编译原理实验指导书

Compilers Principles Experiment Instruction Book

陈贺敏　王林

燕山大学软件工程系

实验 1 词法分析

1.1 实验目的

（1）理解有穷自动机及其应用。

（2）掌握 NFA 到 DFA 的等价变换方法、DFA 最小化的方法。

（3）掌握设计、编码、调试词法分析程序的技术和方法。

1.2 实验任务

编写一个程序对输入的源代码进行词法分析，并打印分析结果。（以下两个任务二选一）

1、借助词法分析工具 GNU Flex，编写一个对使用 C--语言书写的源代码进行词法分析（C--语言的文法参见附录 A），并使用 C 语言完成。

2、自己编写一个 PL/0 词法分析程序，语言不限。

1.3 实验内容

1.3.1 实验要求

你的程序要能够查出源代码中可能包含的词法错误：

词法错误（**错误类型 A**）：（以输入 C--源代码为例）即出现 C--词法中未定义的字符以及任何不符合 C--词法单元定义的字符；

1.3.2 输入格式

1、如果你选择的是任务 1，那么你的程序的输入是一个包含 C--源代码的文本文件，程序需要能够接收一个输入文件名作为参数。例如，假设你的程序名为 cc、输入文件名为 test1、程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行./cc test1 即可获得以 test1 作为输入文件的输出结果。

2、如果你选择的是任务 2，那么你的程序输入是一个包含 PL/0 源代码的文本文件，程序需要能够接收一个输入文件名作为参数，以获得相应的输出结果。

1.3.3 输出格式

实验一要求通过标准输出打印程序的运行结果。对于那些包含词法错误的输入文件，只要输出相关的词法有误的信息即可。在这种情况下，注意不要输出任何与语法树有关的内容。要求输出的信息包括错误类型、出错的行号以及说明文字，

其格式为：Error type [错误类型] at Line [行号]: [说明文字].

说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。请严格遵守实验要求中给定的错误分类（**即词法错误为错误类型 A**），否则将影响你的实验评分。**注意，**输入文件中可能会包含一个或者多个词法错误（但输入文件的同一行中保证不出现多个错误），你的程序需要将这些错误全部

报告出来，每一条错误提示信息在输出中单独占一行。

对于那些没有任何词法错误的输入文件，你的程序需要打印每一个词法单元的名称以及与其对应的词素，无需打印行号。词法单元名与相应词素之间以一个冒号和一个空格隔开。每一条词法单元的信息单独占一行。

表 1-1 词法单元的例子

词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	Pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个"之间，除"以外的任何字符	"core dumped"

表 1-1 给出了一些常见的词法单元、非正式描述的词法单元的模式，并给出了一些示例词素（以下若无特殊说明，词素就是最小词法单位）。下面说明上述概念在实际中是如何应用的。在 C 语句

printf(“Total = %d\n”,score);

中，printf 和 score 都是和词法单元 id 的模式匹配的词素，而“Total = %d\n”则是一个和 literal 匹配的词素。

在很多程序设计语言中，下面的类别覆盖了大部分或所有的词法单元：

1）关键字。一个关键字的类型就是该关键字本身。

2）运算符。它可以表示单个运算符，也可以像表 1-1 中的 comparison 那样，表示一类运算符。

3）标识符。一个表示所有标识符的词法单元。

4）常量。一个或多个表示常量的词法单元，比如数字和字面值字符串。

5）界符。每一个标点符号有一个词法单元，比如左右括号、逗号和分号。

1.3.4 测试环境

任务 1：你的程序将在如下环境中被编译并运行：

1) GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29；

2) GCC version 4.6.3；

3) GNU Flex version 2.5.35；

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基

本上不会出现兼容性方面的问题。注意，实验一的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在你的程序中使用它们。

