

Name: 王何佳

Student ID: 2023211603

Class Number: 2023211804

Container ID: dcccfaf6754c5a978d4cd8b446a0e67638484164a68d29dd6a796d6b8f467d80

Lab 2

一、实验目的

1. **掌握词法分析基础**: 通过本次实验, 旨在深入理解词法分析在编译器前端中的作用, 并掌握词法分析器的基本工作原理。
2. **学习 Flex 工具的使用**: 学习并熟练使用词法分析器生成工具 Flex, 掌握 lex 文件的基本结构, 包括定义区、规则区和辅助函数区的编写方法。
3. **强化正则表达式应用**: 通过解决实际问题, 加强对正则表达式的理解和应用能力, 能够为复杂的模式 (C 语言标识符、IPv4/IPv6 地址) 编写准确的匹配规则。
4. **实践与问题解决**: 通过完成三个具体的编程任务——实现 wc 功能、修正 identifiers 行号统计以及验证 ipaddr 地址——锻炼分析、解决实际编程问题的能力。

二、实验环境

本次实验的全部操作均在 Docker 容器中进行, 以确保一个纯净、统一的 Linux 开发环境。详细环境配置如下:

容器化工具: Docker Desktop

基础镜像: Ubuntu (通过 `docker pull ubuntu:latest` 命令拉取)

容器 ID:

dcccfaf6754c5a978d4cd8b446a0e67638484164a68d29dd6a796d6b8f467d80

核心编译工具: build-essential flex

测试脚本语言: Python 3 (通过 `apt-get install -y python3` 命令安装)

三、实验内容

本次实验围绕 Flex 工具和正则表达式的应用，共分为三个核心部分：

1. **wc 程序分析**：编译并运行一个使用 Flex 实现的 wc（字数统计）程序，将其输出与系统自带的 wc 命令进行对比。通过分析 lex.l 文件中的正则表达式，探究两者单词统计结果差异的根本原因。
2. **identifiers 程序修正**：针对一个用于识别 C 语言标识符但无法正确输出行号的 Flex 程序，通过修改 lex.l 文件。增加对换行符的匹配规则并维护一个行号计数器，成功修正了其行号统计错误的缺陷。
3. **ipaddr 地址验证**：编写一个能够验证 IPv4 及 IPv6 地址合法性的 Flex 程序。通过设计并实现复杂的正则表达式来精确匹配两种地址的格式规范。最终成功通过了实验提供的 ip_test.py 脚本的所有测试用例。

四、实验过程

1. 配置 docker 环境

安装 docker: <https://www.docker.com/products/docker-desktop/>

拉取一个 Linux 镜像：

```
docker pull ubuntu:latest
```

启动 Docker 容器并挂载实验目录：

```
docker run -it -v "C:\Users\62477\Desktop\22\作业\lab2\lab2":/root/lab2 ubuntu:latest
```

在容器中安装必要的工具：

```
apt-get update          # 更新软件包列表
apt-get install -y build-essential flex  # 安装编译工具
```

```
Setting up g++-13-x86-64-linux-gnu (13.3.0-6ubuntu2~24.04) ...
Setting up gcc-x86-64-linux-gnu (4:13.2.0-7ubuntu1) ...
Setting up gpg-wks-client (2.4.4-2ubuntu17.3) ...
Setting up gcc (4:13.2.0-7ubuntu1) ...
Setting up g++-x86-64-linux-gnu (4:13.2.0-7ubuntu1) ...
Setting up g++-13 (13.3.0-6ubuntu2~24.04) ...
Setting up g++ (4:13.2.0-7ubuntu1) ...
update-alternatives: using /usr/bin/g++ to provide /usr/bin/c++ (c++) in auto mode
update-alternatives: warning: skip creation of /usr/share/man/man1/c++.1.gz because
g++.1.gz (of link group c++) doesn't exist
Setting up build-essential (12.10ubuntu1) ...
Setting up libheif1:amd64 (1.17.6-1ubuntu4.1) ...
Setting up libgd3:amd64 (2.3.3-9ubuntu5) ...
Setting up libc-devtools (2.39-0ubuntu8.6) ...
Setting up libheif-plugin-aomdec:amd64 (1.17.6-1ubuntu4.1) ...
Setting up libheif-plugin-libde265:amd64 (1.17.6-1ubuntu4.1) ...
Setting up libheif-plugin-aomenc:amd64 (1.17.6-1ubuntu4.1) ...
Processing triggers for libc-bin (2.39-0ubuntu8.6) ...
root@dccccfaf6754c:/#
```

