

实验报告

一、实验目的

1. 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;
2. 了解 Flex 工作原理和基本思想,学习使用工具自动生成词法分析器;
3. 掌握编译器从前端到后端各个模块的工作原理,词法分析模块与其他 模块之间的交互过程。

二、实验内容

根据 C 语言的词法规则,设计识别 C 语言所有单词的词法分析器的确定有限自动机,并使用 C\C++ 语言,采用程序中心法或数据中心法设计并实现词法分析器。词法分析器的输入为 C 语言源程序,输出为属性字流。

使用 Flex 自动生成词法分析器的 C 语言源代码,并使用 gcc 编译工具进行编译,生成可运行的词法分析器程序。通过 BIT-MINICC 框架,完成词法分析器与框架内其它编译模块的联调测试,完成对词法分析器的功能检验。

三、实现步骤

1、配置实验环境

本次实验在 Linux 环境下进行。

使用命令: `sudo apt-get install flex` 即可完成 flex 环境的安装。

从网址: <https://github.com/jiweixing/bit-minic-compiler> 中下载 BIT-MINICC 框架代码。

从 eclipse 官方网站中下载并安装 eclipse 开发环境。结合 BIT-MINICC 介绍 PPT,使用 eclipse 导入项目,并按照介绍完成相关参数的配置。

2、观察词法分析器输入输出

在完成实验环境的搭建,并完成对 BIT-MINICC 框架在 eclipse 项目中的配置后,运行框架自带的词法分析器生成属性字节流。观察属性字节流,并结合实验要求文档中对 C 语言词法规则的介绍,了解词法分析器的输入输出结构。

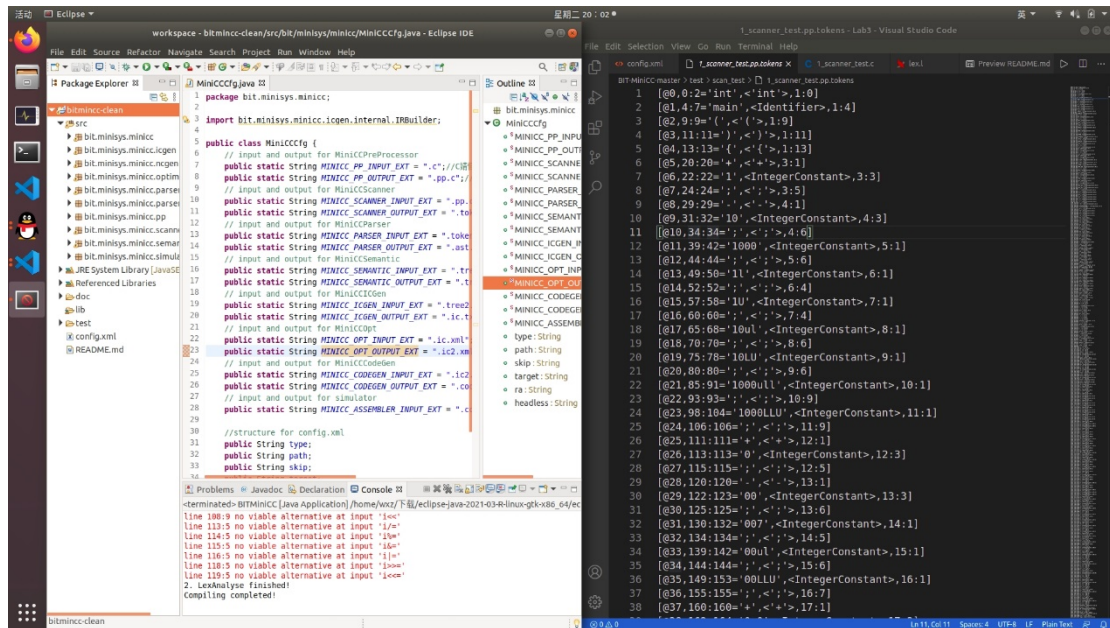


图 1 eclipse 下运行 BIT-MINICC 执行词法分析结果

从上图中，屏幕左侧区域为从 eclipse 载入的 BIT-MINICC 框架。它的下部展示了框架执行词法分析后的结果。屏幕右侧区域为使用 vscode 打开的词法分析器生成的属性字节流。从文件中，我们可以观察到输出的属性字节流与 C 语言词法特征的对应关系。