1.3.5 提交要求

1) 任务 1：Flex 以及 C 语言（任务 2 可以是 Java、C、Python 等任意语言，但需针对 PL/0 语言）的可被正确编译运行的源程序。

2）一份实验报告，内容包括：实验目的；实验任务；实验内容：算法描述，程序结构，主要变量说明，程序清单，调试情况及各种情况运行结果截图等；心得体会。

1.3.6 样例

此部分样例是以**任务 1** 为例给出的，请仔细阅读样例，以加深对实验要求以及输出格式要求的理解。

说明：选择任务 2 的同学，可以参考任务 1 的样例，自行设计测试样例，以反映出正确输出。

样例 1：

输入（行号是为标识需要，并非样例输入的一部分，后同）：

```
1   int main()
2   {
3       int i = 1;
4       int j = ~i;
5   }
```

输出：

这个程序存在词法错误。第 4 行中的字符“~”没有在我们的 C--词法中被定义过，因此你的程序可以输出如下的错误提示信息：

```
Error type A at Line 4: Mysterious character "~".
```

样例 2：

输入：

```
1   int inc()
2   {
3       int   i;
4       i =100;
5   }
```

输出：

这个程序非常简单，也没有任何词法错误，因此你的程序需要输出每一个词法单元信息：

TYPE: int

ID: inc

LP: (

RP:)

LC: {

TYPE: int

ID: i

SEMI: ;

ID: i

RELOP: =

INT: 100

SEMI: ;

RC: }

1.4 实验指导

词法分析和语法分析这两块，可以说是在整个编译器当中被自动化得最好的部分。也就是说，即使没有任何的理论基础，在掌握了工具的用法之后，也可以在短时间内做出功能很全很棒的词法分析程序和语法分析程序。当然这并不意味着，词法分析和语法分析部分的理论基础并不重要。恰恰相反，这一部分被认为是计算机理论在工程实践中最成功的应用之一，对它的介绍也是编译理论课中的重点。但本节指导内容的重点不在于理论而在于工具的使用。

本节指导内容将主要介绍词法分析工具 GNU Flex。如前所述，完成实验一并不需要太多的理论基础，只要看完并掌握了本节的大部分内容即可完成实验一。

1.4.1 词法分析概述

词法分析程序的主要任务是将输入文件中的字符流组织成为词法单元流，在某些字符不符合程序设计语言词法规范时它也要有能力报告相应的错误。词法分析程序是编译器所有模块中唯一读入并处理输入文件中每一个字符的模块，它使得后面的语法分析阶段能在更高抽象层次上运作，而不必纠结于字符串处理这样的细节问题。

高级程序设计语言大多采用英文作为输入方式，而英文有个非常好的性质，就是它比较容易断词：相邻的英文字母一定属于同一个词，而字母与字母之间插入任何非字母的字符（如

空格、运算符等）就可以将一个词断成两个词。判断一个词是否符合语言本身的词法规范也相对简单，一个直接的办法是：我们可以事先开一张搜索表，将所有符合词法规范的字符串都存放在表中，每次我们从输入文件中断出一个词之后，通过查找这张表就可以判断该词究竟合法还是不合法。

正因为词法分析任务的难度不高，在实用的编译器中它常常是手工写成，而并非使用工具生成。例如，我们下面要介绍的这个工具 GNU Flex 原先就是为了帮助 GCC 进行词法分析而被开发出来的，但在 4.0 版本之后，GCC 的词法分析器已经一律改为手写了。不过，实验一如果选择使用工具来做，而词法分析程序生成工具所基于的理论基础，是计算理论中最入门的内容：**正则表达式（Regular Expression）**和**有限状态自动机（Finite State Automata）**。

一个正则表达式由特定字符串构成，或者由其它正则表达式通过以下三种运算得到：

1) **并运算（Union）**：两个正则表达式 r 和 s 的并记作 r | s，意为 r 或 s 都可以被接受。

2) **连接运算（Concatenation）**：两个正则表达式 r 和 s 的连接记作 rs，意为 r 之后紧跟 s 才可以被接受。

3) **Kleene 闭包（Kleene Closure）**：一个正则表达式 r 的 Kleene 闭包记作 r*，它表示：| r | rr | rrr | …。