2.wc 程序分析

2.1 编译与执行结果

首先对 lab2/wc 目录下的 lex.l 文件进行了编译，并分别使用生成的 wc.out 程序和系统自带的 wc 命令对 inferno3.txt 文件进行处理。

```
cd /root/lab2/wc
make wc          # 编译 wc.out 程序
./wc.out inferno3.txt  # 运行 wc.out 程序
wc inferno3.txt    # 运行系统自带的 wc 命令
```

```
root@dccccfaf6754c:~/lab2/wc# make wc
flex lex.l
gcc lex.yy.c -lfl -o wc.out
root@dccccfaf6754c:~/lab2/wc# ./wc.out inferno3.txt
162      1088      6525      inferno3.txt
root@dccccfaf6754c:~/lab2/wc# wc inferno3.txt
162 1074 6525 inferno3.txt
```

2.2 单词统计差异分析

在对 inferno3.txt 文件进行单词统计时，编译的程序 ./wc.out 与系统自带的 wc 命令得到了不同的结果。虽然行数和字符数统计一致，但单词数存在差异：./wc.out inferno3.txt 的输出结果为 1088 个单词，而 wc inferno3.txt 的输出结果为 1074 个

单词。

核心原因：对“单词”的不同定义

- **系统 wc 命令：**Linux 系统中的 wc 命令将由一个或多个空白字符（空格、制表符\t、换行符\n）分隔的非空白字符序列均视为一个独立的单词。因此，hello-world、isn't 以及纯数字 2025 都会被计为一个单词。
- **wc.out 程序：**其行为完全由 lex.l 文件中的正则表达式决定。

lex.l 文件中的正则表达式分析

```
// 定义部分
letter [a-zA-Z]

%%
// 规则部分
{letter}+ { words++; chars+=strlen(yytext); }
...
```

代码首先定义了 **letter** 为任意一个英文字母[a-zA-Z]。接着，规则部分使用{letter}+来匹配单词，只有连续一个或多个英文字母组成的字符串才被视为一个单词。

规则的局限性：此规则不会匹配任何非字母字符。因此，当遇到连字“-”、撇号“'”、数字或任何其他标点符号时，单词的匹配就会被中断。

结论：wc.out 统计出的单词数（1088）比系统 wc（1074）多了 14 个。这表明，在 inferno3.txt 文件中，存在 14 个被系统 wc 视为单个单词，但被 wc.out 程序因其内部包含的非字母字符而拆分为两个单词进行计数的情况。

3.Flex 练习-identifiers

3.1 编译与执行结果

在修改代码之前，先编译并运行一下现有程序，看看错误的输出是什么样的。

```
cd /root/lab2/identifiers
make idcount          # 编译程序
./idcount.out ./test.c # 运行程序
```

```
root@dccccfaf6754c:~# cd /root/lab2/identifiers
root@dccccfaf6754c:~/lab2/identifiers# make idcount
flex lex.l
gcc lex.yy.c -lfl -o idcount.out
root@dccccfaf6754c:~/lab2/identifiers# ./idcount.out ./test.c
line 0: int
line 0: main
line 0: int
line 0: a
line 0: int
line 0: BBA
line 0: a_
line 0: int
line 0: _
line 0: int
line 0: a
line 0: if
line 0: a0
line 0: _
line 0: b0
line 0: else
line 0: _
line 0: b0
line 0: while
line 0: b
line 0: b1
line 0: b0
line 0: b2
line 0: char
line 0: c
line 0: a
line 0: return
There are 27 occurrences of valid identifiers
```