在下侧命令台中，程序输出了执行后的结果。这里出现了一些报错，经查看后，为预处理器对单词的错误处理所造成的结果。对预处理、未预处理的程序同时进行词法分析后，发现两种情况下均会出现报错。这些错误包括预处理程序对浮点数据错误解析，对如 $<<=$ 、 $>=$ 的运算符的错误处理，词法分析器对词法解释的错误。按照微信群中老师的说明，这些错误是词法分析器的错误，可以通过屏蔽词法分析阶段来避免错误。

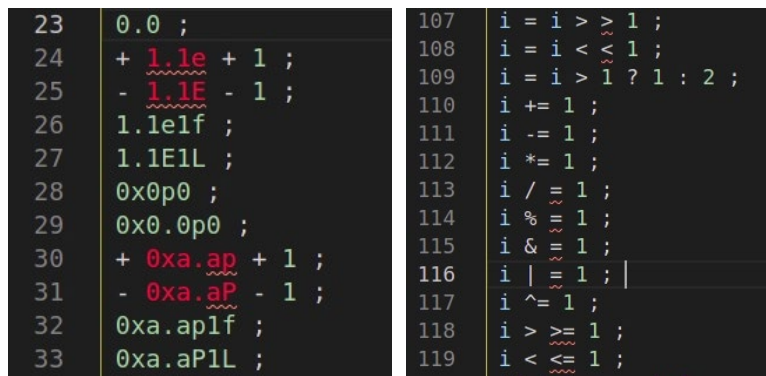


图 2 BIT-MINICC 框架与编译程序的错误

3、实现词法分析器

编写词法分析器源程序，主要的工作在于设计若干正则表达式。通过这些正则表达式，对 C 语言程序中的各单词进行准确的识别。实验要求文档中对 C 语言各类关键词的定义将 C 语言中的单词分为了 8 类。因此设计 8 个正则表达式来对应识别这些单词。

- notes /*(.\\r\\n)*\\/\\/(.)
- keywords
auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|inline|int|long|register|restrict|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while
- op [](){}|->|+|--|&*|+|-
|~|!|/%|<<|>>|<|>|<=|>=|=|=|!=|^|&&|||?:|;|...|=*=|=/=|%=|+=|-=|<=<|>=>|&=|^=|=|=|,|#|##|<:<%|>|%:%:|
- identifier [_a-zA-Z][_a-zA-Z0-9]*
- intconst ((([0-9]+)|([0-7]+)|((0x|0X)[0-9a-fA-F]+))(((u|U)?(l|L|ll|LL)?)|((l|L|ll|LL)?(u|U)?)))
- floatconst (((([0-9]+.[0-9])|([0-9]+\.[0-9](e|E)(+|-)?[0-9]+))|(0x|0X)(([0-9a-fA-F]+)(.[0-9a-fA-F]*)?(p|P)(+|-)?[0-9]+)))([fFIlI])?
- charconst [uUL]?('[^']|[\\0-9]*|'\"|\\?\\\\|\\a|\\b|\\f|\\n|\\r|\\t|\\v)'
- stringconst (u|u8|U|L)?("([0-9a-zA-Z.!~'|\\(\\0-9)]|'\"|\\?\\\\|\\a|\\b|\\f|\\n|\\r|\\t|\\v)"

在输出方面,按照要求的属性字节流输出格式,需要输出识别出的单词的行数、列数、起止列、累计单词数等信息。因此定义了6个C语言下的int型变量,用于存储这些信息。在匹配到相应的单词后,及时刷新上述信息,并进行输出。