有关正则表达式的内容在课本的理论部分讨论过。正则表达式之所以被人们所广泛应用，一方面是因为它在表达能力足够强（基本上可以表示所有的词法规则）的同时还易于书写和理解；另一方面也是因为判断一个字符串是否被一个特定的正则表达式接受可以做到非常高效（在线性时间内即可完成）。比如，我们可以将一个正则表达式转化为一个 **NFA（即不确定的有限状态自动机）**，然后将这个 NFA 转化为一个 **DFA（即确定的有限状态自动机）**，再将转化好的 DFA 进行化简，之后我们就可以通过模拟这个 DFA 的运行来对输入串进行识别了。具体的 NFA 和 DFA 的含义，以及如何进行正则表达式、NFA 及 DFA 之间的转化等，请参考课本的理论部分。这里我们仅需要知道，前面所述的所有转化和识别工作，都可以由工具自动完成。而我们所需要做的，仅仅是为工具提供作为词法规范的正则表达式。

1.4.2 GNU Flex 介绍

Flex 的前身是 Lex。Lex 是 1975 年由 Mike Lesk 和当时还在 AT&T 做暑期实习的 Eric Schmidt，共同完成的一款基于 Unix 环境的词法分析程序生成工具。虽然 Lex 很出名并被广泛 使用，但它的低效和诸多问题也使其颇受诟病。后来伯克利实验室的 Vern Paxson 使用 C 语言 重写 Lex，并将这个新的程序命名为 Flex（意为 Fast Lexical Analyzer Generator）。无论在效 率上还是在稳定性上，Flex 都远远好于它的前辈 Lex。我们在 Linux 下使用的是 Flex

在 GNU License 下的版本，称作 GNU Flex。

GNU Flex 在 Linux 下的安装非常简单。你可以去它的官方网站上下载安装包自行安装，不过在基于 Debian 的 Linux 系统下，更简单的安装方法是直接在命令行敲入如下命令：

sudo apt-get install flex

虽然版本不一样，但 GNU Flex 的使用方法与课本上介绍的 Lex 基本相同。首先，我们需要自行完成包括词法规则等在内的 Flex 代码。至于如何编写这部分代码我们在后面会提到，现在先假设这部分写好的代码名为 lexical.l。随后，我们使用 Flex 对该代码进行编译：

flex lexical.l

编译好的结果会保存在当前目录下的 lex.yy.c 文件中。打开这个文件你就会发现，该文件本质上就是一份 C 语言的源代码。这份源代码里目前对我们有用的函数只有一个，叫做 yylex()，该函数的作用就是读取输入文件中的一个词法单元。我们可以再为它编写一个 main 函数：

```
1  int main(int argc, char** argv) {
2    if (argc > 1) {
3      if (!(yyin = fopen(argv[1], "r"))) {
4        perror(argv[1]);
5        return 1;
6      }
7    }
8    while (yylex() != 0);
9    return 0;
10 }
```

这个 main 函数通过命令行读入若干个参数，取第一个参数为其输入文件名并尝试打开该输入文件。如果文件打开失败则退出，如果成功则调用 yylex()进行词法分析。其中，变量 yyin 是 Flex 内部使用的一个变量，表示输入文件的文件指针，如果我们不去设置它，那么 Flex 会将它自动设置为 stdin（即**标准输入**，通常连接到键盘）。注意，如果你将 main 函数独立设为一个文件，则需要声明 yyin 为外部变量：extern FILE* yyin。

