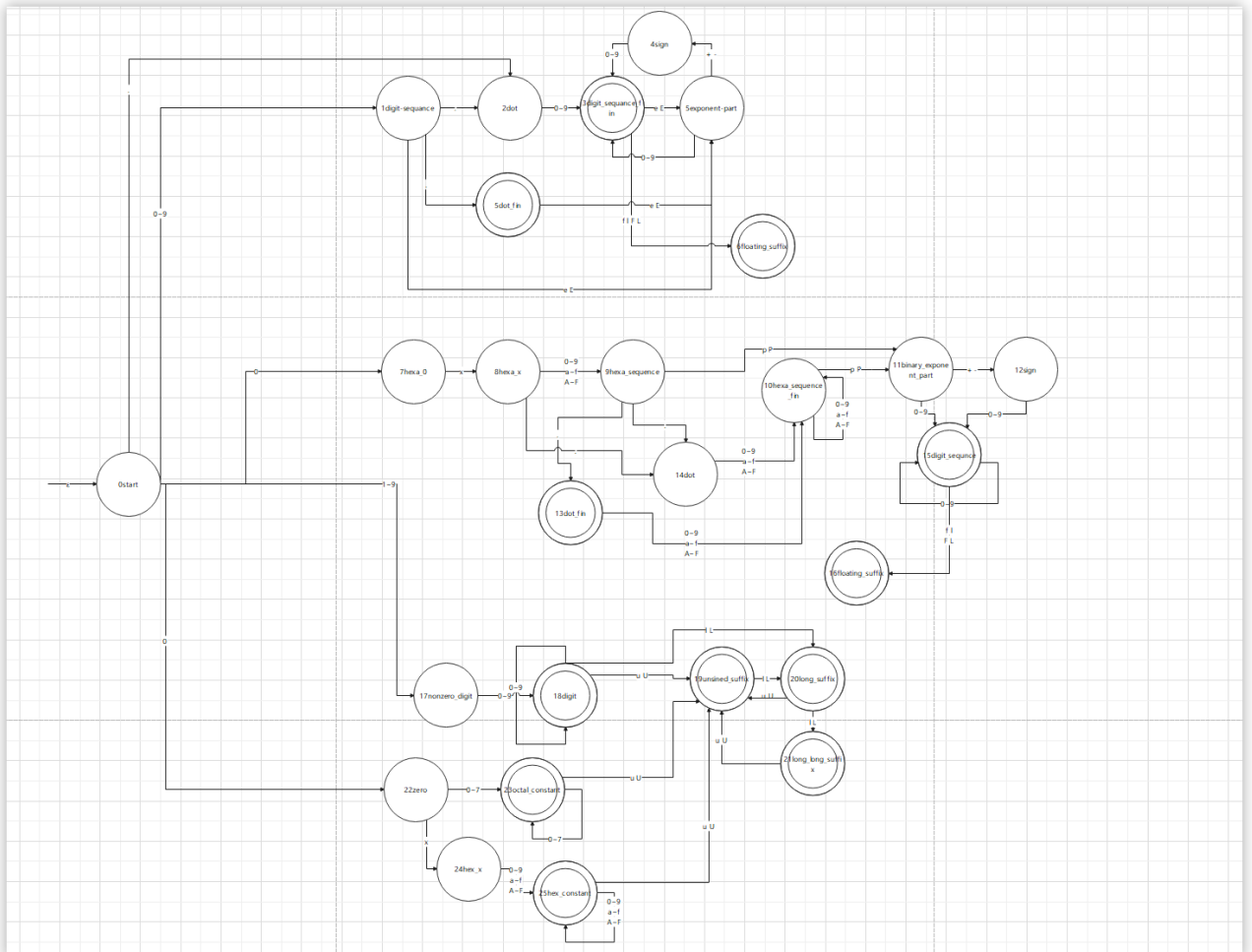


整型识别非确定自动机:



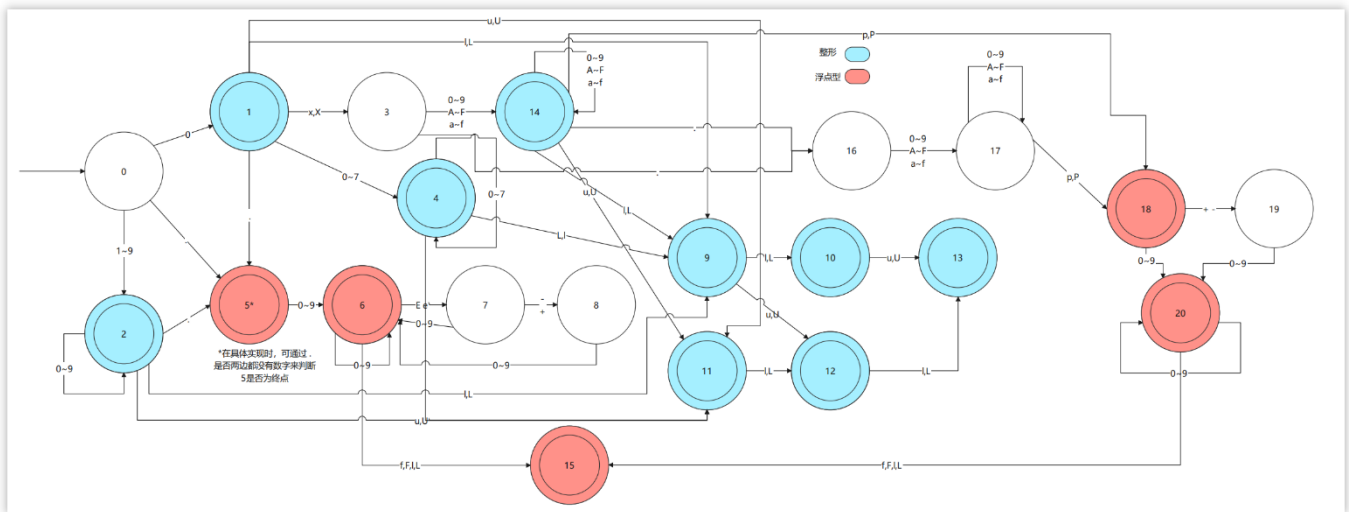
自动机拼接

在分别构造完整形和浮点型的 NFA 后，将其拼接，并简单合并一些模块获得可识别数字型常量的 NFA：



自动机确定化

NFA 合并后，需要将其确定化，以方便编程实现，最后确定化结果如下：



常量识别结果落在橙色圈内，识别为浮点数，落在蓝色圈内识别为整形。

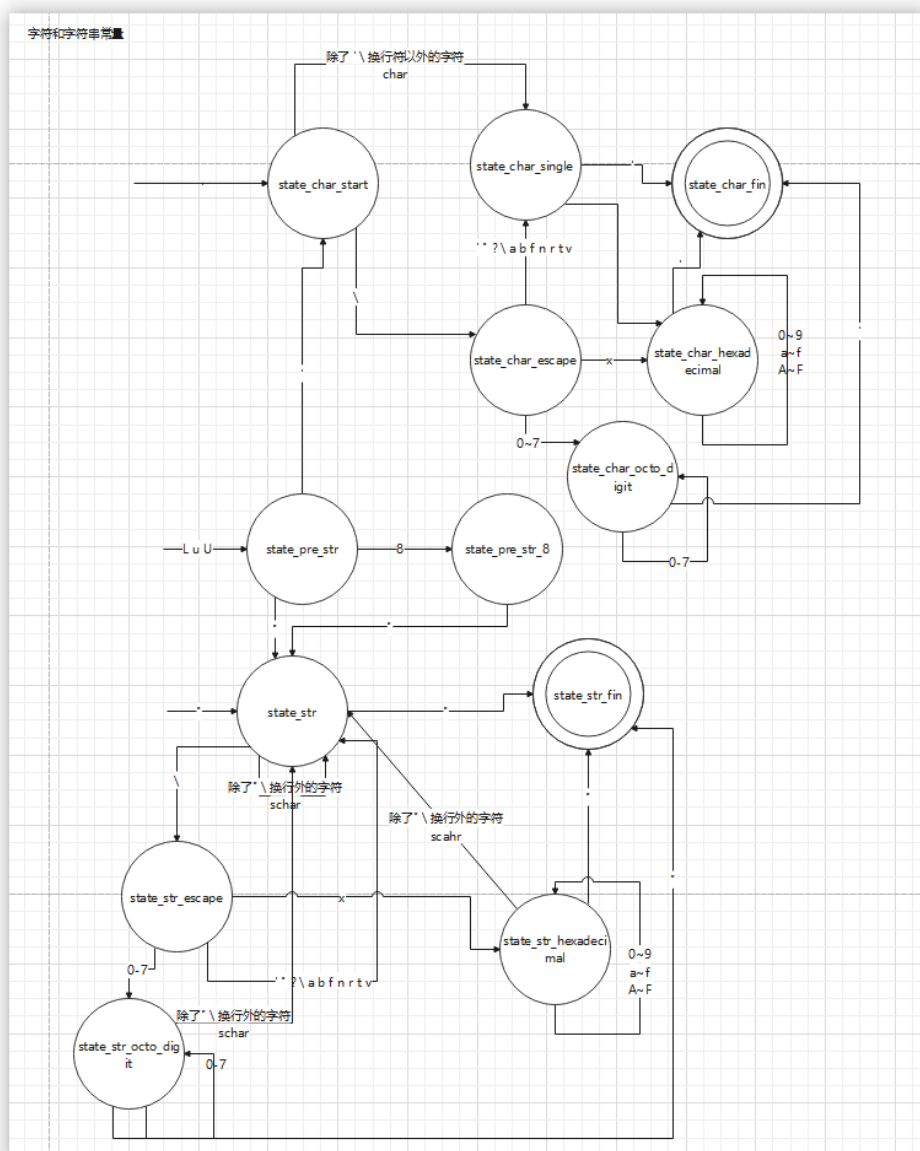
模块间的拼装

在最后将各个部分的自动机拼装时，受老师的关键字 keyword 部分使用 hashset 识别的启发，发现在具体实现时并不一定需要将所有大模块统一生成一个自动机。最后在拼装时，我将 C 语言关键字 keyword、变量标识符 identifier 与常量放入一个模块、操作符和字符串常量放入一个模块，程序扫描单词后初步判断该单词属于哪个模块，最后将单词分别送入该模块内的自动机进行识别，判断单词类型。

编程实现

程序以 ExampleScanner.java 为框架进行改造。

在编程实现时，字符常量和字符串识别模块、运算符模块、常量和变量模块放在同一个自动机内判断，根据字符串识别为 DFA STATE 所新添加的状态如下：



运算符识别模块

运算符主要分为单目运算符和双目/三目运算符。不能与其他符号结合成多目运算符的符号直接识别并打印，将可能是多目运算符的符号存放在 hashset 中，再进行匹配。

判断运算符“是否可结合”可通过函数 `isMultipleOperatorChar` 判断：

```
private boolean isMultipleOperatorChar(char c) {  
    if (c == '+' || c == '-' || c == '*' || c == '/' || c == '=' || c == '<' || c == '>' || c == '%' || c == '[' || c == '#' || c == '%' || c == ':' || c == '!' || c == '^') {  
        return true;  
    }  
    return false;  
}
```

将可能多目的运算符及其可能出现的单目情况放入 HashSet 中，以此识别。

```
private void multipleOperatorAdd() {
    this.operatorSet = new HashSet<String>();

    this.operatorSet.add("++");
    this.operatorSet.add("+");
    this.operatorSet.add("--");
    this.operatorSet.add("-");
    this.operatorSet.add("<<");
    this.operatorSet.add("<");
    this.operatorSet.add(">>");
    this.operatorSet.add(">");
    this.operatorSet.add(">=");
    this.operatorSet.add("<=");
    this.operatorSet.add("=");
    this.operatorSet.add("==");
    this.operatorSet.add("!=");
    this.operatorSet.add("!");
    this.operatorSet.add("&&");
    this.operatorSet.add("&");
    this.operatorSet.add("||");
    this.operatorSet.add("|");
    this.operatorSet.add("*=");
    this.operatorSet.add("*");
    this.operatorSet.add("/=");
    this.operatorSet.add("/");
    this.operatorSet.add("%=");
    this.operatorSet.add("%");
    this.operatorSet.add("+=");
    this.operatorSet.add("-=");
    this.operatorSet.add("<=");
```

常量/变量识别模块

在这个模块内，同时识别了：变量 identifier、数字常量、C 语言关键字 keyword。其中 C 语言关键字和变量 identifier，与数字常量有如下明显区别：