图 3 flex 词法分析器格式化输出代码

同时为了调试的便利，我设计的词法分析器提供三种模式：

1. 命令台输入输出
2. 文件输入命令台输出
3. 文件输入文件输出

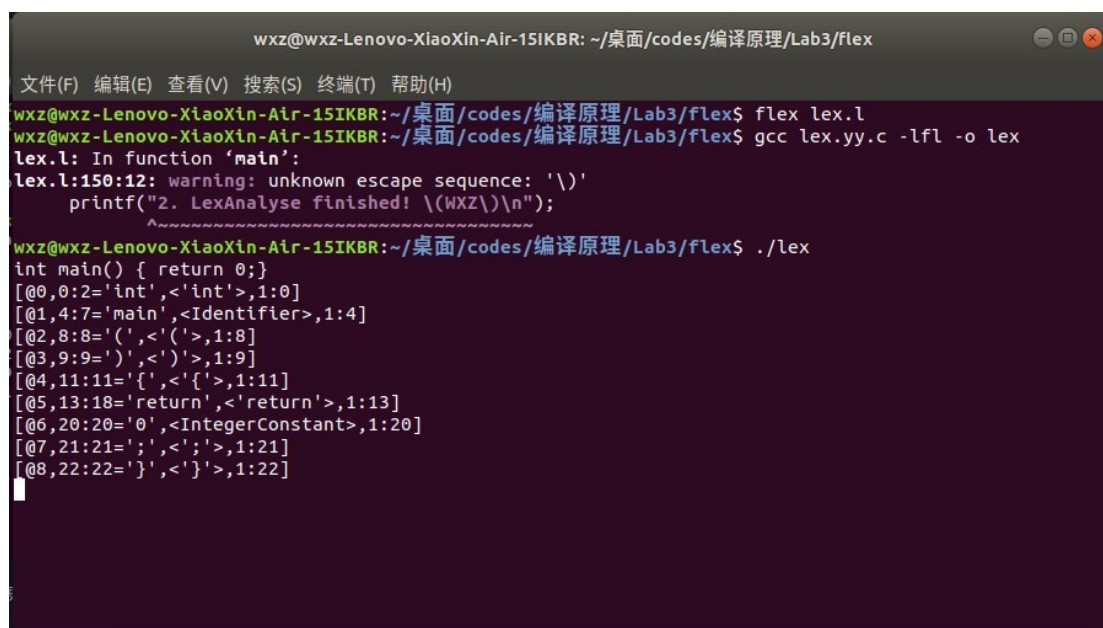
第一种模式为了调试程序方便，在命令台中直接输入相应的程序代码即可进行解析；第三种模式匹配了 BIT-MINICC 框架中的输入输出模式，通过命令行参数指定输入、输出文件的文件路径名，词法分析器即可根据文件路径完成对相应文件的词法分析及输出操作。

4、使用命令生成词法分析器

使用命令：flex lex.l 即可生成名为 lex.yy.c 的词法分析器源程序

使用命令：gcc lex.yy.c -lfl -o lex 即可编译 lex.yy.c 并在同目录下生成名为 lex 的词法分析器可执行程序

使用命令：./lex 即可运行词法分析器程序



```
WXZ@WXZ-Lenovo-XiaoXin-Air-15IKBR: ~/桌面/codes/编译原理/Lab3/flex
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
WXZ@WXZ-Lenovo-XiaoXin-Air-15IKBR:~/桌面/codes/编译原理/Lab3/flex$ flex lex.l
WXZ@WXZ-Lenovo-XiaoXin-Air-15IKBR:~/桌面/codes/编译原理/Lab3/flex$ gcc lex.yy.c -lfl -o lex
lex.l: In function 'main':
lex.l:150:12: warning: unknown escape sequence: '\)'
    printf("2. LexAnalyse finished! \\\n");
    ~~~~~^
WXZ@WXZ-Lenovo-XiaoXin-Air-15IKBR:~/桌面/codes/编译原理/Lab3/flex$ ./lex
int main() { return 0;}
[@0,0:2='int',<'int'>,1:0]
[@1,4:7='main',<Identifier>,1:4]
[@2,8:8='(',<'('>,1:8]
[@3,9:9=')',<')'>,1:9]
[@4,11:11='{',<'{'>,1:11]
[@5,13:13='return',<'return'>,1:13]
[@6,20:20='0',<IntegerConstant>,1:20]
[@7,21:21=';',<'>',1:21]
[@8,22:22=']',<']'>,1:22]
```