将这个 main 函数单独放到一个文件 main.c 中（你也可以直接放入 lexical.l 中的用户自定义代码部分，这样就可以不必声明 yyin；你甚至可以不写 main 函数，因为 Flex 会自动给你配一个，但不推荐这么做），然后编译这两个 C 源文件。我们将输出程序命名为 scanner：

gcc main.c lex.yy.c -lfl -o scanner

注意编译命令中的“-lfl”参数不可缺少，否则 GCC 会因为缺少库函数而报错。之后我们就可以使用这个 scanner 程序进行词法分析了。例如，想要对一个测试文件 test.cmm 进行词法分析，只需要在命令行输入：

./scanner test.cmm

，就可以得到你想要的结果了。

1.4.3 Flex：编写源代码

以上介绍的是使用 Flex 创建词法分析程序的基本步骤。在整个创建过程中，最重要的文件无疑是你所编写的 Flex 源代码，它完全决定了你的词法分析程序的一切行为。接下来我们介绍如何编写 Flex 源代码。

Flex 源代码文件包括三个部分，由“%%”隔开，如下所示：

```
1  {definitions}
2  %%
3  {rules}
4  %%
5  {user subroutines}
```

第一部分为**定义部分**，实际上就是给某些后面可能经常用到的正则表达式取一个别名，从而简化词法规则的书写。定义部分的格式一般为：

name definition

其中 name 是名字，definition 是任意的正则表达式（正则表达式该如何书写后面会介绍）。例如，下面的这段代码定义了两个名字：digit 和 letter，前者代表 0 到 9 中的任意一个数字字符，后者则代表任意一个小写字母、大写字母或下划线：

```
1  …
2  digit [0-9]
3  letter [_a-zA-Z]
4  %%
5  …
6  %%
7  …
```

Flex 源代码文件的第二部分为**规则部分**，它由正则表达式和相应的响应函数组成，其格式为：

pattern {action}

其中 pattern 为正则表达式，其书写规则与前面的定义部分的正则表达式相同。而 action 则为将要进行的具体操作，这些操作可以用一段 C 代码表示。Flex 将按照这部分给出的内容依次尝试每一个规则，尽可能匹配最长的输入串。如果有些内容不匹配任何规则，Flex 默认只将其拷贝到标准输出，想要修改这个默认行为的话只需要在所有规则的最后加上一条“.”（即匹配任何输入）规则，然后在其对应的 action 部分书写你想要的行为即可。

例如，下面这段 Flex 代码在遇到输入文件中包含一串数字时，会将该数字串转化为整数值并打印到屏幕上：

```
1  …
2  digit [0-9]
3  %%
4  {digit}+ { printf("Integer value   %d\n", atoi(yytext)); }
5  …
6  %%
7  …
```

其中变量 yytext 的类型为 char*，它是 Flex 为我们提供的一个变量，里面保存了当前词法单元所对应的词素。函数 atoi()的作用是把一个字符串表示的整数转化为 int 类型。

Flex 源代码文件的第三部分为**用户自定义代码部分**。这部分代码会被原封不动地拷贝到 lex.yy.c 中，以方便用户自定义所需要执行的函数（之前我们提到过的 main 函数也可以写在这里）。值得一提的是，如果用户想要对这部分所用到的变量、函数或者头文件进行声明，可以在前面的定义部分（即 Flex 源代码文件的第一部分）之前使用“%{”和“%}”符号将要声明的内容添加进去。被“%{”和“%}”所包围的内容也会一并拷贝到 lex.yy.c 的最前面。

下面通过一个简单的例子来说明 Flex 源代码该如何书写 1。我们知道 Unix/Linux 下有一个常用的文字统计工具 wc，它可以统计一个或者多个文件中的（英文）字符数、单词数和行数。利用 Flex 我们可以快速地写出一个类似的文字统计程序：

```
1  %{
2      /*  此处省略#include 部分  */
3        int chars = 0;
4        int words = 0;
5        int lines = 0;
6  %}
7  letter [a-zA-Z]
```

```
%%
{letter}+ { words++; chars+= yyleng; }
\n { chars++; lines++; }
. { chars++; }
%%
int main(int argc, char** argv) {
    if (argc > 1) {
      if (!(yyin = fopen(argv[1], "r"))) {
          perror(argv[1]);
          return 1;
      }
    }
    yylex();
    printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
}
```