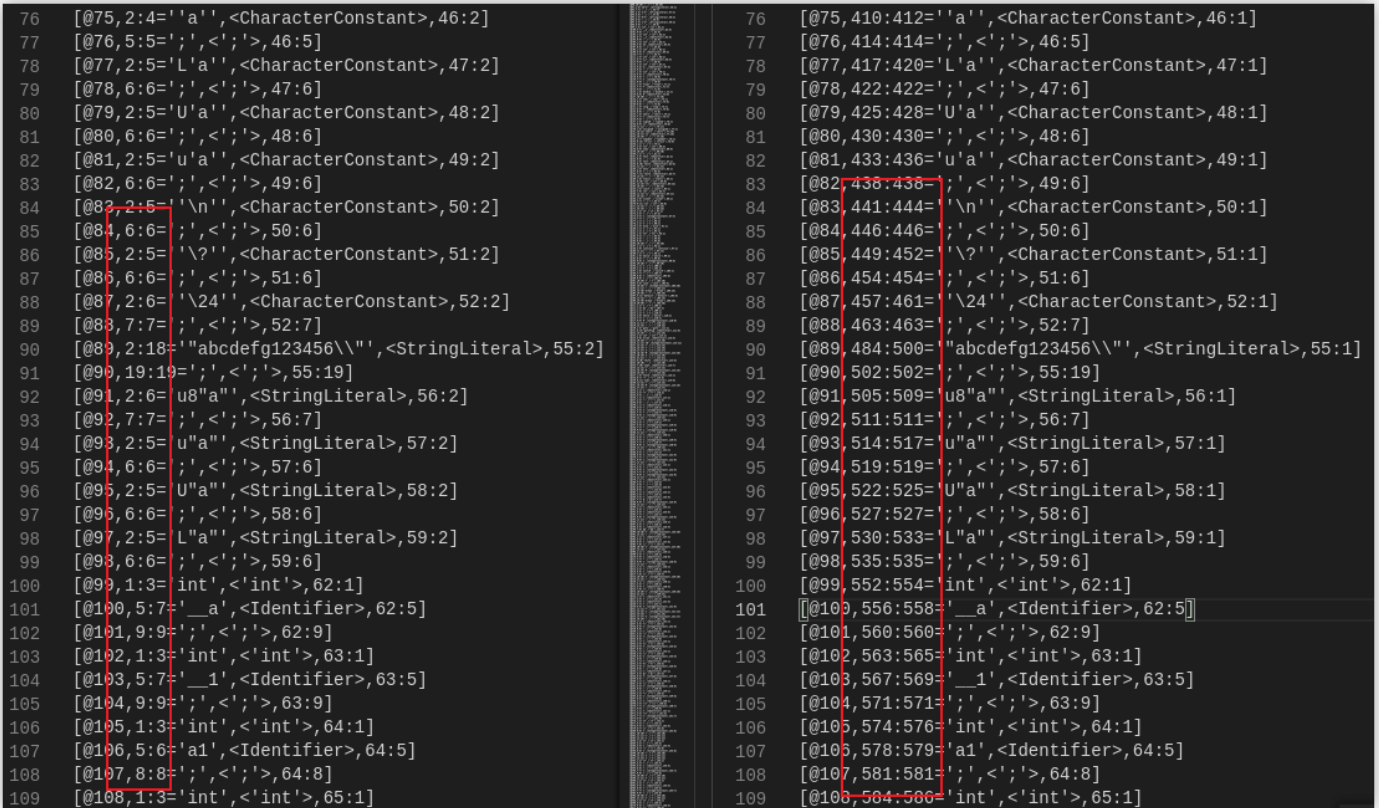
- 数字常量一定以数字开头

通过这个差别，就可以简单的将数字常量同剩下两种类型区分开来。C 语言关键字 keyword 重用了框架已有的代码，在原有的 HashSet 中补全了 C 语言所有的支持的关键字。

将此模块的单词放入有限状态自动机 analyzeConstType(lexme)中，若返回 0，则识别结果为整形常量 IntegerConstant，若返回 1，则识别结果为浮点常量 FloatingConstant，若返回 -1，则识别失败，此为非法常量。