图 4 测试运行词法分析器程序

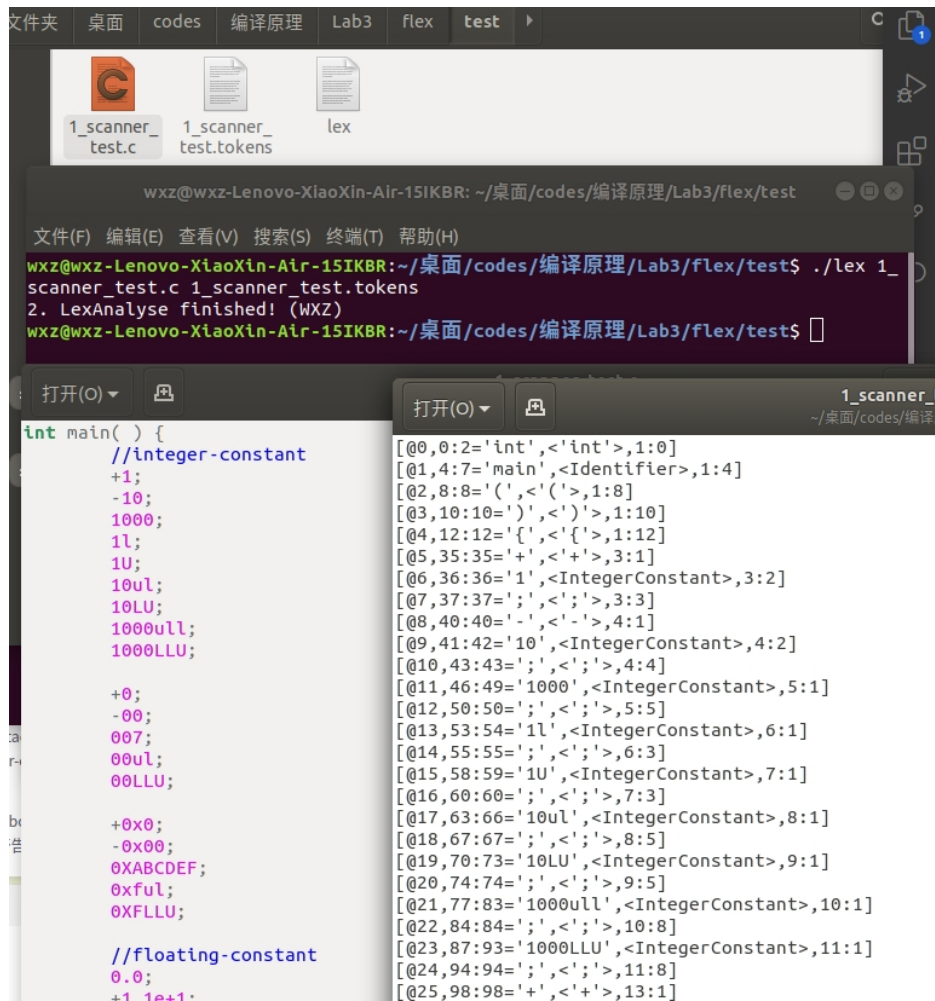
上图为使用命令进行编译、执行的过程。在完成对词法分析器的生成、编译、运行后，使用了一句简单的代码对其进行测试。经过分析，词法分析器可以正确的解析各个单词的意义，基本实现了功能。