其中 yyleng 是 Flex 为我们提供的变量，你可以将其理解为 strlen(yytext)。我们用变量 chars 记录输入文件中的字符数、words 记录单词数、lines 记录行数。上面这段程序很好理解：每遇到一个换行符就把行数加一，每识别出一个单词就把单词数加一，每读入一个字符就把字符数加一。最后在 main 函数中把 chars、words 和 lines 的值全部打印出来。需要注意的是，由于在规则部分里我们没有让 yylex()返回任何值，因此在 main 函数中调用 yylex()时可以不套外层的 while 循环。

真正的 wc 工具可以一次传入多个参数从而统计多个文件。为了能够让这个 Flex 程序对多个文件进行统计，我们可以修改 main 函数如下：

```
int main(int argc, char** argv) {
    int i, totchars = 0, totwords = 0, totlines = 0;
    if (argc < 2) { /* just read stdin */
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
        return 0;
    }
```

```
8       for (i = 1; i < argc; i++) {
9           FILE *f = fopen(argv[i], "r");
10          if (!f) {
11              perror(argv[i]);
12              return 1;
13          }
14          yyrestart(f);
15          yylex();
16          fclose(f);
17          printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
18          totchars += chars; chars = 0;
19          totwords += words; words = 0;
20          totlines += lines; lines = 0;
21      }
22      if (argc > 1)
23        printf("%8d%8d%8d total\n", totlines, totwords, totchars);
24      return 0;
25  }
```

其中 yyrestart(f)函数是 Flex 提供的库函数，它可以让 Flex 将其输入文件的文件指针 yyin 设置为 f（当然你也可以像前面一样手动设置令 yyin = f）并重新初始化该文件指针，令其指向输入文件的开头。

1.4.4 Flex：书写正则表达式

Flex 源代码中无论是定义部分还是规则部分，正则表达式都在其中扮演了重要的作用。那么，如何在 Flex 源代码中书写正则表达式呢？我门下面介绍一些规则：

1) 符号“.”匹配除换行符“\n”之外的任何一个字符。

2) 符号“[”和“]”共同匹配一个字符类，即方括号之内只要有一个字符被匹配上了，那么方括号括起来的整个表达式都被匹配上了。例如，[0123456789]表示 0~9 中任意一个数字字符，[abcABC]表示 a、b、c 三个字母的小写或者大写。方括号中还可以使用连字符“-”表示一个范围，例如[0123456789]也可以直接写作[0-9]，而所有小写字母字符也可直接写成[a-z] 。如果方括号中的第一个字符是“^”，则表示对这个字符类取补，即方括号之内如果没有任何一个字符被匹配上，那么被方括号括起来的整个表达式就认为被匹配上了。例如，

[^_0-9a-zA-Z]表示所有的非字母、数字以及下划线的字符。

3) 符号“^”用在方括号之外则会匹配一行的开头，符号“$”用于匹配一行的结尾，符号“<<EOF>>”用于匹配文件的结尾。

4) 符号“{”和“}”含义比较特殊。如果花括号之内包含了一个或者两个数字，则代表花括号之前的那个表达式需要出现的次数。例如，A{5}会匹配 AAAAA，A{1,3}则会匹配 A、AA 或者 AAA。如果花括号之内是一个在 Flex 源代码的定义部分定义过的名字，则表示那个名字对应的正则表达式。例如，在定义部分如果定义了 letter 为[a-zA-Z]，则{letter}{1,3}表示连续的一至三个英文字母。