行号全为 0，明显有误。

3.2 修改 lex.l

(1) 修改行号初始值

```
# line 7
int lines = 1;    # 原为 int lines = 0;
```

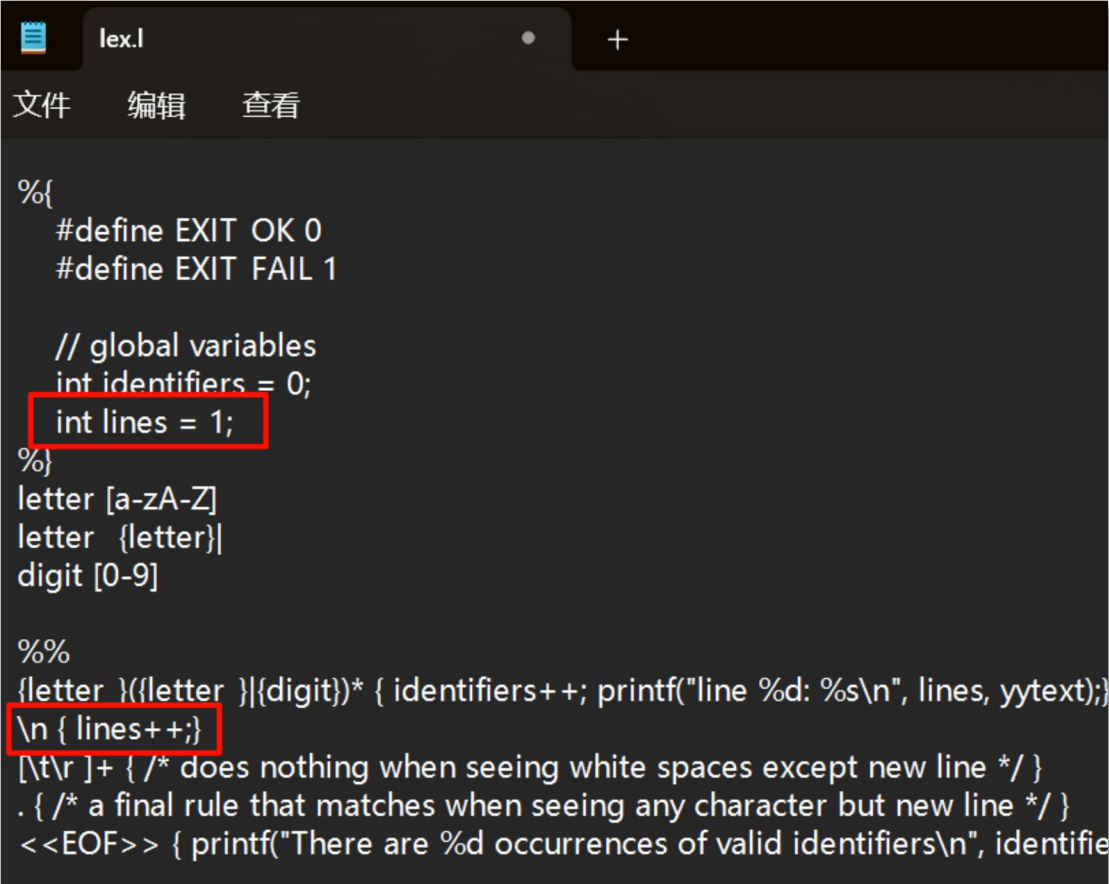
原因：在编程和文本文件中，我们通常习惯于从第 1 行开始计数，这更符合直觉和实际情况。

(2) 为换行符添加动作

```
# line 15
\n { lines++; } # 原为\n {}
```

原因：原来的代码 `\n {}` 执行空操作，导致 `lines` 变量的值不变。`\n { lines++; }` 规则

让程序在每读完一行时，都将行号 `lines` 加一，从而实现了正确的行号跟踪。



```
lex.l

文件 编辑 查看

%{
    #define EXIT_OK 0
    #define EXIT_FAIL 1

    // global variables
    int identifiers = 0;
    int lines = 1;
%}
letter [a-zA-Z]
letter {letter}
digit [0-9]

%%
{letter }({letter }){digit}* { identifiers++; printf("line %d: %s\n", lines, yytext);}
\n { lines++;}
[\t\r ]+ { /* does nothing when seeing white spaces except new line */ }
. { /* a final rule that matches when seeing any character but new line */ }
<<EOF>> { printf("There are %d occurrences of valid identifiers\n", identifiers); }
```