结果分析

通过对 test/scan_test/1_scanner_test.c 文件进行词法分析，运行结果如下：



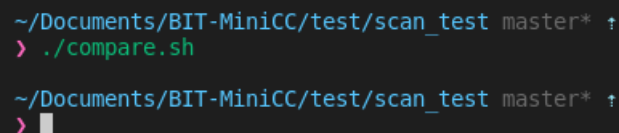
```
76  [@75,2:4='a',<CharacterConstant>,46:2]
77  [@76,5:5=';',<'>',46:5]
78  [@77,2:5='L'a',<CharacterConstant>,47:2]
79  [@78,6:6=';',<'>',47:6]
80  [@79,2:5='U'a',<CharacterConstant>,48:2]
81  [@80,6:6=';',<'>',48:6]
82  [@81,2:5='u'a',<CharacterConstant>,49:2]
83  [@82,6:6=';',<'>',49:6]
84  [@82,2:5='\n',<CharacterConstant>,50:2]
85  [@84,6:6=';',<'>',50:6]
86  [@85,2:5='?',<CharacterConstant>,51:2]
87  [@86,6:6=';',<'>',51:6]
88  [@87,2:6='\24',<CharacterConstant>,52:2]
89  [@88,7:7=';',<'>',52:7]
90  [@89,2:18='\"abcdefg123456\\',<StringLiteral>,55:2]
91  [@90,19:19=';',<'>',55:19]
92  [@91,2:6='u8\"a',<StringLiteral>,56:2]
93  [@92,7:7=';',<'>',56:7]
94  [@93,2:5='\"a',<StringLiteral>,57:2]
95  [@94,6:6=';',<'>',57:6]
96  [@95,2:5='\"a',<StringLiteral>,58:2]
97  [@96,6:6=';',<'>',58:6]
98  [@97,2:5='\"a',<StringLiteral>,59:2]
99  [@98,6:6=';',<'>',59:6]
100 [@99,1:3='int',<'int'>,62:1]
101 [@100,5:7='__a',<Identifier>,62:5]
102 [@101,9:9=';',<'>',62:9]
103 [@102,1:3='int',<'int'>,63:1]
104 [@103,5:7='__1',<Identifier>,63:5]
105 [@104,9:9=';',<'>',63:9]
106 [@105,1:3='int',<'int'>,64:1]
107 [@106,5:6='a1',<Identifier>,64:5]
108 [@107,8:8=';',<'>',64:8]
109 [@108,1:3='int',<'int'>,65:1]

76  [@75,410:412='a',<CharacterConstant>,46:1]
77  [@76,414:414=';',<'>',46:5]
78  [@77,417:420='L'a',<CharacterConstant>,47:1]
79  [@78,422:422=';',<'>',47:6]
80  [@79,425:428='U'a',<CharacterConstant>,48:1]
81  [@80,430:430=';',<'>',48:6]
82  [@81,433:436='u'a',<CharacterConstant>,49:1]
83  [@82,438:438=';',<'>',49:6]
84  [@83,441:444='\n',<CharacterConstant>,50:1]
85  [@84,446:446=';',<'>',50:6]
86  [@85,449:452='?',<CharacterConstant>,51:1]
87  [@86,454:454=';',<'>',51:6]
88  [@87,457:461='\24',<CharacterConstant>,52:1]
89  [@88,463:463=';',<'>',52:7]
90  [@89,484:500='\"abcdefg123456\\',<StringLiteral>,55:1]
91  [@90,502:502=';',<'>',55:19]
92  [@91,505:509='u8\"a',<StringLiteral>,56:1]
93  [@92,511:511=';',<'>',56:7]
94  [@93,514:517='\"a',<StringLiteral>,57:1]
95  [@94,519:519=';',<'>',57:6]
96  [@95,522:525='\"a',<StringLiteral>,58:1]
97  [@96,527:527=';',<'>',58:6]
98  [@97,530:533='\"a',<StringLiteral>,59:1]
99  [@98,535:535=';',<'>',59:6]
100 [@99,552:554='int',<'int'>,62:1]
101 [@100,556:558='__a',<Identifier>,62:5]
102 [@101,560:560=';',<'>',62:9]
103 [@102,563:565='int',<'int'>,63:1]
104 [@103,567:569='__1',<Identifier>,63:5]
105 [@104,571:571=';',<'>',63:9]
106 [@105,574:576='int',<'int'>,64:1]
107 [@106,578:579='a1',<Identifier>,64:5]
108 [@107,581:581=';',<'>',64:8]
109 [@108,584:586='int',<'int'>,65:1]
```

左边为自行实现的程序，右边为 BITMINICC 自带的闭源代码的实现。除了列号外并无差别。

为了验证完全正确，我使用 awk 格式化打印除列号外的部分，再通过 diff 进行比较。脚本如下，对应于 compare.sh：

```
a='./my_paraphrase.txt'
b='./target_paraphrase.txt'
cat ./1_scanner_test.tokens| awk -F ' ' '{print substr($2,index($2,"\""),length($2)) $3}' > $a
cat ./target_token.tokens| awk -F ' ' '{print substr($2,index($2,"\""),length($2)) $3}' > $b
diff $a $b
```



```
~/Documents/BIT-MiniCC/test/scan_test master* ↑
> ./compare.sh

~/Documents/BIT-MiniCC/test/scan_test master* ↑
>
```

通过运行结果可知，两者输出结果相同。词法分析输出的属性字流满足要求。

实验心得体会

词法分析实验的自动机构造要比想象的困难许多。即使是语法结构较为简单、语法限制较多的 C 语言，在其自动机的实际构造中，也感受到了困难，同时也得知语言中许多较为灵活的词法特性。

在编程实现阶段，对于关键字和运算符的识别，使用了 `hashset` 这种较为取巧的方法，在我看来这相当于为原本该自动机识别的部分引入了部分图灵机的算法与功能，使得识别变得简便了许多，这让我体会到了不同计算模型间计算能力的巨大差别。