5) 符号 “*” 为 **Kleene 闭包**操作，匹配零个或者多个表达式。例如{letter}*表示零个或者多个英文字母。

6) 符号 “+” 为**正闭包**操作，匹配一个或者多个表达式。例如{letter}+表示一个或者多个英文字母。

7) 符号“?”匹配零个或者一个表达式。例如表达式-?[0-9]+表示前面带一个可选的负号的数字串。无论是*、+还是?，它们都只对其最邻近的那个字符生效。例如 abc+表示 ab 后面跟一个或多个 c，而不表示一个或者多个 abc。如果你要匹配后者，则需要使用小括号“(”和“)” 将这几个字符括起来：(abc)+。

8) 符号“|”为选择操作，匹配其之前或之后的任一表达式。例如，faith | hope | charity 表示这三个串中的任何一个。

9) 符号“\”用于表示各种转义字符，与 C 语言字符串里“\”的用法类似。例如，“\n”表示换行，“\t”表示制表符，“*” 表示星号，“\\” 代表字符 “\” 等。

10) 符号 “"”（英文引号）将逐字匹配被引起来的内容（即无视各种特殊符号及转义字符）。例如，表达式"..."就表示三个点而不表示三个除换行符以外的任意字符。

11) 符号 “/” 会查看输入字符的上下文，例如，x/y 识别 x 仅当在输入文件中 x 之后紧跟着 y，0/1 可以匹配输入串 01 中的 0 但不匹配输入串 02 中的 0。

12) 任何不属于上面介绍过的有特殊含义的字符在正则表达式中都仅匹配该字符本身。

下面我们通过几个例子来练习一下 Flex 源代码里正则表达式的书写：

1) 带一个可选的正号或者负号的数字串，可以这样写：[+-]?[0-9]+。

2) 带一个可选的正号或者负号以及一个可选的小数点的数字串，表示起来要困难一些，可以考虑下面几种写法：

a) [+-]?[0-9.]+会匹配太多额外的模式，像 1.2.3.4；

b) [+-]?[0-9]+\.?[0-9]+会漏掉某些模式，像 12.或者.12；

c) [+-]?[0-9]*\.?[0-9]+会漏掉 12.；

d) [+-]?[0-9]+\.?[0-9]*会漏掉.12；

e) [+-]?[0-9]*\.?[0-9]*会多匹配空串或者只有一个小数点的串；

f) 正确的写法是：[+-]?([0-9]*\.?[0-9]+|[0-9]+\.)。

3) 假设我们现在做一个汇编器，目标机器的 CPU 中有 32 个寄存器，编号为 0…31。汇编源代码可以使用 r 后面加一个或两个数字的方式来表示某一个寄存器，例如 r15 表示第 15 号寄存器，r0 或 r00 表示第 0 号寄存器，r7 或者 r07 表示第 7 号寄存器等。现在我们希望识别汇编源代码中所有可能的寄存器表示，可以考虑下面几种写法：

g) r[0-9]+可以匹配 r0 和 r15，但它也会匹配 r99999，目前世界上还不存在 CPU 能拥有一百万个寄存器。

h) r[0-9]{1,2}同样会匹配一些额外的表示，例如 r32 和 r48 等。

i) r([0-2][0-9]?|[4-9]|(3(0|1)?))是正确的写法，但其可读性比较差。

j) 正确性毋庸置疑并且可读性最好的写法应该是：r0 | r00 | r1 | r01 | r2 | r02 | r3 | r03 | r4 | r04 | r5 | r05 | r6 | r06 | r7 | r07 | r8 | r08 | r9 | r09 | r10 | r11 | r12 | r13 | r14 | r15 | r16 | r17 | r18 | r19 | r20 | r21 | r22 | r23 | r24 | r25 | r26 | r27 | r28 | r29 | r30 | r31，但这样写的话可扩展性又非常差，如果目标机器上有 128 甚 至 256 个寄存器呢？