四、运行效果

本部分使用 BIT-MINICC 框架中提供的词法分析测试代码进行测试。

1、词法分析器单独运行效果

在终端使用命令对词法分析器的功能进行测试。使用命令行参数指定待分析文件的路径和属性字节流输出路径。



```
wxz@wxz-Lenovo-XiaoXin-Air-15IKBR: ~/桌面/codes/编译原理/Lab3/flex/test
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
wxz@wxz-Lenovo-XiaoXin-Air-15IKBR:~/桌面/codes/编译原理/Lab3/flex/test$ ./lex 1_scanner_test.c 1_scanner_test.tokens
2. LexAnalyse finished! (WXZ)
wxz@wxz-Lenovo-XiaoXin-Air-15IKBR:~/桌面/codes/编译原理/Lab3/flex/test$
```

```
int main( ) {
    //integer-constant
    +1;
    -10;
    1000;
    1l;
    1U;
    10ul;
    10LU;
    1000ull;
    1000LLU;

    +0;
    -00;
    007;
    00ul;
    00LLU;

    +0x0;
    -0x00;
    0XABCDEF;
    0xful;
    0XFLLU;

    //floating-constant
    0.0;
    +1.1e+1;
}
```

```
[@0,0:2='int',<'int'>,1:0]
[@1,4:7='main',<Identifier>,1:4]
[@2,8:8='(',<'('>,1:8]
[@3,10:10=')',<')'>,1:10]
[@4,12:12='{',<'{'>,1:12]
[@5,35:35='+',<'+'>,3:1]
[@6,36:36='1',<IntegerConstant>,3:2]
[@7,37:37=';',<'>',3:3]
[@8,40:40='-',<'-'>,4:1]
[@9,41:42='10',<IntegerConstant>,4:2]
[@10,43:43=';',<'>,4:4]
[@11,46:49='1000',<IntegerConstant>,5:1]
[@12,50:50=';',<'>,5:5]
[@13,53:54='1l',<IntegerConstant>,6:1]
[@14,55:55=';',<'>,6:3]
[@15,58:59='1U',<IntegerConstant>,7:1]
[@16,60:60=';',<'>,7:3]
[@17,63:66='10ul',<IntegerConstant>,8:1]
[@18,67:67=';',<'>,8:5]
[@19,70:73='10LU',<IntegerConstant>,9:1]
[@20,74:74=';',<'>,9:5]
[@21,77:83='1000ull',<IntegerConstant>,10:1]
[@22,84:84=';',<'>,10:8]
[@23,87:93='1000LLU',<IntegerConstant>,11:1]
[@24,94:94=';',<'>,11:8]
[@25,98:98='+',<'+'>,13:1]
```

图 5 词法分析器单独运行结果

上图展示了词法分析器的执行结果。在指定的路径下，程序生成了对应的属性字节流。经过对比，生成的属性字节流与 BIT-MINICC 框架中词法分析器输出的属性字节流结果一致，是正确的。

2、词法分析器与 BIT-MINICC 框架联调

在 BIT-MINICC 框架中，config.xml 指定了程序编译过程中各个阶段所使用的程序文件。在参数缺省的情况下，框架会调用框架默认的程序文件完成编译。也可以通过指定参数，指定编译的各个阶段所使用的程序。因此，通过编译 config.xml 文件，即可使用框架调用我所编写的词法分析器程序。


```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <config name="config.xml">
3      <phases>
4          <phase>
5              <phase skip="false" type="java" path="" name="preprocess" />
6              <phase skip="false" type="c" path="./bin_c/lex" name="scan" />
7              <!-- <phase skip="false" type="java" path="" name="scan" /> -->
8              <phase skip="true" type="java" path="" name="parse" />
9              <phase skip="true" type="java" path="" name="semantic" />
10             <phase skip="true" type="java" path="" name="icgen" />
11             <phase skip="true" type="java" path="" name="optimize" />
12             <phase skip="true" type="java" path="" name="ncgen" target="mips" ra="ls" />
13             <phase skip="true" type="mips" path="" name="simulate" />
14         </phase>
15     </phases>
16 </config>

```

图 6 config.xml 文件配置

在编译器框架下添加名为 bin_c 的文件夹，用于存放 c 语言生成可执行文件。将已经编译生成好的词法分析器拷贝到此文件夹，并在 config.xml 中指定此程序作为词法分析的程。由于主要验证词法分析功能，因此将词法分析后的所有编译环节均进行屏蔽。同时，对执行、不执行预处理的两种情况下词法分析器的运行表现分别进行测试。在 eclipse 环境下，运行框架程序。。

```

Problems  @ Javadoc  Declaration  Console
<terminated> BITMiniCC [Java Application] /home/wxz/下
Start to compile ...
1. PreProcess finished!
2. LexAnalyse finished! (WXZ)
Compiling completed!

```

图 7 BIT-MINICC 框架下调用词法分析器执行结果（使用预编译）

图 7 为使用预处理下，进行编译生成的输出结果。可以看到，框架成功调用了我所编写的词法分析器程序，并在执行结束后打印输出了相应日志。在文件夹中，也输出了相应的属性字节流。经过对比，此属性字节流与 BIT-MINICC 生成的属性字节流一致。

```

Problems  @ Javadoc  Declaration  Console
<terminated> BITMiniCC [Java Application] /home/wxz/下
Start to compile ...
2. LexAnalyse finished! (WXZ)
Compiling completed!