3.3 验证结果

```
make idcount          # 重新编译程序
./idcount.out ./test.c # 运行程序并检查结果
```

```

root@dccccfaf6754c:~/lab2/identifiers# make idcount
flex lex.l
gcc lex.yy.c -lfl -o idcount.out
root@dccccfaf6754c:~/lab2/identifiers# ./idcount.out ./test.c
line 1: int
line 1: main
line 3: int
line 3: a
line 4: int
line 4: BBA
line 4: a_
line 5: int
line 5: _
line 6: int
line 6: a
line 7: if
line 7: a0
line 7: _
line 7: b0
line 8: else
line 8: _
line 8: b0
line 9: while
line 9: b
line 9: b1
line 9: b0
line 9: b2
line 10: char
line 10: c
line 10: a
line 11: return
There are 27 occurrences of valid identifiers

```

现在行号是正确的。

4. Flex 练习-ipaddr

4.1 编写 lex.l 文件

(1) 核心正则表达式的设计

```

dec_octet    ([1-9]?[0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])
hex_group    [0-9a-fA-F]{1,4}
v4           {dec_octet}(\.{dec_octet}){3}
v6           ({hex_group}{hex_group}){7}|(:{hex_group}{hex_group}){0,6})|({hex_group}::({hex_group}{hex_group}){0,5})|({hex_group}{hex_group}){1}::({hex_group}{hex_group}){0,4})|({hex_group}{hex_group}){2}::({hex_group}{hex_group}){0,3})|({hex_group}

```

```
(:{hex_group}{3}::({hex_group}(:{hex_group}){0,2})?)|({hex_group}(:{hex_group}){4}::({hex_group}(:{hex_group}){0,1})?)|({hex_group}(:{hex_group}){5}::({hex_group})?)|({hex_group}(:{hex_group}){6}::)
```

(2) IPv4 正则表达式

首先定义子模式 `dec_octet`，用于匹配一个 0 到 255 之间的数字。
([1-9]?[0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])这个表达式通过“|”将数字范围划分为 0-99、100-199、200-249 和 250-255。

其中[1-9]?[0-9]能匹配一位数和两位数，同时有效地防止了如“01”这样不合法的“前导零”。

最终的 v4 模式`{dec_octet}\.{dec_octet}{3}`描述了“由四个 `dec_octet` 数字和三个点号‘.’组成”的完整结构。

(3) IPv6 正则表达式

通过枚举的方式覆盖了所有可能的合法 IPv6 结构。

主要包含两种情况：

- 完整形式：如`(:{hex_group}(:{hex_group}){7})`，匹配由 8 个十六进制段组成的未压缩地址。
- 压缩形式：如`(::({hex_group}(:{hex_group}){0,6})?)`匹配以“::”开头的地址，`(:{hex_group}(:{hex_group}){6}::)`匹配以“::”结尾的地址，以及其他“::”在中间的各种情况。

(4) 两阶段混合校验策略

```
^{v6}$ {  
    if (is_canonical_ipv6(yytext)) {  
        indicator = 6;  
    }  
}
```

结构校验：`^{v6}$`正则表达式首先对输入字符串进行匹配。`^`和`$`确保了整个字符串被完全匹配，不多也不少。

逻辑校验：一旦结构校验通过，规则的动作部分`{...}`就会被执行。调用 C 函数 `is_canonical_ipv6()`，并将匹配到的字符串 `yytext` 传递给它，进行更深层次的、正则表达式难以处理的逻辑校验。

(5) C 辅助函数 `is_canonical_ipv6`

特殊用例处理

```
if (strcmp(ip, "::") == 0 || strcmp(ip, "::1") == 0) {  
    return false;  
}
```

“规范化”表示法校验

用于处理 2001:db8:85a3:0::8a2E:0370:7334 这样的情况。测试要求“::”的压缩必须最高效，不允许“::”旁边紧邻一个全为“0”的段。

4.2 编译运行

```
cd /root/lab2/ipaddr  
make ipaddr          # 编译  
apt-get install -y python3  # 安装 python  
python3 ip_test.py   # 运行测试脚本
```

```
root@dccccfaf6754c:~/lab2/ipaddr# make ipaddr  
flex lex.l  
gcc lex.yy.c --shared -fPIC -o libip.so  
root@dccccfaf6754c:~/lab2/ipaddr# python3 ip_test.py  
All tests passed!  
root@dccccfaf6754c:~/lab2/ipaddr# |
```

显示 All tests passed!表示通过全部的测试。