1.4.5 Flex：高级特性

前面介绍的 Flex 内容已足够帮助完成实验一的词法分析部分。下面介绍一些 Flex 里面的高 级特性，能让你在使用 Flex 的过程中感到更方便和灵活。这部分内容是可选的，跳过也不会对实验一的完成产生负面的影响。

yylineno 选项：

在写编译器程序的过程中，经常会需要记录行号，以便在报错时提示输入文件的哪一行出现了问题。为了能记录这个行号，我们可以自己定义某个变量，例如 lines，来记录词法分析程序当前读到了输入文件的哪一行。每当识别出“\n”，我们就让 lines = lines + 1。

实际上，Flex 内部已经为我们提供了类似的变量，叫做 yylineno。我们不必去维护 yylineno 的值，它会在每行结束自动加一。不过，默认状态下它并不开放给用户使用。如果我们想要读取 yylineno 的值，则需要在 Flex 源代码的定义部分加入语句“%option yylineno”。

需要说明的是，虽然 yylineno 会自动增加，但当我们在词法分析过程中调用 yyrestart() 函数读取另一个输入文件时它却不会重新被初始化，因此我们需要自行添加初始化语句 yylineno = 1。

输入缓冲区：

课本上介绍的词法分析程序其工作原理都是在模拟一个 DFA 的运行。这个 DFA 每次读入一个字符，然后根据状态之间的转换关系决定下一步应该转换到哪个状态。事实上，实用的词法分析程序很少会从输入文件中逐个读入字符，因为这样需要进行大量的磁盘操作，效率较低。更加高效的办法是一次读入一大段输入字符，并将其保存在专门的输入缓冲区中。

在 Flex 中，所有的输入缓冲区都有一个共同的类型，叫做 YY_BUFFER_STATE。你可以通过 yy_create_buffer()函数为一个特定的输入文件开辟一块输入缓冲区，例如：

```
1  YY_BUFFER_STATE bp;
2  FILE* f;
3  f = fopen(…, "r");
4  bp = yy_create_buffer(f, YY_BUF_SIZE);
5  yy_switch_to_buffer(bp);
6  …
7  yy_flush_buffer(bp);
8  …
9  yy_delete_buffer(bp);
```

其中 YY_BUF_SIZE 是 Flex 内部的一个常数，通过调用 yy_switch_to_buffer()函数可以让词法分析程序到指定的输入缓冲区中读字符，调用 yy_flush_buffer()函数可以清空缓冲区中的内容，而调用 yy_delete_buffer()则可以删除一个缓冲区。

如果你的词法分析程序要支持文件与文件之间的相互引用（例如 C 语言中的#include），

你可能需要在词法分析的过程中频繁地使用 yyrestart()切换当前的输入文件。在切换到其他输入文件再切换回来之后，为了能继续之前的词法分析任务，你需要无损地保留原先输入缓冲区的内容，这就需要使用一个栈来暂存当前输入文件的缓冲区。虽然 Flex 也有提供相关的函数来帮助你做这件事情，但这些函数的功能比较弱，建议自己手写更好。

Flex 库函数 input：

Flex 库函数 input()可以从当前的输入文件中读入一个字符，这有助于你不借助正则表达式来实现某些功能。例如，下面这段代码在输入文件中发现双斜线“//”后，将从当前字符开始一直到行尾的所有字符全部丢弃掉：

```
1  %%
2  "//" {
3  char c = input();
```

```
4    while (c != '\n') c = input();
5    }
```

Flex 库函数 unput：

Flex 库函数 unput(char c)可以将指定的字符放回输入缓冲区中，这对于宏定义等功能的实现是很方便的。例如，假设之前定义过一个宏#define BUFFER_LEN 1024，当在输入文件中遇到字符串 BUFFER_LEN 时，下面这段代码将该宏所对应的内容放回输入缓冲区：

```
1    char* p = macro_contents("BUFFER_LEN"); // p = “1024”
2    char* q = p + strlen(p);
3    while (q > p) unput(*--q); // push back right-to-left
```