```

图 8 BIT-MINICC 框架下调用词法分析器执行结果（禁用预编译）

图 8 为不使用预处理程序下，进行编译的输出结果。可以看到，框架成功调用了词法分析器。在源程序的目录文件夹下，输出了属性词节流。图 8 展示了执行的结果。经对比检验，在框架调用下我所编写的词法分析器运行正常，结果正确。

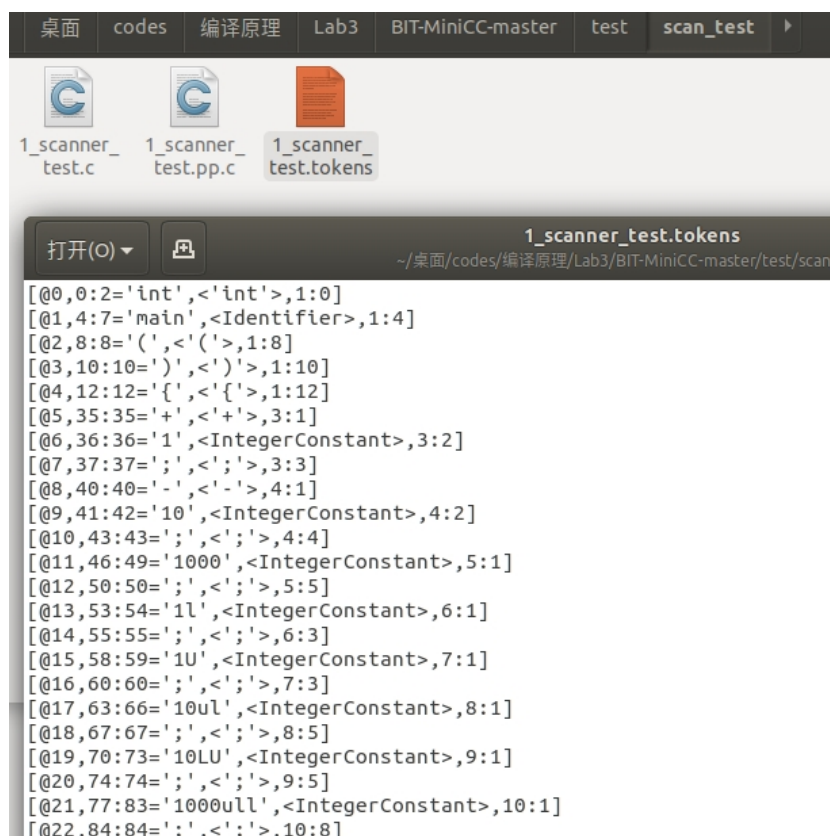


图 9 BIT-MINICC 框架下词法分析执行结果文件

在此次实验过程中，出现了一些错误。在刚开始的编译尝试时，框架并未成功输出属性字节流文件。经检查，源程序文件中被覆盖写入了"EOF"的属性字节流。这个 BUG 是较为奇怪的，因为词法分析器的文件输入、文件输出模式已经进行了测试，并且功能正常。在与同学、老师交流后，发现这是框架部分的 BUG。参考 Github 中最新的一条 pull request，对此 BUG 进行了修复。之后进行的实验操作结果均正常。

五、心得体会

本次实验中，我使用 flex 工具构建了 C 语言的词法分析器程序。本次实验的重点内容在于使用正则表达式描述 C 语言的词法特征。借用 flex 和 gcc 编译器工具可以完成从正则表达式及属性字节流输出格式到可执行词法分析器程序的生成。在完成了词法分析器程序后，我使用 BIT-MINICC 框架调用了词法分析器程序，并进行了词法分析器程序正确性的检测。在实验过程中，也发现了一些程序、框架存在的问题和 BUG。在和老师、同学交流后，这些问题都得到了解决。同时，在实际测验中我设计的词法分析器也有很多不足。如 BIT-MINICC 框架已经实现的词法分析器可以对词法的错误做报错，而我设计的词法分析器并未对此功能进行实现。在进一步的改进中，可以对报错功能进行实现。

通过本次实验，我对词法分析器的设计和工作原理有了更加深刻的认识，对 flex 的使用和 C 语言语法的认识也有了加深。同时，我对于 BIT-MINICC 编译器架构有了一定的了解和认识。在以后的学习中，我要将以后实验中实现的编译各环节程序联调整合，同时对现在的词法分析器进行改进，最终实现一个具有简单功能的编译器程序。