Flex 库函数 yyless 和 yymore：

Flex 库函数 yyless(int n)可以将刚从输入缓冲区中读取的 yyleng　n 个字符放回到输入缓冲区中，而函数 yymore()可以告诉 Flex 保留当前词素，并在下一个词法单元被识别出来之后将下一个词素连接到当前词素的后面。配合使用 yyless()和 yymore()可以方便地处理那些边界难以界定的模式。例如，我们在为字符串常量书写正则表达式时，往往会写成由一对双引号引起来的所有内容\"[^\"]*\"，但有时候被双引号引起来的内容里面也可能出现跟在转义符号之后的双引号，例如"This is an \"example\""。那么如何使用 Flex 处理这种情况呢？方法之一就是借助于 yyless 和 yymore：

```
1    %%
2    \"[^\"]*\" {
3        if (yytext[yyleng - 2] == '\\') {
4            yyless(yyleng - 1);
5            yymore();
6        } else {
7            /* process the string literal */
8        }
9    }
```

Flex 宏 REJECT：

Flex 宏 REJECT 可以帮助我们识别那些互相重叠的模式。当我们执行 REJECT 之后，Flex 会进行一系列的操作，这些操作的结果相当于将 yytext 放回输入之内，然后去试图匹配当前规则之后的那些规则。例如，考虑下面这段 Flex 源代码：

```
1  %%
2  pink { npink++; REJECT; }
3  ink { nink++; REJECT; }
4  pin { npin++; REJECT; }
```

这段代码会统计输入文件中所有的 pink、ink 和 pin 出现的个数，即使这三个单词之间互有重叠。

Flex 还有更多的特性，感兴趣的读者可以参考其用户手册。

1.4.6 词法分析提示

如果你选择任务 1，为了完成实验一，首先需要阅读 C--语言文法（附录 A），包括其文法定义和补充说明。除了 INT、FLOAT 和 ID 这三个词法单元需要你自行为其书写正则表达式之外，剩下的词法单元都没有难度。

阅读完 C--语言文法，对 C--的词法有大概的了解之后，就可以开始编写 Flex 源代码了。在敲入所有的词法之后，为了能检验你的词法分析程序是否工作正常，你可以暂时向屏幕打印

当前的词法单元的名称，例如：

```
1  %%
2  "+" { printf(“PLUS\n”); }
3  "-" { printf(“SUB\n”); }
4  "&&" { printf(“AND\n”); }
5  "||" { printf(“OR\n”); }
6  …
```

为了能够报告错误类型 A，你可以在所有规则的最后增加类似于这样的一条规则：

```
1 %%
2 …
3  .  {
4          printf("Error type A at Line %d: Mysterious characters \'%s\'\n",
5                  yylineno, yytext);
6      }
```

完成 Flex 源代码的编写之后，使用前面介绍过的方法将其编译出来，就可以自己书写一些小规模的输入文件来测试你的词法分析程序了。一定要确保你的词法分析程序的正确性！如果词法分析这里出了问题没有检查出来，到了后面语法分析发现了前面的问题再回头

调试，那将增加许多不必要的麻烦。为了使你在编写 Flex 源代码时少走弯路，以下是几条建议：

1) 留意空格和回车的使用。如果不注意，有时很容易让本应是空白符的空格或者回车变成正则表达式的一部分，有时又很容易让本应是正则表达式一部分的空格或回车变成 Flex 源代码里的空白符。

2) 正则表达式和其所对应的动作之间，永远不要插入空行。

3) 如果对正则表达式中的运算符优先级有疑问，那就不要吝啬使用括号来确保正则表达式的优先级确实是你想要的。

4) 使用花括号括起每一段动作，即使该动作只包含有一行代码。

5) 在定义部分我们可以为许多正则表达式取别名，这一点要好好利用。别名可以让后面的正则表达式更加容易阅读、扩展和调试。

6) 在正则表达式中引用之前定义过的某个别名（例如 digit）时，时刻谨记该别名一定要用花括号“{”和“}”括起来。